Diagram Predicate Framework

A Formal Approach to MDE

ADRIAN RUTLE

Diagram Predicate Framework

A Formal Approach to MDE

ADRIAN RUTLE

Dissertation for the degree of philosophiae doctor (PhD)
at the University of Bergen

September, 2010

ISBN 978-82-308-1650-9
University of Bergen, Norway, 2010-09-02 (print version 2010-12-10).

Printed by AIT.

Prepared with LaTeX and set in *times* with program code in `txtt`.

To my parents
and
to Liva

Scientific Environment

The research presented in this dissertation has been conducted at the Department of Computer Engineering, Bergen University College, in cooperation with the Programming Theory Group of the Department of Informatics at the University of Bergen.

Contents

Preface

Finally! I have looked forward to this moment for more than six months. A PhD student who kindly gave me the LaTeX sources, styles, etc for her thesis, wrote something about PhD dissertations not being read by so many people, and expressed her frustration about that, since the amount of time spent in writing a document like this deserves some readers. I could not agree more, therefore, I keep this modified version of her preface – and frustration – here.

The past four years of my life have gone to write down the content of this document. Many late nights (and late mornings) and several weekends had to be devoted to make this document as perfect as it is. I should admit that these years have also been very rewarding, interesting and fun. So if you have received a nice, printed copy of this thesis and you are actually going to read it, I hope you will enjoy it and find it interesting. Otherwise, I hope you appreciate it, and that it will look good on your bookshelf.

Bergen, 2010-08-31

ACKNOWLEDGEMENTS

It was impossible to write this first sentence without mentioning the names Uwe Wolter and Yngve Lamo, my supervisors who have dedicated years of their life to support me in this journey. They deserve much of the credit for this PhD dissertation. They have been available in weekdays and weekends, and supported me when we got papers accepted and when we got papers rejected, and when I drove right, wrong, right, left, or straight forward (to Keflavik). I thank them for being excellent mentors, and for all the nice memories, all the good knowledge they have taught me, as well as all the interesting discussions we have had – even though some of them were (mis?)understood as fighting by other people at the HIB canteen.

Who else to thank if not Alessandro Rossini. Although sometimes I think his formalisation of everything has traumatised me, I am in deep gratitude for all the support I have got from him. He has been a brilliant colleague, an excellent friend and a perfect travel companion. He has taught me things which I could never learn from anyone else.

Thanks to my family for showing patience, support and understanding during this period, my parents, my sister, my brothers, Kari and Kristoffer.

I spent most of my spare time with my dearest friends Christer, Thorbjørn, Kent Inge, Dag Viggo and Jana. Thanks to Christer and Thorbjørn for all the wonderful weekends and all the less wonderful Brann matches we have experienced together. Thanks to Dag Viggo and Kent Inge for all the good concerts, the long discussions about music, and the important things they have taught me, especially about Trappist beers. Thanks to Jana for patiently and faithfully waiting for me all this time, despite everything else I had to do to finish this dissertation.

I also thank the rock climbing group and the alpine group for many wonderful days together at the climbing walls, and the skiing slopes, especially, Camilla Hosfeld Diesen, who introduced me to rock climbing. Also, Henning Klafstad and Harald Moen, for good advices and late night discussions, also Marianne Mathiesen, Anette Olene Fristad, Liv Kjelleberg, Annicka Langeland, Karin Berg and Siv (princess) Birgitta Systad for being at the other end of the rope.

Many thanks to all others who have been involved in this project, Zinovy Diskin for good conversations and suggestions on the research direction, Gabriele Taentzer for teaching me graph transformations, Florian Mantz for, yes, many interesting discussions and good friendship both when I was abroad and now, the Master students Ørjan Hatland, Stian Skjerveggen, Øyvind Bech and Dag Viggo Lokøen. And all others who have reviewed this thesis and given comments.

Special thanks also to colleagues with whom I have shared office and my lunch breaks, Yi Wang, Hege Erdal, Piotr Kazmiercak and Sami Taktak.

I have been so lucky to be part of two Universities and two research groups, I have got the best from both sides, both when it comes to good colleagues, and when it comes to travel money. I would like to thank the Department of Computer Engineering at Bergen University College, and Department of Informatics at the University of Bergen for supporting me during my PhD period. Special thanks also to the DISTECH project members, especially Thomas Ågotnes and Lars Michael Kristensen. Thanks to Terje Kristensen, for always being a good listener and an excellent adviser. Thanks also to Ida Holen, for being very helpful and always making the bureaucratic matters easy for me, and Dag Hovland, for several "quiet" lunch breaks at the HIB canteen. At last but not the least, thanks to Anya Helene Bagge for giving me this beautiful LaTeX style, and for her inspiration in writing this preface and acknowledgement.

Thanks, finally, to my opponents Gabriele Taentzer and Øystein Haugen, for all the time they have spent reviewing this work – it is much appreciated; and to Marc Bezem for leading the committee. Thanks to those who reviewed my papers and this thesis, especially Barbara Joan Blair for excellent feedback on my language. I am also thankful to all the anonymous reviewers who pointed out flaws, possible improvements and suggested new directions in my work. This also goes for the feedback on the conference paper presentations.

Abstract

Model-driven engineering (MDE) is a software engineering discipline which promotes models as first-class entities. It represents a shift of paradigm in software development, from being code-centric to become model-centric. MDE is an attempt to organise modelling, metamodelling and model transformation in a well-structured engineering methodology. This thesis is all about formalisation of MDE-concepts in a diagrammatic specification formalism which we call Diagram Predicate Framework (DPF). DPF provides a formal diagrammatic approach to (meta)modelling and model transformation based on category theory. It is a generic graph-based specification framework that tends to adapt first-order logic and categorical logic to software engineering needs.

This thesis is based on a sequence of publications and it is the intended purpose of this thesis to consolidate the present state of development regarding DPF. Some of the foundation for DPF was already under construction before this work was initiated. The main contributions of this thesis are:

- An introduction to formal diagrammatic modelling and diagrammatic constraints
- A comparison of some of the state-of-the-art modelling languages, techniques and frameworks to DPF
- A neat, diagrammatic formalisation of the metamodelling hierarchy as proposed by the Object Management Group
- A formal approach to model transformations and to constraint-awareness in model transformation
- A formalisation of the fundamental concepts and processes of version control in the context of MDE

This thesis is organised as follows. The first chapter is dedicated to introduce MDE, its technological basis and challenges, and to motivate the formalisation approach presented in this thesis. This introduction is meant as a guide for newcomers to MDE, especially for theoreticians. The main part of the thesis details DPF and its formal background. This part is more theoretically oriented, and is meant to elucidate the formal foundation of the DPF framework for software engineers. More precisely, DPF is presented as a formal approach to (meta)modelling, model transformation and version control. The last chapter is dedicated to a discussion of related work, further work and concluding remarks.

The content of this thesis is neither purely theoretical nor purely practical; rather it seeks to bridge the gap between these worlds. It provides a formal approach to diagrammatic modelling, model transformation and version control motivated and illustrated by practical examples. We introduce only the theoretical elements which are necessary to investigate, formalise, and to solve the practical problems. More precisely, we explicitly define the formal concepts and constructions needed in order to understand the thesis, such as graph, graph homomorphism, categories, pullback and pushout.

CHAPTER 1

Introduction to Model-Driven Engineering

In this chapter model-driven engineering (MDE) will be introduced along with a discussion regarding some of its main concepts, techniques and standards. In addition, an outline of challenges related to the state-of-the-art of MDE will be presented.

1.1 Introduction

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

EVOLUTION OF PROGRAMMING

Initially, models were adopted in software development processes for sketching the architectural design or documenting an existing implementation. In contrast, the latest trend in software engineering regards models as first-class entities of the development process. These models are used to automatically generate (parts of) software systems by means of model-to-model and model-to-code transformations. This trend has led to a branch of software engineering which promotes modelling as the main activity of software development and 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 use the term MDE.

MODEL-DRIVEN ENGINEERING

The advantages of MDE are many. MDE enhances productivity and quality by automating repetitive, error-prone and time-consuming tasks. Moreover, MDE improves communication by exploiting abstraction and domain-specificity which can target different audiences. In addition, it facilitates the separation of business logic from application technologies. Computing infrastructures are continuously expanding in every dimension in response to business needs and technological development. By adopting MDE, business logic and application technologies can evolve independently of each other and organisations are able to integrate existing systems with those being built in the future [93].

ADVANTAGES OF MDES

The reference industrial implementation of MDE is the model-driven architecture (MDA), which was initiated by the Object Management Group (OMG) [84] late in 2000 [47; 68; 85; 93]. The basic ideas of MDA are closely related [17] to generative programming, software factories [53], domain-specific modelling languages [73], etc. MDA is based on multiple standards, including the Unified Modeling Language (UML) [91], the Meta-Object Facility (MOF) [87] and XML Metadata Interchange (XMI) [89].

MDA

1.2 Concepts in MDE

In this section, central concepts in MDE will be introduced and explained, starting with the very concept of model.

MEANING OF MODEL

The word *model* has different meanings in different contexts. In the Cambridge Dictionaries Online [25], one of the definitions of the word 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”. This definition corresponds to the way models are used in most engineering disciplines. In software engineering, a model is an abstraction in the sense that it may not represent all aspects and properties of the real system [17; 102], but only those which are relevant in the given context. Models are used to tackle the complexity of software by enabling developers to reason about and to deal with a software system at a higher level of abstraction.

MODELS IN FORMAL SPECIFICATIONS

In contrast, in formal specifications such as formal logic and universal algebra, a system is represented by a specification, i.e. a set of logical formulae. A model of such a specification consists of a mathematical structure that satisfies these formulae. Thus, *formal specifications* correspond to *models* in terms of software modelling. Examples of formal specification techniques are the Z notation [7; 64; 129], the Vienna Development Method’s Specification Language (VDM-SL) [62; 65] and the Abstract Machine Notation (AMN) of the B-Method [61; 116]. In this thesis, we interpret the word model from the software engineering perspective.

DIAGRAMS

In software engineering, models are often diagrammatic. The word *diagram* has also different meanings in different contexts. In Dictionary.com [9], one of the definitions of the word diagram is “a drawing or plan that outlines and explains the parts, operation, etc., of something”; e.g. chart diagrams and cake diagrams. In software engineering, the same word denotes structures which are based on graphs, i.e. a collection of nodes together with a collection of arrows between nodes. Graphs are a well-known, well-understood and frequently used

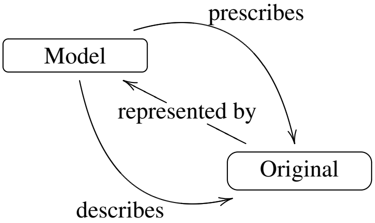


Figure 1.1: A model may describe or prescribe an original

means to represent structural or behavioural properties of a software system [41]; e.g. Entity-Relationship (ER) diagrams [27]. In contrast, in mathematics a diagram has a precise meaning as it denotes a graph homomorphism from a shape graph into a graph [45]. In this thesis, we interpret the word diagram from the software engineering perspective.

VISUAL VS DIAGRAMMATIC

Since graph-based structures are often visualised in a natural way, *visual* and *diagrammatic* modelling are often treated as synonyms. In this thesis, we distinguish clearly between visualisation and diagrammatic syntax and focus on precise syntax (and semantics) of diagrammatic models independent of their visualisation. By visualisation we mean rendering a model perceptible and intuitive for humans, while by diagrammatic modelling we mean techniques targeting graph-based structures. Although it is feasible to visualise graph-based structures, it may be a challenging task, and sometimes even impossible, to find appropriate and intuitive visualisations for all aspects of diagrammatic models.

DESCRIPTIVE VS PRESCRIPTIVE

A general categorisation of models is whether they describe or prescribe originals, or both. A descriptive model is used to describe an existing original, e.g. a map of a real city with streets, buildings, etc. On the contrary, a prescriptive model captures 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: models are used to represent relevant aspects of a real system, and later on used to drive the implementation of the real system based on these models.

ORIGINALS

There are also a variety of names used for modelled artefacts, such as real system, original, subject under study (SUS), etc. [17; 18; 50; 59; 91; 111]. In this thesis, we use the term *original* to denote everything that might be subject to modelling.

1.3 Diagrammatic Modelling

Diagrammatic models have already been around in software engineering for some decades; e.g. Flowcharts (Seventies) for the description of behavioural properties of software systems; Petrinets (Eighties) for the representation of discrete distributed systems; ER diagrams (Eighties) for the conceptual representation of data structures, UML diagrams (Nineties) for the representation of structural and behavioural

Table 1.1: Advantages of diagrammatic modelling

Property	**Advantage**	**Achieved by**
Documentation and communication	Facilitating intuitiveness	*Visual models*
Abstraction	Independence of the implementation platform	*Abstract models* and *model transformations*
Validation and verification	Revealing errors and flaws before the system is implemented	*Formal models* and *model checking*

DIAGRAMMATIC MODELLING

properties of software systems. Diagrammatic models have become popular because they facilitate the conception of (aspects of) a software system at a high level of abstraction while programming languages do not. Some of the advantages of diagrammatic modelling are summarised in Table 1.1.

CONCEPTUAL TWO-DIMENSIONALITY

Another factor which has helped in popularisation of diagrammatic modelling is the conceptual two-dimensionality of the modelled universes; e.g. nodes and edges, *Classes* and *Associations*, *Entities* and *Relations*, *States* and *Transitions*, *Objects* and *Links*, etc [38]. Each of these conceptual models may be represented by graphs or graph-based structures.

UML

Several modelling languages have emerged in the last years as attempts to facilitate MDE. In the state-of-the-art of MDE, models are often specified by means of the UML. UML consists of a set of languages which are used to describe or specify various aspects of software systems; such as system structure in class diagrams, system behaviour in activity and use-case diagrams, etc. The next example illustrates usage of UML class diagrams to specify the structure of a software system.

Example 1 (A UML Class Diagram) Assume that we want to specify an information system for the management of employees and departments in which the following set of requirements are satisfied at any state of the system:

1. An employee must work for at least one department.
2. A department may have zero or many employees.

Fig. 1.2 shows a UML class diagram which expresses the requirements above. Note that a UML class diagram consists primarily of a graph with nodes representing *class*es (or concepts) and edges representing *association*s (or relations) between classes. ♢

The model in Example 1 may be used as documentation or as communication basis between developers and stakeholders. However, in the context of MDE,

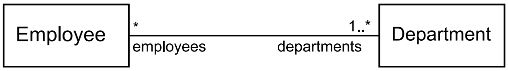


Figure 1.2: A UML class diagram for management of employees and departments

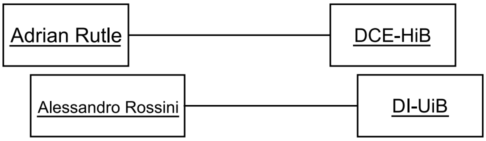


Figure 1.3: A UML object diagram for the software system in Fig 1.2

models are also used as starting points for driving the implementation of software systems. One of the main goals of MDE is to automate this step. As a consequence, the semantics of each model element needs to be agreed upon and defined formally. One approach is to define the semantics of a model as the set of its *instances*; i.e. by all structures that satisfy the requirements specified by the model. In UML, instances of a model are represented by informal snapshots (illustrations) of the running system which the model specifies. The UML models used to define these snapshots are called *UML object diagrams*.

INSTANCE

Example 2 (A UML Object Diagram) Recall Example 1. If we assume that a Java program is developed based on the UML class diagram, then the snapshots will represent the Java objects in memory. Fig. 1.3 shows a UML object diagram which represents some runtime Java objects and their relations. Note that a UML object diagram consists primarily of a graph with nodes representing *object*s and edges representing *link*s between objects. ♢

1.4 Metamodelling

We may define the term *metamodelling* as “the act and science of engineering metamodels” [50]. However, the precise definition of *metamodel* is abundantly debated in the literature (see [10; 17; 50; 59; 71; 72; 111] 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” [85]. Technically, a metamodel specifies the abstract syntax of a modelling language. The abstract syntax defines the set of modelling concepts, their attributes and their relationships, as well as the rules for combining these concepts to specify valid models [91]. That is, models which are specified by a modelling language should conform to the corresponding metamodel of the language. This means that a metamodel restricts the set of its instances in the same way a model restricts its instances, which is reflected in the definition “a model is an instance of a metamodel” [91].

METAMODEL

Recall that in Example 1 we stated that UML class diagrams consist primarily of classes and associations between classes. The specification of these *types*, i.e. classes and associations, is done in the metamodel of UML class diagrams. Moreover, the metamodel defines rules for how to combine classses and associations in

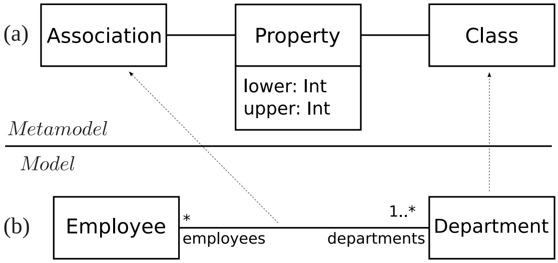


Figure 1.4: A simplified metamodel for UML class diagram

order to define (syntactically) correct class diagrams. The next example illustrates this.

Example 3 (A Simplified Metamodel for UML Class Diagrams) Fig. 1.4a shows a simplified metamodel for UML class diagrams. Some dashed, gray arrows are used to indicate the relation between the UML class diagram and the metamodel. ♢

METAMODELLING LANGUAGES

Metamodels, in turn, are specified by means of metamodelling languages. Metamodels which are specified by a metamodelling language are regarded as models which conform to the corresponding metamodel of the language. A metamodelling language is just a modelling language that is used for specifying models which, in turn, serve as the corresponding metamodels of other modelling languages (see Fig. 1.5). Following this line of reasoning, it is possible to identify a generic pattern which leads to a (meta)modelling hierarchy in which models at each level are

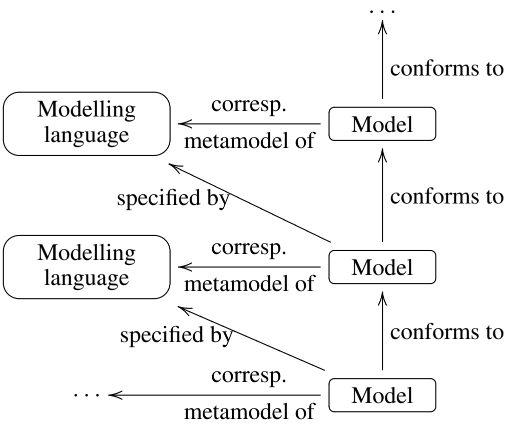


Figure 1.5: Generic pattern: modelling languages and (meta)models

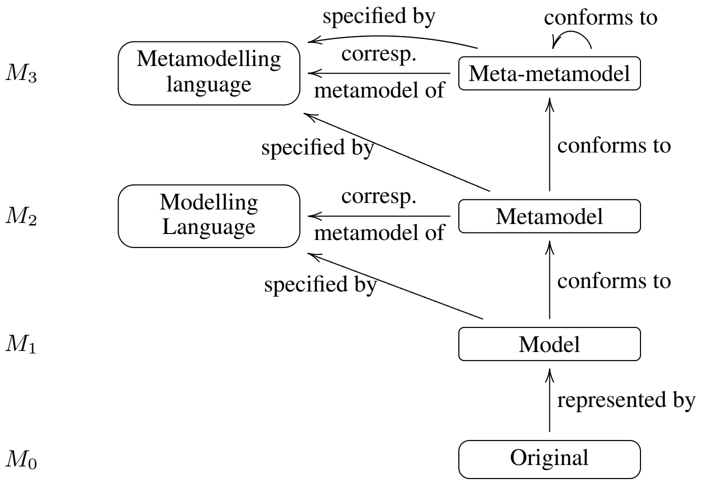


Figure 1.6: OMG's 4-layered hierarchy illustrating (meta)modelling languages and their corresponding metamodels

specified by a modelling language at the level above and conform to the corresponding metamodel of the language. Hence, a model at a certain level conforms to a metamodel at the level above and acts as a metamodel for models at the level below. Hypothetically, this pattern may continue *ad infinitum*. In practice, metamodelling hierarchies stop usually with a reflexive modelling language, which is a modelling language able to define its own metamodel. Besides this, any hierarchy will eventually reach a fixed-point.

OMG' 4-LAYERED HIERARCHY

In MDE, models are often specified by means of UML. The metamodel of UML is in turn specified by means of the MOF, which is a metamodelling and metadata repository standard developed by the OMG. According to the OMG's vision of MDE, models, modelling languages and metamodelling languages are organised in four levels $M_0 - M_3$ in the so-called OMG's 4-layered hierarchy [18; 91]. The characterisation of each level of the OMG's 4-layered hierarchy and the relations between them has been abundantly debated in the literature (see [10; 17; 18; 50; 59; 71; 72; 111] for a comprehensive discussion). A complete treatment of this topic is beyond the scope of this thesis. The most agreed-upon interpretation of the OMG's 4-layered hierarchy is summarised as follows (see Fig. 1.6 and 1.7):

TERMINOLOGY

- Level M_0 contains *originals*, e.g. the person "Adrian Rutle" in the real world
- Level M_1 contains *models*, e.g. a UML class diagram and a snapshot (illustration) of its instances
- Level M_2 contains *metamodels*, e.g. the UML metamodel
- Level M_3 contains the *meta-metamodel* MOF

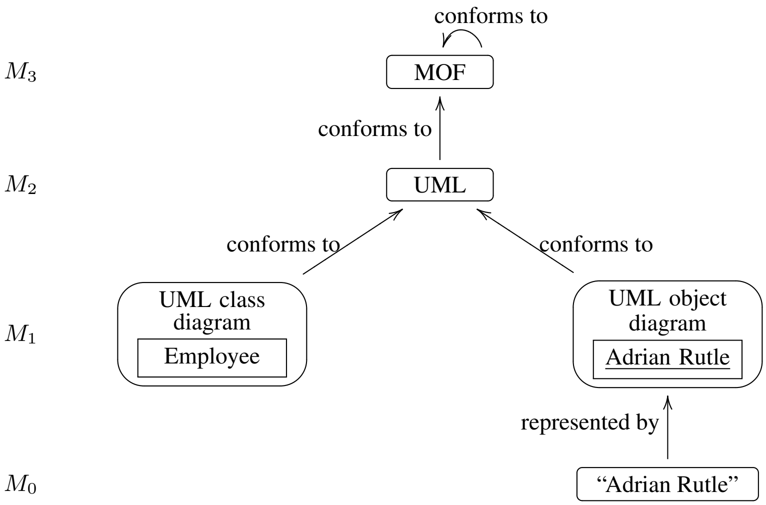


Figure 1.7: OMG's 4-layered hierarchy by examples

- An original at level M_0 is *represented by* a model at level M_1
- A model at level M_1 *conforms to* a metamodel at level M_2
- A metamodel at level M_2 *conforms to* the meta-metamodel MOF at level M_3
- MOF *conforms to* itself

In OMG's 4-layered hierarchy, UML object diagrams and UML class diagrams are located at the same level M_1 although UML object diagrams can be regarded as models which conform to UML class diagrams. At the same time, since UML object diagrams are at level M_1, they are regarded as models which conform to the UML metamodel (see Fig. 1.7). These two flavours of conformance, i.e. between adjacent levels and within one level, are referred to as *linguistic* and *ontological* conformance, respectively [10; 71; 72] (see Fig. 1.8). In this thesis, we follow the same levelling pattern as in Fig. 1.5; i.e. models at each level conform to a model at the level above. Hence we will use two different levels for a model and its instances [71; 86].

LINGUISTIC AND ONTOLOGICAL

One of the main challenges related to the OMG's metamodelling hierarchy is that a formalisation of the relation between modelling languages and metamodels, as well as a formalisation of the conformance between models and metamodels, is not included in the OMG standards. This is despite the fact that many researchers in the field claim that unless a complete formalisation of these relations is given, the potentials of MDE may not be fully unfolded (see [14; 17; 21; 34; 96; 106] for further references). In this thesis, we formally define and distinguish between two levels of conformance relations: *typed by* and *conforms to*; where "conforms

TYPED BY VS CONFORMS TO

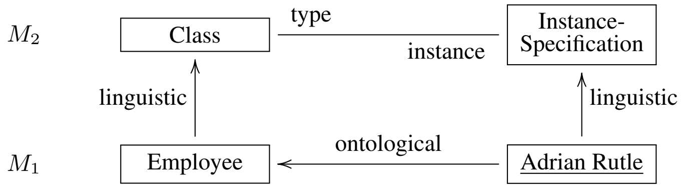


Figure 1.8: Linguistic and ontological conformance; adopted from [71]

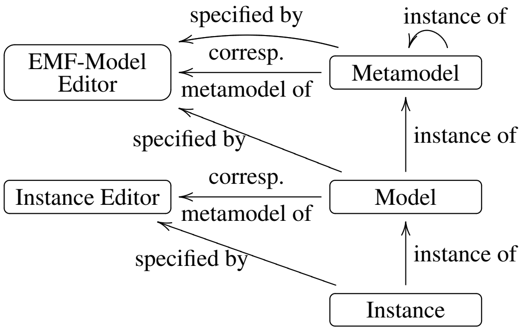


Figure 1.9: EMF's hierarchy

to" is a stronger relation and includes "typed by". These relations will be used between models at any two adjacent levels. Details of these relations are given in Section 4.2.

Being an industrial standard, MOF is used as a basis for several other technologies and frameworks, such as the Eclipse Modeling Framework (EMF) [39], initially developed by IBM and currently maintained as part of the Eclipse project. EMF models are used to specify structural data models and by default the framework may be used to generate Java code for the system which the model specifies. There is a one-to-one correspondence between the metamodel of EMF, called Ecore, and a subset of MOF, called Essential MOF (EMOF). Ecore is basically the same as EMOF. Henceforth the term *MOF-based modelling languages* will be used to denote UML, EMF and other languages and frameworks which are based on MOF.

EMF

MOF-BASED MODELLING LANGUAGES

According to EMF, the metamodelling hierarchy is organised in a 3-layered hierarchy (see Fig. 1.9). From top to bottom, these layers are called metamodel, model and instance. In contrast to OMG's hierarchy, the instance level in EMF is clearly distinguished from the model level. Being inspired by object-oriented modelling, especially Java, the relations between models in the hierarchy are called *instance of*.

EMF'S METAMODELLING HIERARCHY

1.5 Constraints

MOF-based modelling languages allow the specification of simple constraints such as multiplicity and uniqueness constraints, hereafter called *structural constraints*. These constraints are usually specified by properties of classes in the corresponding metamodel of the modelling language. For instance, the requirement "An employee must work for at least one department" in the UML model from Example 1 was forced by a multiplicity constraint which uses the properties lower and upper of the class Property of the UML metamodel (see Fig. 1.10a). Instances of the UML model should satisfy this multiplicity constraint. However, these structural constraints may not be sufficient to specify complex system requirements. Hence, textual constraint languages such as the Object Constraint Language (OCL) [88] are usually used to define complex constraints, hereafter called *attached constraints*. Example 4 illustrates this.

Example 4 (Revisiting the UML Class Diagram) Building on Example 1, we refine the requirements to illustrate the usage of constraint languages to express constraints which are not expressible by UML itself. In Fig. 1.2, we showed a UML class diagram of an information system for the management of employees and departments. Now in addition to the requirements in Example 1, we also require the following:

3. A project may involve zero or many employees.
4. A project must be controlled by at least one department.
5. An employee involved in a project must work in the controlling department.

The requirements 3 and 4 are specified by means of UML syntax itself. However, forcing the fifth requirement can only be achieved by using an attached OCL constraint (see Fig. 1.10b), for example:

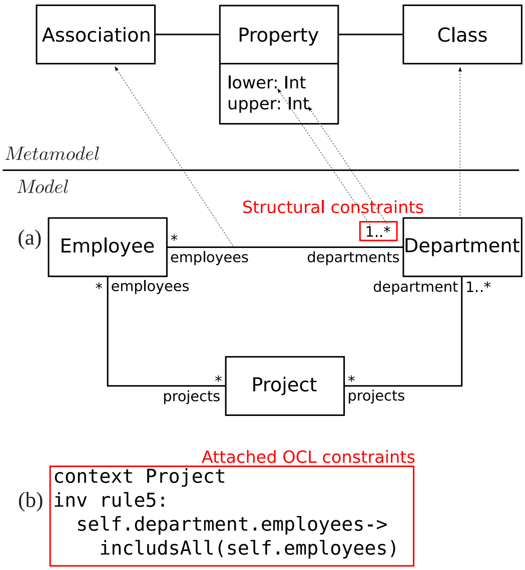
```
context Project
  inv rule5: self.department.employees->
    includesAll(self.employees)
```

◇

Remark 1 *A completely diagrammatic representation of the model in Fig. 1.2 is described in the Sections 2.2 and 3.3.*

CONSTRAINT CLASSIFICATION

We may classify constraints in MOF-based modelling languages based on two factors: their origin, i.e. where they come from; and their effect, i.e. what they constrain. Considering the origin, constraints may come from the modelling language itself, i.e. structural constraints; or from additional constraint languages, e.g. attached OCL constraints. Structural constraints include also typing constraints defined by the metamodel of the modelling language. These are constraints restricting which *types* of elements the models can contain and how these elements can be related to each other, e.g. according to the simplified metamodel of UML



```
context Project
inv rule5:
  self.department.employees->
    includsAll(self.employees)
```

Figure 1.10: Constraints in MOF-based modelling languages: (a) structural constraints in UML (b) attached OCL constraint

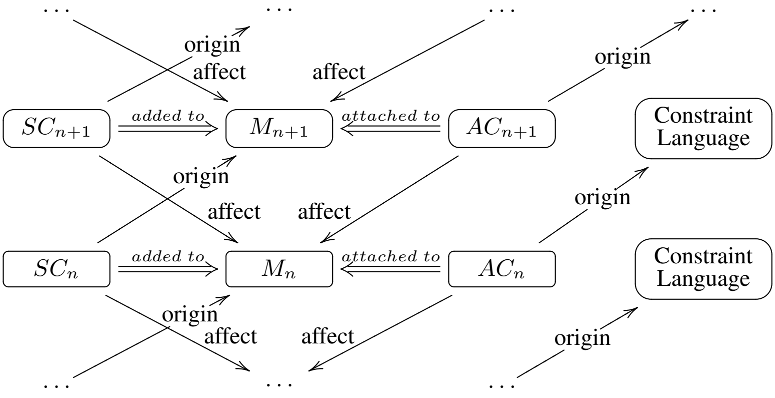


Figure 1.11: Constraints in metamodelling

presented in Fig. 1.10a, we may have *classes*, *associations* and *properties* in a UML class diagram. Considering the effect, we have constraints which should be satisfied by models defined by the modelling language; and constraints which should be satisfied by instances of these models. Thus, for a modelling language with its metamodel at level M_{n+1} we may identify these three kinds of constraints (see Fig. 1.11).

- SC_n: Structural constraints which are added to models at level M_n. The origin of these constraints is the modelling language which has its corresponding metamodel at level M_{n+1}. The effect of these constraints is that they should be satisfied by models at level M_{n-1}.
- AC_n: Attached constraints which are added to models at level M_n. The origins of these constraints are external languages such as OCL. The effect of these constraints is that they should be satisfied by models at level M_{n-1}.
- SC_{n+1}, AC_{n+1}: Structural and attached constraints which are added to models at level M_{n+1}. The origin of these constraints is either the modelling language which has its corresponding metamodel at level M_{n+2}, or an external language such as OCL. The effect of these constraints is that they should be satisfied by models at level M_n.

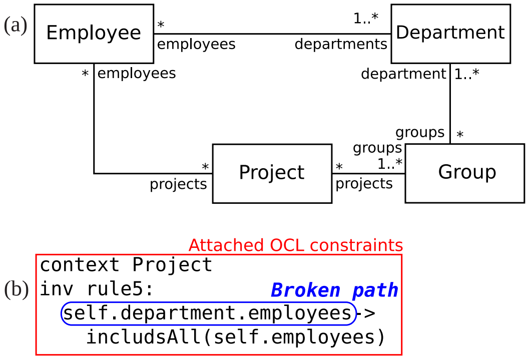
Mixing MOF-based modelling languages with OCL is just a special case of a general pattern where diagrammatic modelling languages use textual languages to define constraints that are difficult to express by their own syntax and semantics. While this solution is to some extent accepted among software developers, we propose in this thesis a completely diagrammatic approach for specifying and reasoning about structural models for the following reasons:

Firstly, the fact that OCL constraints are term-based expressions while models specified by means of MOF-based modelling languages are graph-based structures makes automatic reasoning about these models challenging. As an example, consider the UML class diagram in Fig 4: checking the state of the system against the model will involve two steps: firstly, checking the structure and some of the constraints in UML; secondly, checking the rest of the constraints by an OCL engine. Moreover, any modification in the structure of the UML class diagram must be reflected in the OCL constraints, which are related to the modified structure. This requires the definition of automatic synchronisation of OCL constraints for arbitrary model modifications. However, the identification of classes of modifications, for which an automatic synchronisation of OCL constraints is possible, requires complex machinery to be implemented by tool vendors and may not be possible at all [79].

Example 5 (Constraint Synchronisation Challenge) Building on Example 4, we refine the requirements to illustrate how a modification of the UML class diagram will affect the attached OCL constraint. We modify the fourth requirement from Example 4 to the following:

4a. A group must belong to exactly one department.

4b. A project must be assigned to at least one group.



```
context Project
inv rule5:
  self.department.employees->
    includsAll(self.employees)
```

Figure 1.12: Adding the class Group to the UML class diagram will lead to a broken path in the attached OCL constraint

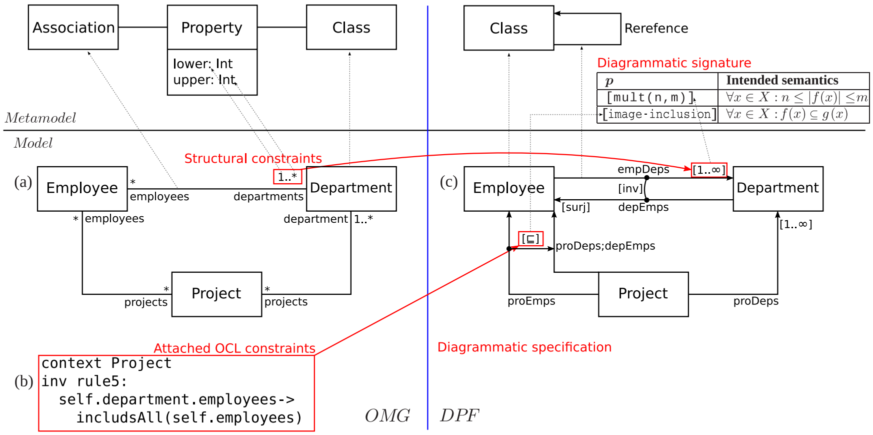
Modifying the UML class diagram by adding the class Group will lead to a broken path in the attached OCL constraint. Fig. 1.12 shows the updated version of the UML class diagram from Fig. 1.10. ◇

Secondly, in order to obey the "everything is a model" vision of MDE [17], it is desirable to have both structure and constraints in the same diagrammatic, model-centric format. This enables models to serve their purpose "to tackle the complexity of software by enabling developers to reason about and deal with a real system at a higher level of abstraction" [102]. Recall again the model in Example 4. Since some of the semantics of the model is hidden in the OCL code, the model development process may become complex and error-prone in the long run. In particular, domain experts may have difficulties in understanding the OCL code – something which may force the developers to use the list of requirements in a natural language instead of the OCL rules and in turn may lead to misunderstandings [22].

One of the main goals of this thesis is to propose a fully diagrammatic formalisation of OMG's metamodelling hierarchy. This formalisation gave rise to the development of the Diagram Predicate Framework (DPF). Among the main features of DPF is the integration of structural and attached constraints in diagrammatic specifications. These constraints are defined by means of predicates which belong to a predefined signature. In this way, constraints are defined diagrammatically, and treated uniformly. While the framework is explained in detail in the remainder of this thesis, Fig. 1.13 gives an informal overview on how specifications and signatures are related to the OMG-world.

1.6 Model Transformation

Model transformation is one of the key techniques in MDE which is used to automate several model-related activities such as code generation, refactoring, optimi-



```
context Project
inv rule5:
  self.department.employees->
    includsAll(self.employees)
```

p	**Intended semantics**
`[mult(n,m)]`	$\forall x \in X : n \leq \lvert f(x) \rvert \leq m$
`[image-inclusion]`	$\forall x \in X : f(x) \subseteq g(x)$

Figure 1.13: Constraints in MOF-based modelling languages and DPF: (a) structural constraints in UML (b) attached OCL constraint (c) integration of constraints in diagrammatic specifications

sation, language translation etc. [112]. Model transformations have many applications in MDE. Some of these applications are listed here:

APPLICATIONS

- Implementation: generation of code from models.
- Refinement: enrichment of software models with details.
- Refactoring: changing software's structure without changing behaviour.
- Translation: translation of software from one language to another.
- Adaptation: changing software to conform to new specifications.
- Evolution: describing relations between different versions of the same model.
- Migration: changing software from one programming language or one framework to another.
- Integration: integration of two or more software models into one.

CONCEPTS

A general definition of model transformation is given in [68] and further generalised in [81] as follows: A *transformation* is the automatic generation of target models from source models, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A *transformation rule* is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language.

Model transformations are carried out automatically by tools in *transformation processes*. Each transformation process is described by a transformation definition, which in turn is written in a *transformation definition language*; hereafter referred

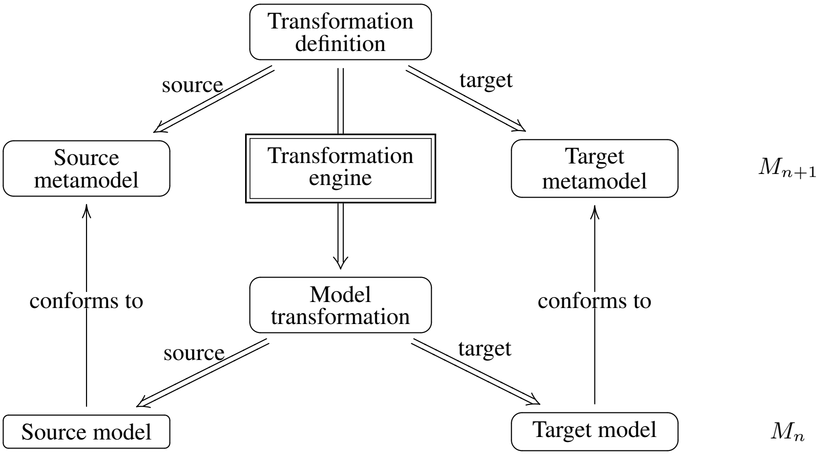


Figure 1.14: Model transformation overview

to only as transformation languages. The tool which is used for the execution of model transformations is called a *transformation engine*.

Given a metamodelling hierarchy, model transformations are defined at a certain level and executed at the level below. Thus for transforming models at level M_n, the transformation rules are defined at level M_{n+1} (see Fig. 1.14). Hence, while writing a transformation definition, the transformation definition language needs to *know about* the types of the model elements which are to be transformed. The next example illustrates this.

Example 6 (Sample Transformation of UML Class Diagram to EMF) Fig. 1.15 shows a simplified transformation definition which describes the transformation of UML class diagrams to EMF models. The transformation definition contains a rule which transforms each `Class` to an `EClass` and a rule which transforms each (binary) `Association` to a pair of `EReference`s which are *opposite* of each other (expressed by the `eOpposite` relation between `EReference`s). The model elements mentioned in the rules, such as `Class` and `EClass`, exist in the UML and Ecore metamodels, respectively. Given a source UML class diagram (bottom left of Fig. 1.15), the transformation engine will search for all model elements of type `Class` and create a corresponding model element of type `EClass` in the target model. More precisely, the model element `Department` which is typed by `Class` in the source model is transformed to a model element `Department` which is typed by `EClass` in the target model. The same will be done for the second rule. ♢

Several classifications of model transformations are given in [31; 81]. A first classification is based on whether the transformation is used to transform between CLASSIFICATION

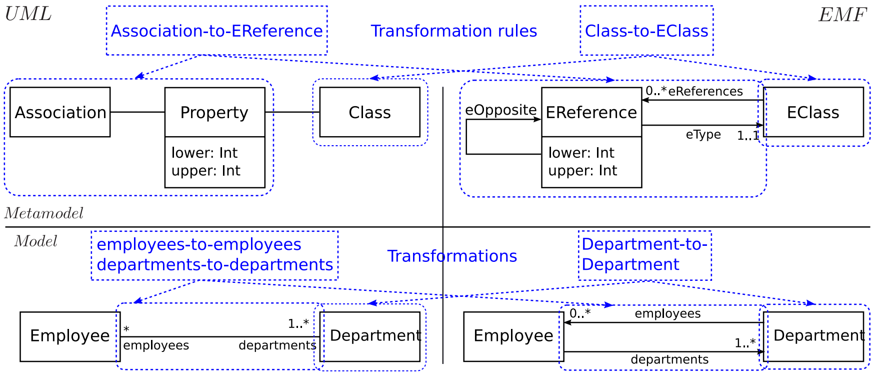


Figure 1.15: Transformation of a UML class diagram to an EMF model

models specified by one modelling language, called *homogeneous* transformation, or between models specified by different modelling languages, called *heterogeneous* transformation. A second classification is based on whether the target model is created from scratch, called *out-place*, or the source model is modified in order to obtain the target model, called *in-place*. The former class of transformations is suitable for model refactoring and optimisation [19; 78], while the latter is suitable for model translation and migration. A third classification is based on the underlying technique which is used to carry out the transformations, e.g. logic programming, functional programming, graph transformation, etc. A fourth classification is based on which properties of the models are preserved by the model transformation; e.g. the structure, behaviour or semantics of the model.

These classifications are orthogonal to each other, e.g. both homogeneous and heterogeneous model transformations may be carried out in-place or out-place, by using logic programming or graph transformation, etc.

DECLARATIVE VS IMPERATIVE APPROACHES

Mechanisms used for model transformations may be either declarative such as functional programming, logic programming and graph transformation; or imperative/operational such as QVT Operational Mappings. Among the features of declarative approaches one can mention the following. Firstly, they are theoretically good and well-founded. Secondly, they support bidirectionality. Thirdly, they are compact and maintainable since they hide procedural information. Finally, they enjoy a simpler semantic model since order of execution, traversal of source models, as well as generation of target models are implicit. However, operational approaches may increase efficiency through incrementally updating models. Moreover, they facilitate control over the order of execution by providing sequences, selections and iterations.

As examples of model transformation approaches we may mention relational, graph-transformation-based and hybrid approaches [31]. The relational approach is a declarative approach in which the main concepts are mathematical relations and mapping rules based on set-theory. Relations between element types from the source and target models are stated in mathematical relations, which are specified by constraints. This approach has the advantage of a good balance between declarative expressiveness, flexibility (rule scheduling) and simplicity [31]. Among proposals which follow this approach are [2; 26; 90; 120].

RELATIONAL APPROACH

GRAPH TRANSFORMATION APPROACH

The graph-transformation-based approach is also a declarative approach inspired by the theoretical work on graph transformations between typed, directed graphs [41]. In this approach, the models to be transformed are graphs. Graph transformation rules define patterns in the source graph that will be transformed to patterns in the target graph. This approach is powerful, declarative, visual, formal and allow for composition; though it has some problems with scalability, tool support and incompatibility of various approaches [31; 81]. Among tools which adopt this approach are AGG, AToM^3, VIATRA2, GReAT [13; 33; 124].

HYBRID APPROACH

Hybrid approaches, where different concepts and paradigms are applied depending on the application domain, seem to be more useful. In the hybrid approach, users can combine the expressive power of graph-based transformations with the flexibility of the relational approach to design their transformation definitions. Among tools and proposals which follow this approach are [4; 11; 90].

FEATURES OF TRANSFORMATION LANGUAGES

In [81], the authors identify a set of *quality requirements* which any model transformation tool or language must meet, such as usability, usefulness, scalability etc. Moreover, they identify some *success criteria* which should be used to evaluate these languages and tools. Among these criteria are reuse and customisation of transformation definitions, guarantee of syntactic and semantic correctness of transformation definitions, etc. Furthermore, in [68; 81; 93] some of the *features* which transformation languages should provide are summarised as follows.

- CRUD: Create, Read, Update and Delete transformation definitions.
- Traceability: it should be possible to trace target model constructs back to their counterpart construct(s) in the source model.
- Incremental consistency: changes in the target model, for example handwritten code, must persist in spite of re-transformation.
- Bidirectionality: the source model can be generated from the target model by application of the inverse of the transformation.
- Rule scheduling: transformation rules can be applied in a user-defined sequence.
- Guarantee of syntactic and semantic correctness: enabling users to answer whether given a well-formed input model the output model is also well-formed, whether the output have an expected property, etc.
- Compose and decompose transformations.
- Test, validate and verify transformations.

Design and specification of transformation definition languages is a relatively new field in software engineering. OMG's initial request for proposals in 2002

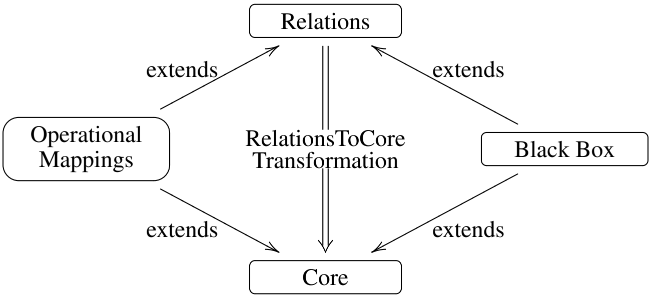


Figure 1.16: QVT Overview; adopted from [90]

on Query/View/Transformation (QVT) was the first call for a standardisation of transformation definition languages. A large number of tools (for example; AGG, GReAT, AToM^3, VIATRA, ATL, QVTP etc, see [31; 74; 81] for a comprehensive list of tools) have been proposed in reply to the OMG's request. However, many of those proposals were already forced by practical needs independent of the OMG's request.

QVT

QVT is an OMG standard proposed for describing transformation definitions [90]. MOF 2.0 is used to define the abstract syntax of QVT, and OCL is used for querying the models and implementing the transformations. QVT is composed of three languages: *Relations*, *Core* and *Operational Mappings* (see Fig. 1.16). The first two are declarative languages and the third is imperative. Relations language is at a higher level of abstraction than the Core language. The semantics of Relations language is described as a transformation into the Core language, a transformation that may be defined in the Relations language itself [66]. The semantics of the Core language is given in a semi-formal set-theoretical notation [90]. The Relations language defines transformations as a set of relations (each containing a set of patterns) among models.

The Operational Mappings language and the Black Box implementation extend the Relations and Core languages and are mechanisms intended to define transformations that are difficult to express in the Relations language. Traceability links are handled automatically by the Relations and Operational Mappings languages, while these links must be handled manually in the Core language [90]. Rules in the Relations and Core languages are multidirectional, while they are unidirectional in the Operational Mappings. The Black Box implementation provides a mechanism for the execution of algorithms and for the reuse of code and libraries written in arbitrary languages.

Recall the discussion about constraints in metamodelling in Section 1.5. When it comes to model transformations, the way constraints are specified in MOF-based

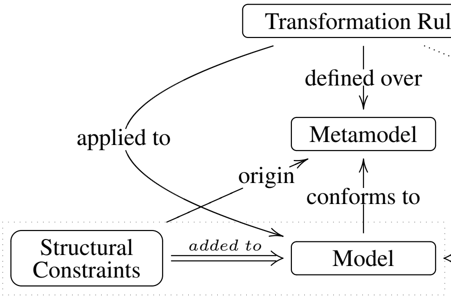


Figure 1.17: Transformation rules are defined over the metamodel of the language; these rules are able to describe transformation of structural constraints but they are unable to describe the transformation of attached constraints (see Fig. 1.11)

CONSTRAINTS IN MODEL TRANSFORMATION

modelling languages may introduce a challenge related to their transformation. While existing model transformation techniques take into account structural constraints, they often ignore the attached constraints [79; 95; 107]. This is because model transformation rules are defined over elements of the metamodels corresponding to the modelling languages, while attached constraints are specified by a different language (see Fig. 1.17). The following example illustrates this.

Example 7 (Constraint Transformation Challenge) Building upon Examples 4 and 6. Suppose that we want to apply the transformation rules to the class Project and the associations connected to it. Since the transformation rules are defined over the metamodel elements, and since the attached OCL constraint is not provided by these metamodels, transforming this constraint automatically will not be possible by these rules. More precisely, the attached OCL constraint in Fig. 1.10b cannot be transformed automatically by the transformation rules in Fig. 1.15. Moreover, it is not possible to define a rule which transforms arbitrary OCL constraints. ◇

This challenge is closely related to the fact that the conformance relation between models and metamodels is not formally defined for MOF-based modelling languages [34; 96], especially when OCL constraints are involved [21]. As mentioned, the DPF based approach to metamodelling addresses this issue by integrating structural and attached constraints in diagrammatic specifications. In order to handle these constraints, this metamodelling approach is supplemented by a formal approach to constraint-aware model transformation detailed in Chapter 5. This approach can be regarded as a further development or extension of graph transformation systems in the sense that it can be used to transform the structure of models as well as attached constraints. Thus it offers more sophisticated means to describe, control and execute model transformations.

1.7 Model Management

As mentioned, models are the first-class artefacts of the software development process in MDE. These models are typically developed by distributed environments consisting of teams at different organisations and locations. These teams usually build multiple overlapping models which represent different aspects of the same systems. In addition, these teams may work on different versions of the same models. Model management is concerned with describing the relations between these models and providing systematic techniques to manipulate these relations as well as the models themselves [23].

Various aspects of model management include how to identify and how to refine the relationships between independently-developed models, how to combine models with respect to relations between them, how to ensure consistency between models originating from different sources and how to propagate changes made on one model to other models related to it [23]. One of the techniques used to support model management activities is version control of models. Version control is used during software evolution to keep track of different versions of software artefacts produced over time.

VERSION CONTROL

In general, there are two main approaches on which major version control technologies are based: *lock-modify-unlock* and *copy-modify-merge* [30]. In the lock-modify-unlock approach, the software artefacts are stored in a repository which allows only one developer to work on a particular artefact at a time. This approach is workable if the developers know who is planning to do what at any given time and can communicate with each other quickly. However, if the development group is too large or distributed, dealing with locking issues may become problematic.

LOCK-MODIFY-UNLOCK

In the copy-modify-merge approach, each developer accesses a repository and creates a personal *working copy* – a snapshot of the repository's files and directories. Then, the developers modify their working copies simultaneously and independently. Finally, the local modifications are merged together into the repository. The version control system (VCS) assists in the merging by detecting conflicting modifications. When a conflict is detected, the system requires manual intervention by the developers.

COPY-MODIFY-MERGE

Traditional VCSs such as Subversion facilitate efficient concurrent development of source code by using the copy-modify-merge approach. Unfortunately, these VCSs focus on the management of text-based files, such as source code. Thus, the difference calculation, merging of modifications and conflict detection are based on a per-line textual comparison. Since the structure of models is graph-based rather than text- or tree-based, the existing techniques are not suitable for MDE.

Recent research has led to a number of findings in model evolution. The interested reader may consult [76] for difference calculation, [28] for difference representation, [82] for conflict detection and [80] for a survey on software merging, to cite a few. However, some of the proposed solutions are not formalised enough to enable automatic reasoning about model evolution. For example, operations such as *change* or *update* are given different and ambiguous semantics in different works. Moreover, the terminology used in these solutions is not precise, e.g. terms *add*, *create* and *insert* are often used to refer to the same operations. Furthermore, the approach to version control (e.g. copy-modify-merge) is not formalised explicitly and concepts such as *synchronisation* and *commit* are only defined semi-formally.

To follow the success of code-based version control, the copy-modify-merge approach should be adopted in MDE also. This would require formal techniques which target graph-based structures. As a first step, we introduce in this thesis a formalisation of the copy-modify-merge approach for models. This formalisation supplements the metamodelling and model transformation approaches of DPF [101; 105].

CHAPTER 2

Introduction to Diagrammatic Modelling

As an introduction to diagrammatic modelling, we start this part by introducing graphs and related concepts. We shortly review how directed multi-graphs may be used as a first approximation to represent (meta)models in OMG's metamodelling hierarchy. The chapter ends by a gentle introduction to the DPF based approach to diagrammatic modelling.

2.1 Introduction

In software engineering, diagrammatic models are graph-based structures where different kinds of graphs, e.g. simple graphs, directed graphs, directed multi-graphs, attributed graphs, hypergraphs, bipartite graphs, etc., may be used as a basis for these models. A graph is a mathematical structure consisting of a collection of nodes and a collection of edges between these nodes. Graphs and graph homomorphisms are used to represent many concepts in the thesis. For this reason, we start this chapter by defining some graph related concepts and the category **Graph**.

Definition 1 (Graph) *A graph $G = (G_0, G_1, src^G, trg^G)$ is given by a collection G_0 of nodes, a collection G_1 of arrows and two maps $src^G, trg^G : G_1 \to G_0$ assigning the source and target to each arrow, respectively. We write $f : X \to Y$ to indicate that $src(f) = X$ and $trg(f) = Y$.*

Definition 2 (Subgraph) *A graph $G = (G_0, G_1, src^G, trg^G)$ is subgraph of a graph $H = (H_0, H_1, src^H, trg^H)$, written $G \sqsubseteq H$, iff $G_0 \subseteq H_0$, $G_1 \subseteq H_1$ and $src^G(f) = src^H(f), trg^G(f) = trg^H(f)$, for all $f \in G_1$.*

Definition 3 (Graph Homomorphism) *A graph homomorphism $\varphi : G \to H$ is a pair of maps $\varphi_0 : G_0 \to H_0$, $\varphi_1 : G_1 \to H_1$ which preserve the sources and* *targets; i.e. for each arrow $f : X \to Y$ in G we have $\varphi_1(f) : \varphi_0(X) \to \varphi_0(Y)$ in H.*

Remark 2 (Inclusion Graph Homomorphism) *$G \sqsubseteq H$ iff the inclusion maps $inc_0 : G_0 \hookrightarrow H_0$, $inc_1 : G_1 \hookrightarrow H_1$ define a graph homomorphism $inc : G \hookrightarrow H$.*

After defining graphs and graph homomorphisms, it is natural to consider all graphs and graph homomorphisms as a whole. The most convenient concept for this purpose is the concept of a category [15; 45].

Definition 4 (Category of Graphs) *The category* **Graph** *has graphs as objects and its morphisms are graph homomorphisms.*

The composition $\varphi;\psi : G \to K$ of two graph homomorphisms $\varphi : G \to H$ and $\psi : H \to K$ is defined component-wise $\varphi;\psi = (\varphi_0, \varphi_1);(\psi_0, \psi_1) := (\varphi_0;\psi_0, \varphi_1;\psi_1)$. The identity graph homomorphisms $id^G : G \to G$ are also defined component-wise $id^G = (id^{G_0}, id^{G_1})$. This ensures that the composition of graph homomorphisms is associative and that identity graph homomorphisms are identities with respect to composition. By $\mathbf{Graph}_0$ *we denote the collection of all objects in this category; i.e. the collection of all graphs.*

2.2 Diagrammatic Modelling by Graphs

Typing

In our graph-based formalisation of metamodelling, models are represented by directed multi-graphs. Moreover, the conformance relation between models and metamodels is represented by a *typing morphism*; that is, a graph homomorphism which assigns a type, i.e. an element of the metamodel, to each element of the model. A model is said to be *typed by* a metamodel if there is a typing morphism from the model to the metamodel.

Definition 5 (Type Graph and Typing Morphism) *A type graph is a distinguished graph $TG = (TG_0, TG_1, src^{TG}, trg^{TG})$. A typed graph (G, ι) which is typed by TG is a graph G together with a graph homomorphism $\iota : G \to TG$. The homomorphism ι is called a typing morphism.*

Definition 6 (Typed Graph Homomorphism) *Given a type graph TG, a typed graph homomorphism $\phi : (G, \iota^G) \to (H, \iota^H)$ is a graph homomorphism $\iota : G \to H$ such that $\iota^G = \phi;\iota^H$.*

$$\begin{tikzcd}
 & TG & \\
 & = & \\
G \arrow[uur, "\iota^G"] \arrow[rr, "\phi"] & & H \arrow[uul, "\iota^H"']
\end{tikzcd}$$

Semantics

When using graphs to represent models, nodes and arrows of the graphs have to be interpreted in a way which is appropriate for the corresponding modelling environment [106]. For structural object-oriented models, for example, it is appropriate to interpret nodes as sets and arrows $X \xrightarrow{f} Y$ 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. $\wp(Y) = \{K \mid K \subseteq Y\}$. 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 reason for this choice of interpretation is that in object-oriented structural modelling each object may be related to a set of other objects. On the other hand, for relational data models it is appropriate to interpret nodes as sets and arrows as single-valued functions. This is because each data-row in a relational table may be related to exactly one data-row of another (or the same) table.

The semantics of nodes and arrows of a graph can be formally defined in either *indexed* or *fibred* way [38; 128]. 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).

INDEXED SEMANTICS

In the fibred version, the semantics of a graph G is given by the set of its instances (I, ι) where $\iota : I \to G$ is a graph homomorphism. A node X in G is interpreted by ι as the set of $\iota^{-1}(X)$ of nodes in I; and an arrow $X \xrightarrow{f} Y$ in G represents a multi-valued function $\iota^{-1}(f)$ from $\iota^{-1}(X)$ to $\wp(\iota^{-1}(Y))$, where for any nodes x and y in I we have $y \in \iota^{-1}(f)(x)$ if and only if there is an arrow $g : x \to y$ in I with $\iota(g) = f$. f represents a total (and single-valued) function if for each node x in I there is exactly one y and one $g : x \to y$ in I with $\iota(g) = f$.

FIBRED SEMANTICS

Software engineers prefer the fibred semantics because it reflects the conformance relation between models and metamodels as described in the metamodelling hierarchy. In contrast, mathematicians prefer the indexed semantics. Fortunately, the switch between these two semantics is possible where the Grothendieck construction, as described in [128] for graphs, transfers indexed into fibred semantics.

The following example revisits Example 4, and explains how the graph-based formalisation of the metamodelling hierarchy works in practice. This example is kept intentionally simple, retaining only the details which are relevant for our discussion. The requirements are relaxed compared to the original ones in Example 4; we will treat all the requirements in Section 3.3.

Example 8 (Diagrammatic Modelling by Graphs) Let us consider an information system for the management of employees and projects. At any state of the system the following requirements should be satisfied:

1. An employee may work for zero or many departments.
2. A department may have zero or many employees.
3. A project may involve zero or many employees.
4. A project may be controlled by zero or many departments.

Fig. 2.1a shows a graph G_3 representing a generic meta-metamodel. Fig. 2.1b shows a graph G_2 representing a metamodel for the specification of structural object-oriented models. Fig. 2.1c shows a graph G_1 representing a structural object-oriented model specifying the requirements above. In these graphs, nodes and arrows are interpreted as sets and multi-valued functions. Fig. 2.1d shows a graph G_0 representing an instance of the model G_1.

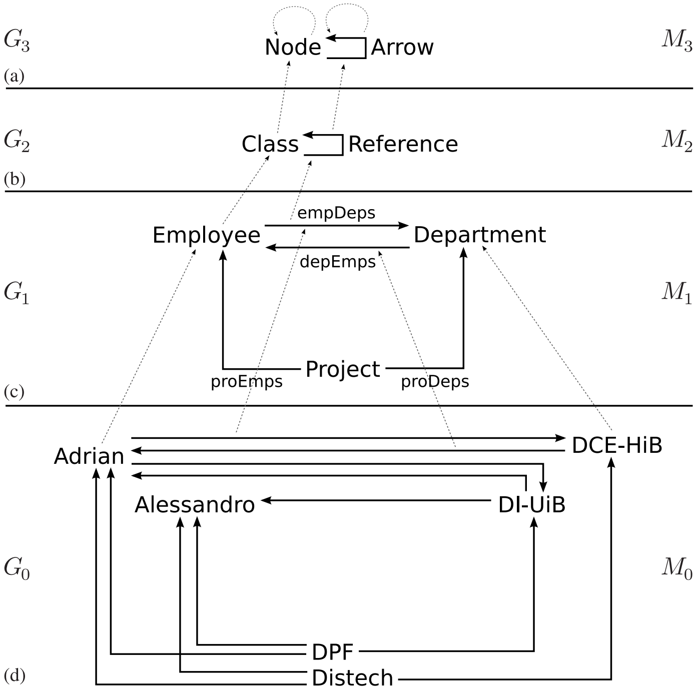


Figure 2.1: A sample metamodelling hierarchy using the graph-based formalisation

The graph G_3 is typed by itself (reflexive). That is, it satisfies the restriction that nodes and arrows may only be of type Node and Arrow, respectively. Moreover, the graph G_2 is typed by G_3. That is, it satisfies the restriction that nodes and arrows may only be of type Node and Arrow, respectively, e.g. the node Class is of type Node and the arrow Reference is of type Arrow. Furthermore, the graph G_1 is typed by G_2. That is, it satisfies the restriction that nodes and arrows may only be of type Class and Reference, respectively, e.g. the nodes Employee and Department are of type Class and the arrows connecting them are of type Reference. Similarly, the instance G_0 is typed by G_1. In Fig. 2.1, some of the typing morphisms are denoted by gray, dashed arrows between the levels in the hierarchy. ♢

2.3 From Graphs to Diagrammatic Specifications

Although the usage of graphs for the representation of model structures is a success story, an enhancement of the formal basis is needed to:

- Express well-formed diagrammatic constraints which go beyond the structural constraints that are formalised as graph constraints [41; 118].

- Formalise the relations between models at adjacent levels of the metamodelling hierarchy. This relationship is expressed by the concepts of *type-* and *typed graphs* in graph theory [41], which does not capture the type of constraints mentioned above.

A natural choice for an enhancement of graph theory is category theory, and in particular the sketch formalism, which can be used to define semantics of diagrams, thus of diagrammatic models. In the categorical sketch formalism, models are represented as graphs, and model properties are expressed by universal properties such as; limit, colimit, and commutativity constraints [15; 45]. 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 [38].

CATEGORICAL SKETCHES

The proposed formalisation approach in this thesis is the Diagram Predicate Framework (DPF) [101; 104–107], which is a generalisation and adaptation of the categorical sketch formalism, where user-defined diagrammatic predicate signatures represent the constructs of modelling languages in a more direct and adequate way. In particular, DPF is an extension of the Generalised Sketches [77] formalism originally developed by Diskin et al. in [34; 35; 37]. DPF aims to combine mathematical rigour – which is necessary to enable automatic reasoning – with diagrammatic modelling.

GENERALISED SKETCHES

DPF is a generic graph-based specification framework that tends to adapt first-order logic and categorical logic to software engineering needs. DPF is generic in the sense that it supports any of kind of graph structures (see [38] for the general case). However, the variant of DPF which we employ in this thesis is based on directed multi-graphs.

DPF

DPF provides a formal, diagrammatic approach to metamodelling, in which models at any level are formalised as diagrammatic specifications. Each *diagrammatic specification* consists of an underlying graph together with a set of diagrammatic constraints. Moreover, modelling languages are formalised as modelling formalisms. Each *modelling formalism* consists of a corresponding meta-specification, which specifies the types allowed by the language, and a *diagrammatic predicate signature*, which collects the set of predicates used to add constraints to specifications specified by the modelling formalism. Furthermore, the *conformance relation* between a specification at any level and a specification at the level directly above it is formalised as a graph homomorphism between the underlying graphs of the specifications which satisfies the constraints that are added to the upper level specification. In addition, the conformance relation is strengthened by the concept of *universal constraints*. These constraints are connected to a certain modelling formalism and are used to express overall requirements that each specification specified by the modelling formalism should satisfy.

DPF CONCEPTS

CHAPTER 3

Diagrammatic Modelling in DPF

This chapter explains the formal foundation of DPF and its usage in the formalisation of diagrammatic modelling.

3.1 Introduction

In Example 8 of Section 2.2, the structural object-oriented model is compliant with the initial requirements. However, in most practical cases these requirements might not be sufficient. For instance, we may add the requirement "an employee involved in a project must work in the controlling department", as done in Example 4. This additional requirement can not be forced easily by means of graphs and typing morphisms alone. To cope with this, DPF provides a more sophisticated formalisation of diagrammatic modelling.

In DPF, independent from their position in the metamodelling hierarchy, models are represented by *diagrammatic specifications*. A diagrammatic specification $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$ consists of an underlying graph S together with a set of *atomic constraints* $C^{\mathfrak{S}}$. The graph represents the structure of the model, and predicates from a predefined *diagrammatic predicate signature* Σ are used to add constraints to this structure. With regard to the classification of constraints in Section 1.5, the atomic constraints added to a specification at level n of a metamodelling hierarchy may represent both structural constraints, SC_n, and additional constraints, AC_n. In the sequel, the concepts of signatures, constraints and specifications are explained.

3.2 Signatures

A signature in DPF consists of a collection of diagrammatic predicates. During modelling in DPF, these predicates are used to add constraints on the underlying structure of specifications. Each predicate has a name, a shape, a visualisation (if possible) and a semantic interpretation. Which kind of predicates are included in a signature, and which semantics these predicates have, are dependent on the modelling environment in which the signature is used. For example, a signature used for constraining relational database models will contain primary key and foreign key predicates.

Definition 7 (Signature) *A (diagrammatic predicate) signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of predicate symbols P^{Σ} with a map α^{Σ} that assigns a graph to each predicate symbol $p \in P^{\Sigma}$. $\alpha^{\Sigma}(p)$ is called the arity of the predicate symbol p.*

Table 3.1 shows a sample signature $\Sigma_2 = (P^{\Sigma_2}, \alpha^{\Sigma_2})$[1]. The first column of the table shows the names of the predicates. The second and the third columns show the arities of predicates and a possible visualisation of the corresponding constraints, respectively. In the fourth column, the semantic interpretation of each predicate is specified. For readability reasons, the semantic interpretation is presented in a set-theoretical indexed manner where arrows are interpreted as multi-valued functions. The predicates in Table 3.1 are generalisations, or general patterns, for constraints that are used in structural object-oriented modelling; e.g. `[mult,(m,n)]` for multiplicity constraints in UML class diagrams.

Remark 3 (Predicate Names) *Some of the predicate names in Σ_2 in Table 3.1 refer to unique predicates, e.g. `[surjective]`, while some others refer to a family of predicates, e.g. `[mult(n,m)]` and `[jointly-surjective_2]`. In the case of `[mult(n,m)]`, the predicate is parameterised by the integers n and m, which represent the lower and upper bounds, respectively, of the cardinality of the function which is constrained by this predicate. In the case of `[jointly-surjective_2]`, the integer 2 serves as a parameter which defines the number of arrows in the arity of the predicate.*

3.2.1 SEMANTICS OF PREDICATES

In DPF, semantics of predicates is defined in a fibred manner. That is, the semantics of a predicate p is given by the set of its instances $\iota : O \to \alpha(p)$ where each ι is a graph homomorphism into the arity of the predicate.

There are different ways to define semantics of predicates.

- In Table 3.1, we have used the mathematical language of set theory to define the semantic interpretation of predicates.
- In categorical sketches [15], where the signature is restricted to limit, colimit and commutativity predicates, the semantics of these predicates is mathematically “pre-defined” for any category according to the universal nature of these special predicates.

[1] We use the subscript 2 for this signature since later it will be used at level M_2 of the metamodelling hierarchy

Table 3.1: A sample signature Σ_2

p	$\alpha^{\Sigma_2}(p)$	Proposed vis.	Semantic Interpretation
`[mult(n,m)]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[r, "f", "{[n..m]}"'] & \boxed{Y} \end{tikzcd}$$	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[irreflexive]`	$$\begin{tikzcd} 1 \arrow[loop above, "f"] \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[loop above, "f", "{[irr]}"'] \end{tikzcd}$$	$\forall x \in X : x \notin f(x)$
`[injective]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[r, "f", "{[inj]}"'] & \boxed{Y} \end{tikzcd}$$	$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[non-overlapping]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[r, "f", "{[nov]}"'] & \boxed{Y} \end{tikzcd}$$	$\forall x, x' \in X : f(x) \cap f(x') \neq \emptyset$ implies $x = x'$
`[surjective]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[r, "f", "{[surj]}"'] & \boxed{Y} \end{tikzcd}$$	$f(X) = Y$
`[jointly-surjective_2]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \\ & 3 \arrow[u, "g"'] \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[r, "f"] & \boxed{Y} \\ & \boxed{Z} \arrow[u, "g"', "{[js]}"] \end{tikzcd}$$	$f(X) \cup g(Z) = Y$
`[inverse]`	$$\begin{tikzcd} 1 \arrow[r, bend left, "f"] & 2 \arrow[l, bend left, "g"] \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[r, bend left, "f"] \arrow[r, phantom, "{[inv]}"] & \boxed{Y} \arrow[l, bend left, "g"] \end{tikzcd}$$	$\forall x \in X$, $\forall y \in Y : y \in f(x)$ iff $x \in g(y)$
`[composition]`	$$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[dr, "h"'] & 2 \arrow[d, "g"] \\ & 3 \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[r, "f"] \arrow[dr, "h\ {[comp]}"'] & \boxed{Y} \arrow[d, "g"] \\ & \boxed{Z} \end{tikzcd}$$	$\forall x \in X : h(x) = \bigcup \{ g(y) \mid y \in f(x) \}$
`[image-inclusion]`	$$\begin{tikzcd} 1 \arrow[r, bend left, "f"] \arrow[r, bend right, "g"'] & 2 \end{tikzcd}$$	$$\begin{tikzcd} \boxed{X} \arrow[r, bend left, "f"] \arrow[r, bend right, "g"'] \arrow[r, phantom, "{[\sqsubseteq]}"] & \boxed{Y} \end{tikzcd}$$	$\forall x \in X : f(x) \subseteq g(x)$

- In modelling tools, one can rely on a less descriptive but more algorithmic way to define the semantics of predicates, e.g. by implementing a validator for each predicate.

However, in order to analyse and formalise diagrammatic modelling, it is not necessary to decide for one of the above mentioned possibilities; it is sufficient to know that any of these possibilities defines "valid instances of predicates".

Definition 8 (Semantics of Predicates) *A semantic interpretation* $[\![..]\!]^{\Sigma}$ *of a signature* $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$ *is given by a mapping that assigns to each* $p \in P^{\Sigma}$ *a set* $[\![p]\!]^{\Sigma}$ *of graph homomorphisms* $\iota : O \to \alpha^{\Sigma}(p)$ *called valid instances of* p*, where* O *may vary over all graphs. We assume that* $[\![p]\!]^{\Sigma}$ *is closed under isomorphisms.*

For a predicate p, we will write $[\![p]\!]$ instead of $[\![p]\!]^{\Sigma}$ if the signature Σ is obvious from the context.

Example 9 (Semantics of Predicates) We consider the predicates `[surjective]`, `[injective]` and `[nonoverlapping]` all with arity $1 \xrightarrow{f} 2$ from Table 3.1. Let the following be graph homomorphisms:

$$\begin{tikzcd}
 & a_1 \arrow[r] & b_1 & & \\
\iota_1 : & & & \longrightarrow & (1 \xrightarrow{f} 2) \\
 & a_2 \arrow[r] & b_2 & &
\end{tikzcd}$$

$$\begin{tikzcd}
 & a_1 \arrow[r] & b_1 & & \\
\iota_2 : & & & \longrightarrow & (1 \xrightarrow{f} 2) \\
 & a_2 \arrow[r] \arrow[uur] & b_2 & &
\end{tikzcd}$$

The graph homomorphisms ι_1, ι_2 represent multi-valued functions $e_1, e_2 : \{a_1, a_2\} \to \wp(\{b_1, b_2\})$, respectively, where $\wp(\{b_1, b_2\}) = \{\emptyset, \{b_1\}, \{b_2\}, \{b_1, b_2\}\}$, with $e_1(a_1) = e_2(a_1) = \{b_1\}$, $e_1(a_2) = \{b_2\}$ and $e_2(a_2) = \{b_1, b_2\}$. We have $\iota_1, \iota_2 \in [\![surjective]\!]$, $\iota_1, \iota_2 \in [\![injective]\!]$ and $\iota_1 \in [\![non - overlapping]\!]$, but $\iota_2 \notin [\![non - overlapping]\!]$. $\diamondsuit$

Remark 4 (Relations between Predicates) *During the design of a signature, the designer declares a set of predicates and defines arities, visualisations and semantic interpretations for predicates. It may be desired, in addition, to define relations between these predicates. Defining these relations can be seen as defining properties of the semantic interpretation of these predicates. For example in the signature in Table 3.1, an instance of the predicate* `[non-overlapping]` *is also an instance of (or satisfies) the predicate* `[injective]`*. This kind of relation is called predicate dependency in [38; 127]. Considering predicate dependencies will turn a signature* Σ *into a pair* $(P^{\Sigma}, \alpha^{\Sigma})$ *where* P^{Σ} *is a graph (or a category) of predicate symbols, and where* α^{Σ} *is a graph homomorphism (or a functor)*

$\alpha^{\Sigma} : P^{\Sigma} \to \mathsf{Graph}^{op}$. Hence, a predicate dependency has the form $p \vdash^{r} q$ with the arities $\alpha^{\Sigma}(r) : \alpha^{\Sigma}(q) \to \alpha^{\Sigma}(p)$ and for any semantic interpretation $[\![..]\!]^{\Sigma}$ it is required that any valid instance of p can be turned, by a pullback construction, into a valid instance of q. In this thesis, we generalise and formalise this kind of relation as specification entailment in Section 3.4.3.

3.3 Diagrammatic Specifications

As mentioned, in DPF models are represented by (diagrammatic) specifications. In this section we define the syntax and semantics of specifications.

3.3.1 Syntax of Specifications

The syntax of specifications reflects the graph-based nature of models; that is, it consists primarily of a graph. Parts of this graph are marked with atomic constraints[2]. These constraints represent properties which instances of specifications should satisfy.

Definition 9 (Atomic Constraint) *Given a signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$, an atomic constraint (p, δ) added to a graph S is given by a predicate symbol p and a graph homomorphism $\delta : \alpha^{\Sigma}(p) \to S$.*

Definition 10 (Specification) *Given a signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$, a (diagrammatic) specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ is given by a graph S and a set $C^{\mathfrak{S}}$ of atomic constraints (p, δ) on S with $p \in P^{\Sigma}$.*

The following example explains how the DPF based approach to diagrammatic modelling works in practice.

Example 10 (Diagrammatic Modelling in DPF) Building on Examples 4 and 8, let us refine the requirements 1 and 2 of the information system for the management of employees and projects as follows:

1'. An employee must work for at least one department.

2. A department may have zero or many employees.

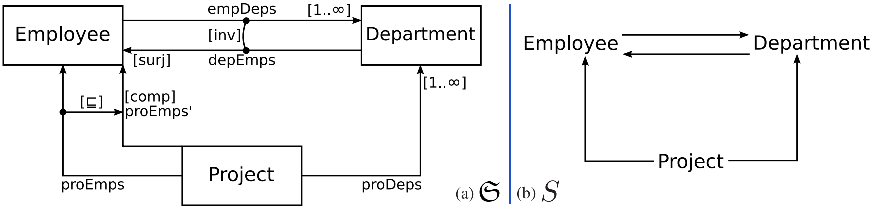
3. A project may involve zero or many employees.

4'. A project must be controlled by at least one department.

5. An employee involved in a project must work in the controlling department.

Fig. 3.1a shows a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma_2)$ representing a structural object-oriented model corresponding to the requirements above. Moreover, the underlying graph S of $\mathfrak{S}$ is shown in Fig. 3.1b, which is exactly the same as the graph G_1 in Fig. 2.1c. The constraints added to the underlying graph show some

[2] The concept of *atomic constraints* corresponds to the concept of *diagrams* in category theory [15].



(a) $\mathfrak{S}$ (b) S

Figure 3.1: A sample specification $\mathfrak{S} = (S, C^{\mathfrak{S}}\!:\!\Sigma_2)$ and its underlying graph S

of the extension which DPF contributes to modelling by plain graphs presented in Section 2.2.

We present how two of the above mentioned requirements are forced in $\mathfrak{S}$ by means of atomic constraints. These atomic constraints are formulated by predicates from the signature Σ_2 in Table 3.1. The requirement "an employee must work for at least one department" is forced in $\mathfrak{S}$ by the predicate `[mult(1,∞)]` on the arrow empDeps; that is by the atomic constraint $($`[mult(1,∞)]`$, \delta_1)$ (see Table 3.2 for the complete list of atomic constraints). Furthermore, the requirement "an employee involved in a project must work in the controlling department" is forced in $\mathfrak{S}$ by the predicates `[composition]` and `[image-inclusion]` on the arrows proEmps' and proEmps, with $proEmps' := proDeps; depEmps$, i.e. the composition of the arrow $proDeps$ with $depEmps$. ♢

INTEGRATION OF CONSTRAINTS

The requirements in Example 10 were specified in a model by mixing UML and OCL syntax as shown in Example 4 of Section 1.3. Both structural constraints and the attached OCL constraint are integrated in the specification shown in Fig 3.1a by using predicates from the signature Σ_2 (see also Fig. 1.13). That is, instead of using OCL or another language to add constraints to the models, in DPF we create or extend a signature with predicates according to the needs.

Remark 5 (Named Boxes) *Note that the rectangular boxes around the model elements in the specification $\mathfrak{S}$ in Fig. 3.1a come from the typing of these elements. This is just a convention adopted from UML and other diagrammatic languages which represent concepts as named rectangular boxes. Typing of model elements will be discussed in Section 4.2; and the notations for typing in Remark 16.*

Remark 6 (OCL Constraints in DPF) *Note that OCL-constraints which can be seen as properties of sets, functions or collections of sets and functions as a whole, can be specified in DPF. In order to cover all OCL-constraints, we need to extend DPF with "diagram operations"; i.e. operations which are used to show derived information [36]. This extension is out of the scope of this thesis.*

Table 3.2: The set $C^{\mathfrak{S}}$ of constraints

(p, δ)	$\alpha^{\Sigma_2}(p)$	$\delta(\alpha^{\Sigma_2}(p))$
`([mult(1,∞)],`δ_1`)`	$1 \xrightarrow{f} 2$	$Employee \xrightarrow{empDeps} Department$
`([surjective],`δ_2`)`	$1 \xrightarrow{f} 2$	$Department \xrightarrow{depEmps} Employee$
`([mult(1,∞)],`δ_3`)`	$1 \xrightarrow{f} 2$	$Project \xrightarrow{proDeps} Department$
`([composition],` δ_4`)`	$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[dr, "h"'] & 2 \arrow[d, "g"] \\ & 3 \end{tikzcd}$	$\begin{tikzcd} Project \arrow[r, "proDeps"] \arrow[dr, "proEmps'"'] & Department \arrow[d, "depEmps"] \\ & Employee \end{tikzcd}$
`([image-inclusion],`δ_5`)`	$\begin{tikzcd} 1 \arrow[r, bend left, "f"] \arrow[r, bend right, "g"'] & 2 \end{tikzcd}$	$\begin{tikzcd} Project \arrow[r, bend left, "proEmps"] \arrow[r, bend right, "proEmps'"'] & Employee \end{tikzcd}$
`([inverse],`δ_6`)`	$\begin{tikzcd} 1 \arrow[r, bend left, "f"] & 2 \arrow[l, bend left, "g"] \end{tikzcd}$	$\begin{tikzcd} Employee \arrow[r, bend left, "empDeps"] & Department \arrow[l, bend left, "depEmps"] \end{tikzcd}$

3.3.2 Semantics of Specifications

In this thesis, we describe semantics of specifications in a fibred manner. That is, 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 $\mathfrak{S}$ is a graph I together with a typing graph homomorphism $\iota : I \to S$ which satisfies the constraints $C^{\mathfrak{S}}$.

To check that a constraint is satisfied in a given instance of $\mathfrak{S}$, it is enough to inspect only the part of $\mathfrak{S}$ which is affected by the constraint. This kind of "restriction to a subpart" is described by the pullback construction [15; 45], a generalisation of the inverse image construction (see Appendix A.1.1).

Definition 11 (Instance of Specification) *Given a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$, an instance (I, ι) of $\mathfrak{S}$ is a graph I together with a graph homomorphism $\iota : I \to S$ such that for each constraint $(p, \delta) \in C^{\mathfrak{S}}$ we have $\iota^* \in [\![p]\!]$, where $\iota^* : O^* \to \alpha^{\Sigma}(p)$ is given by the following pullback diagram*

$$\begin{tikzcd} \alpha^{\Sigma}(p) \arrow[r, "\delta"'] & S \\ O^* \arrow[u, "\iota^*"'] \arrow[r, "\delta^*"'] \arrow[ur, phantom, "PB"] & I \arrow[u, "\iota"'] \end{tikzcd}$$

We use $(I, \iota) \vDash p$ to denote $\iota^* \in [\![p]\!]$ and $(I, \iota) \vDash C^{\mathfrak{S}}$ if $(I, \iota) \vDash p$ for all $(p, \delta) \in C^{\mathfrak{S}}$.

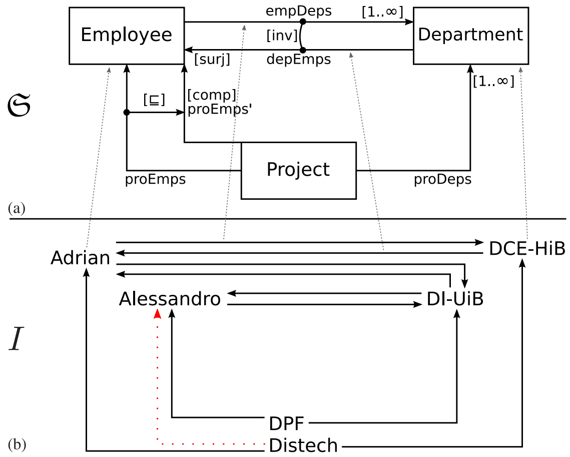


Figure 3.2: A sample specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma_2)$ and a possible instance (I, ι); the dashed, red arrow represents an arrow which would violate the constraint $(\mathtt{[image\text{-}inclusion]}, \delta_5)$

The following example builds on Example 10 and explains the usage of Definition 11 to check whether a given graph is an instance of a specification.

Example 11 (A Sample Specification) Fig. 3.2 shows the specification $\mathfrak{S}$ from Fig. 3.1a in Example 10 together with an instance $\iota : I \to S$ of $\mathfrak{S}$. In this specification, nodes and arrows are interpreted as sets and multi-valued functions. Moreover, some of the typing morphisms from I to $\mathfrak{S}$ are shown as dashed, gray arrows.

In addition to the typing restrictions, the graph I satisfies the set of atomic constraints $C^{\mathfrak{S}}$ (see Table 3.2). To verify that (I, ι) is indeed an instance of $\mathfrak{S}$, we need to construct the pullback for each constraint (p, δ) in $C^{\mathfrak{S}}$. In this example, we validate three of these constraints: $(\mathtt{[mult(1,}\infty\mathtt{)]}, \delta_1)$, $(\mathtt{[composition]}, \delta_4)$ and $(\mathtt{[image\text{-}inclusion]}, \delta_5)$, but, we show the pullback only for the first one since the technique is similar.

First, we look at the constraint $(\mathtt{[mult(1,}\infty\mathtt{)]}, \delta_1)$. The pullback of $\alpha(\mathtt{[mult(1,}\infty\mathtt{)]}) \xrightarrow{\delta_1} S \xleftarrow{\iota} I$ will be $\alpha(\mathtt{[mult(1,}\infty\mathtt{)]}) \xleftarrow{\iota^*} O^* \xrightarrow{\delta_1^*} I$, where $\iota^* : O^* \to \alpha(\mathtt{[mult(1,}\infty\mathtt{)]})$ is as in Fig. 3.3. The graph homomorphism ι^* is a valid instance of the predicate $\mathtt{[mult(1,}\infty\mathtt{)]}$ since the semantics of the constraint which is set by the predicate is not violated. The constraint, which holds in this case, is that

$$\forall emp \in \iota^{-1}(Employee) \exists dep \in \iota^{-1}(Department) : dep \in \iota^{-1}(empDeps)(emp)$$

Moreover, we consider the composition constraint $(\mathtt{[composition]}, \delta_4)$. Note that this constraint is special in the sense that it indicates derived information which is used later by the constraint $(\mathtt{[image\text{-}inclusion]}, \delta_5)$. To which

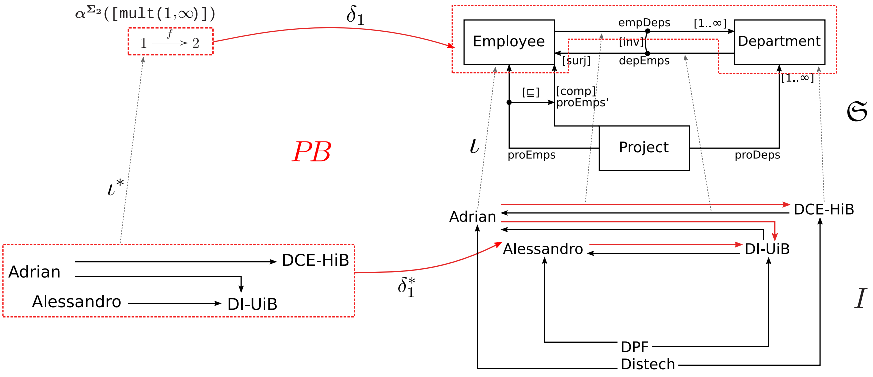


Figure 3.3: The specification $\mathfrak{S}$ and its instance from Fig.3.2 together with the pullback $\alpha(\texttt{[mult(1,}\infty\texttt{)]}) \xleftarrow{\iota^*} O^* \xrightarrow{\delta_1^*} I$ of $\alpha(\texttt{[mult(1,}\infty\texttt{)]}) \xrightarrow{\delta_1} S \xleftarrow{\iota} I$

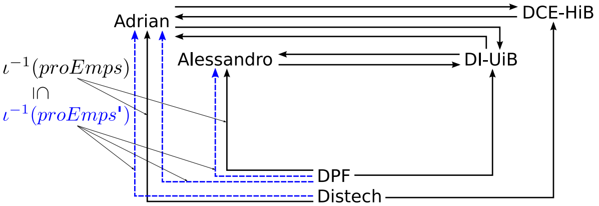


Figure 3.4: The instance I from Figs. 3.2 and 3.3 together with the arrows $\iota^{-1}(proEmps')$ shown as dashed, blue arrows

extent derived information is stored or represented explicitly in instances depends on the context; e.g. derived information may be stored explicitly if its calculation is time-consuming. In this example, the arrows $\iota^{-1}(proEmps')$ are omitted in I, but shown as dashed, blue arrows in Fig. 3.4 for explanation purposes. Getting back to the main topic of the example, we can check that this constraint is not violated by verifying that

$$\iota^{-1}(proEmps') = \iota^{-1}(proDeps); \iota^{-1}(depEmps)$$

Finally, we will look at the subset constraint ([image-inclusion], δ_5). We can check that this constraint is not violated by verifying that

$$\forall pro \in \iota^{-1}(Project) : \iota^{-1}(proEmps)(pro) \subseteq \iota^{-1}(proEmps')(pro)$$

Intuitively, if the graph I contained an arrow connecting the nodes Distech and Alessandro (shown as a red, dotted arrow in Fig. 3.2b), it would not be a valid instance of $\mathfrak{S}$ since it would violate the constraint formulated by the `[image-inclusion]` predicate. This arrow represents the information "the employee Alessandro is involved in the project Distech", but, according to requirement 5 "the employee Alessandro can not be involved in the project Distech because he does not work for the controlling department DCE-HiB". ♢

Remark 7 (Instance of Specification vs Instance of Graph) *Note that the graph G_0 in Fig. 2.1d resembles the graph I in Fig. 3.2b. The graph G_0 is a valid instance of the graph G_1 since there exists a typing graph homomorphism from G_0 to G_1. However, although one could find a graph homomorphism from G_0 to $\mathfrak{S}$, the graph G_0 does not satisfy all the constraints $C^{\mathfrak{S}}$. Some of these constraints, which are not satisfied by G_0 in Fig. 2.1d, are:*

- *the constraint $(\texttt{[image-inclusion]}, \delta_5)$ is not satisfied since the node "Alessandro" is the target of an arrow from "Distech" while there is no path from "Distech" to "Alessandro" via a node which is typed by* Department*.*
- *the constraint $(\texttt{[inverse]}, \delta_6)$ is not satisfied since the node "DI-UiB" has an outgoing arrow to the node "Alessandro" while there is no arrow going in the other direction.*

Implicit conjunction

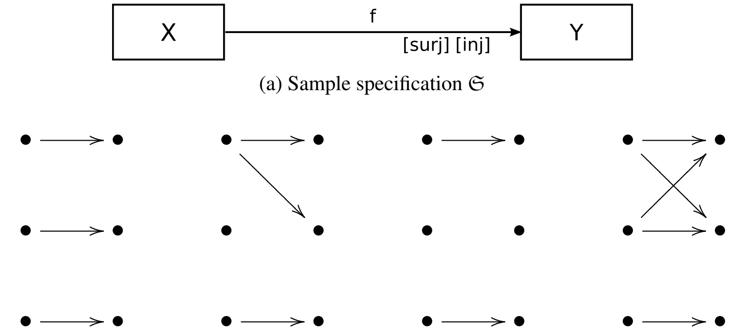
It is important to note that there is an implicit relation between the constraints of a specification: for any specification $(S, C^{\mathfrak{S}} : \Sigma)$, due to Definition 11 there is an implicit conjunctive connection between the constraints in $\{(p_1, \delta_1), \ldots, (p_n, \delta_n)\} \in C^{\mathfrak{S}}$. The following example illustrates this.

Example 12 (Conjunction of Constraints) For the specification $(S, C^{\mathfrak{S}} : \Sigma)$ in Fig. 3.5a, we have $(\texttt{[injective]}, \delta)$ and $(\texttt{[surjective]}, \delta)$ on the same arrow f. Two graphs which satisfy the atomic constraints $C^{\mathfrak{S}}$ are shown in Fig. 3.5b and 3.5c. These graphs are valid instances since the arrows $\iota^{-1}(f)$ represent both injective and surjective functions according to the definitions in Table 3.1. Moreover, two graphs which do not satisfy the atomic constraints $C^{\mathfrak{S}}$ are shown in Fig. 3.5d and 3.5e. These graphs are invalid instances since the arrows $\iota^{-1}(f)$ in (d) do not represent surjective functions; and in (e) they do not represent injective functions. ♢

Next, for a given specification $\mathfrak{S}$, the category of instances of $\mathfrak{S}$ is defined as follows.

Definition 12 (Category of Instances) *For any specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$, we obtain a category $\mathbf{Inst}(\mathfrak{S})$ with objects all instances (I, ι) of $\mathfrak{S}$ and morphisms between instances $\phi : (I, \iota) \to (I', \iota')$ are graph homomorphisms $\phi : I \to I'$ such that $\phi; \iota' = \iota$.*

$$\begin{tikzcd}
& S \arrow[d, phantom, "="] & \\
I \arrow[ur, "\iota"] \arrow[rr, "\phi"] & {} & I' \arrow[ul, "\iota'"']
\end{tikzcd}$$



(a) Sample specification $\mathfrak{S}$

(b) I_1: Injective and surjective

(c) I_2: Injective and surjective

(d) I_3: Not surjective

(e) I_4: Not injective

Figure 3.5: A sample specification $\mathfrak{S}$ together with four graphs of which two satisfying the atomic constraints $C^{\mathfrak{S}}$ and two not satisfying these constraints

Moreover, $\mathbf{Inst}(\mathfrak{S})$ *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 [15]. That is, we have an inclusion functor* $inc^{\mathfrak{S}} : \mathbf{Inst}(\mathfrak{S}) \to \mathbf{Inst}(S)$.

3.4 Relations between Specifications

This section discusses how relations between specifications are defined, and what features and properties these relations possess. We give a stepwise definition of these relations: first between the underlying graphs of specifications; second, specification morphisms; and finally, specification entailment.

3.4.1 Translation of Typed Graphs

Since the underlying structure of specifications are graphs, the first kind of relations we may describe between specifications are graph homomorphisms between the underlying graphs of specifications. These graph homomorphisms will induce a translation of instances of graphs; i.e. they induce a change of typing.

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* $(I, \iota) \in \mathbf{Inst}(S)$

$$\begin{tikzcd}
S \arrow[r, "\phi"] & S' \\
I \arrow[u, "\iota"] \arrow[ur, "\iota;\phi"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{Inst}(S) \arrow[r, "\phi_\bullet"] & \mathbf{Inst}(S')
\end{tikzcd}$$

Furthermore, 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 construction of pullbacks

$$\begin{tikzcd}
S \arrow[r, "\phi"] & S' \\
I^* \arrow[u, "{\iota'}^*"] \arrow[r, "\phi^*"'] \arrow[ur, phantom, "PB"] & I' \arrow[u, "\iota'"']
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{Inst}(S) & \mathbf{Inst}(S') \arrow[l, "\phi^\bullet"']
\end{tikzcd}$$

Proof. The proof of $\phi_\bullet$ is given by the composition $\iota;\phi$ of graph homomorphisms. The proof of $\phi^\bullet$ is given by pullback in category **Graph** as shown in [38]. ♢

3.4.2 SPECIFICATION MORPHISMS

Another relation between specifications in DPF is represented by specification morphisms. These specification morphisms are graph homomorphisms – between the underlying graphs of specifications – which preserve atomic constraints.

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

$$\begin{tikzcd}
\alpha^{\Sigma}(p) \arrow[r, "\delta"] \arrow[rr, bend left, "\delta;\phi"] \arrow[rr, phantom, bend left=20, "="] & S \arrow[r, "\phi"] & S'
\end{tikzcd}$$

Remark 8 (Inclusion Specification Morphism) *A specification $\mathfrak{S}$ is a subspecification of a specification $\mathfrak{S}'$, written $\mathfrak{S} \sqsubseteq \mathfrak{S}'$, iff S is a subgraph of S' and the inclusion graph homomorphism $inc : S \hookrightarrow S'$ defines a specification morphism $inc : \mathfrak{S} \hookrightarrow \mathfrak{S}'$.*

A closer look at Definition 13 shows that it is implicitly based on a translation of specifications induced by graph homomorphisms.

Remark 9 (Graph Homomorphism and Constraints) *Any graph homomorphism $\phi : S \to S'$ induces a constraint translation. That is, for any specification $\mathfrak{S} = (S, C^{\mathfrak{S}} \colon \Sigma)$ we obtain a specification $\phi(\mathfrak{S}) = (S', C^{\phi(\mathfrak{S})} \colon \Sigma)$ with $C^{\phi(\mathfrak{S})} = \phi(C^{\mathfrak{S}}) = \{(p, \delta; \phi) \mid (p, \delta) \in C^{\mathfrak{S}}\}$*

$$\begin{tikzcd}
\alpha^{\Sigma}(p) \arrow[r, "\delta"] \arrow[rr, bend left, "\delta;\phi"] \arrow[rr, phantom, bend left=20, "="] & S \arrow[r, "\phi"] & S'
\end{tikzcd}$$

The condition for specification morphisms can now be reformulated as follows: a specification morphism $\phi : \mathfrak{S} \to \mathfrak{S}'$ is a graph homomorphism $\phi : S \to S'$ such that $\phi(\mathfrak{S}) \sqsubseteq \mathfrak{S}'$. That is, $C^{\phi(\mathfrak{S})} = \phi(C^{\mathfrak{S}}) \subseteq C^{\mathfrak{S}'}$.

Having defined specifications and specification morphisms, the category of specifications is defined as follows.

Definition 14 (Category of Specifications) *For any signature Σ, we obtain a category $\mathsf{Spec}(\Sigma)$ of all specifications $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ and all specification morphisms. The associativity of graph homomorphism composition ensures that the composition of two specification morphisms becomes a specification morphism as well. It ensures also that the composition of specification morphisms is associative. Furthermore, the identity graph homomorphisms $id^S : S \to S$ define identity specification morphisms $id^{\mathfrak{S}} : \mathfrak{S} \to \mathfrak{S}$, which are 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 $\phi^{\bullet}(\mathsf{Inst}(\mathfrak{S}')) \subseteq \mathsf{Inst}(\mathfrak{S})$, that is, the functor $\phi^{\bullet} : \mathsf{Inst}(S') \to \mathsf{Inst}(S)$ restricts to a functor $\phi^{\bullet} : \mathsf{Inst}(\mathfrak{S}') \to \mathsf{Inst}(\mathfrak{S})$*

$$\begin{tikzcd}
S \arrow[d, "\phi"'] & \mathsf{Inst}(S) & & \mathsf{Inst}(\mathfrak{S}) \arrow[ll, hook'] & \mathfrak{S} \arrow[d, "\phi"] \\
S' & \mathsf{Inst}(S') \arrow[u, "\phi^{\bullet}"] \arrow[urr, phantom, "="] & & \mathsf{Inst}(\mathfrak{S}') \arrow[ll, hook'] \arrow[u, "\phi^{\bullet}"'] & \mathfrak{S}'
\end{tikzcd}$$

Proof. As shown in [38], the proof follows from the result that the composition of two pullbacks is again a pullback [15] and from our assumption that $[\![p]\!]$ is closed under isomorphisms (see Definition 8).

$$\begin{tikzcd}
\alpha^{\Sigma}(p) \arrow[r, "\delta"] \arrow[rr, bend left, "\delta;\phi"] & S \arrow[r, "\phi"] & S' \\
O \arrow[u, "\iota^*"'] \arrow[r, "\delta^*"'] \arrow[rr, bend right, "\delta^*;\phi^*"'] \arrow[ur, phantom, "PB"] & I \arrow[u, "\iota"'] \arrow[r, "\phi^*"'] \arrow[ur, phantom, "PB"] & I' \arrow[u, "\iota'"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\alpha^{\Sigma}(p) \arrow[r, "\delta;\phi"] & S' \\
O \arrow[u, "\iota^*"'] \arrow[r, "\delta^*;\phi^*"'] \arrow[ur, phantom, "PB"] & I' \arrow[u, "\iota'"']
\end{tikzcd}$$

◇

3.4.3 SPECIFICATION ENTAILMENT

A third kind of relations between specifications is *entailment*. A specification entailment has the structure $Premise \vdash Conclusion$, where both premise and conclusion are specifications with the same underlying graph. We use specification entailments to express properties of predicates (see Remark 4).

Definition 15 (Specification Entailment) *A specification entailment $\mathfrak{L} \vdash \mathfrak{R}$ is given by two specifications $\mathfrak{L} = (L, C^{\mathfrak{L}} : \Sigma)$ and $\mathfrak{R} = (R, C^{\mathfrak{R}} : \Sigma)$ with the same underlying graph $L = R$ called the context graph.*

$$\begin{tikzcd}
 & \mathfrak{L} & & \vdash & & \mathfrak{R} & \\
\alpha(p_1) \arrow[rrrd, swap, "\delta_{p_1}"] & \cdots & \alpha(p_n) \arrow[rd, "\delta_{p_n}"] & \vdash & \alpha(q_1) \arrow[ld, swap, "\delta_{q_1}"] & \cdots & \alpha(q_m) \arrow[llld, "\delta_{q_m}"] \\
 & & & L = R & & &
\end{tikzcd}$$

Specification entailments are devoted to describe or to require properties of the semantic interpretation of predicates. A specification entailment is valid if and only if all instances of the premise are also instances of the conclusion.

Definition 16 (Semantic Interpretation and Specification Entailment) *A specification entailment $\mathfrak{L} \vdash \mathfrak{R}$ with $\mathfrak{L} = (L, C^{\mathfrak{L}} : \Sigma)$ and $\mathfrak{R} = (R, C^{\mathfrak{R}} : \Sigma)$ is valid for a semantic interpretation of predicates $[\![..]\!]^{\Sigma}$ iff $\mathsf{Inst}(\mathfrak{L}) \subseteq \mathsf{Inst}(\mathfrak{R})$.*

Remark 10 (Unique Context for an Entailment) *It is not necessary to consider different contexts within one specification entailment. That is, given $\mathfrak{L} = (L, C^{\mathfrak{L}} : \Sigma)$ and $\mathfrak{R} = (R, C^{\mathfrak{R}} : \Sigma)$ and a graph homomorphism $\phi : R \to L$, we obtain a specification entailment $\mathfrak{L} \vdash \phi(\mathfrak{R})$ and the two conditions $\mathsf{Inst}(\mathfrak{L}) \subseteq \mathsf{Inst}(\phi(\mathfrak{R}))$ and $\phi^{\bullet}(\mathsf{Inst}(\mathfrak{L})) \subseteq \mathsf{Inst}(\mathfrak{R})$ are equivalent due to the composition properties of pullbacks.*

Example 13 (Specification Entailment) Fig. 3.6 shows a specification entailment $\mathfrak{L} \vdash \mathfrak{R}$ with

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

$$\begin{tikzcd}
L = R = \boxed{\mathsf{X}} \arrow[r, bend left, "\mathsf{f}"] & \boxed{\mathsf{Y}} \arrow[l, bend left, "\mathsf{g}"]
\end{tikzcd}$$

and the following assignments:

$\delta_1(1) = X \quad \delta_1(2) = Y \quad \delta_1(f) = f$
$\delta_2(1) = X \quad \delta_2(2) = Y \quad \delta_2(f) = f \quad \delta_2(g) = g$
$\delta_3(1) = X \quad \delta_3(2) = Y \quad \delta_3(g) = g$

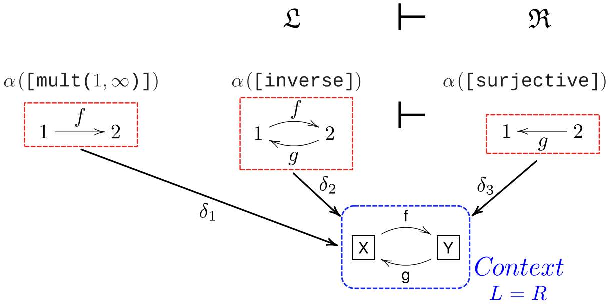


Figure 3.6: A sample specification entailment $\mathfrak{L} \vdash \mathfrak{R}$

The predicates [mult(1,∞)], [inverse] and [surjective] come from the signature Σ_2 in Table 3.1. We can easily show that the semantic interpretation given in Table 3.1 satisfies $\textsf{Inst}(\mathfrak{L}) \subseteq \textsf{Inst}(\mathfrak{R})$ as follows:

Since $\forall x \in X : |f(x)| \geq 1$, we have that $\forall x \in X : \exists y \in Y : y \in f(x)$. And since $\forall x \in X$ and $\forall y \in Y : x \in g(y)$ iff $y \in f(x)$ we have that $\forall x \in X \exists y \in Y : x \in g(y)$ which leads to $\forall x \in X : \bigcup\{g(y)\} = X$, i.e. g is surjective. ♢

Remark 11 (Specification Entailment vs Predicate Dependencies) *As mentioned in Remark 4, the relation between predicates are described as predicate dependencies in [38; 127]. A predicate dependency* $p \vdash^{r} q$ *with the arities* $\alpha(r) : \alpha(q) \to \alpha(p)$ *can be represented by a specification entailment* $\mathfrak{L} \vdash \mathfrak{R}$ *where*

$$\mathfrak{L} = (L = \alpha(p), C^{\mathfrak{L}} = \{(p, id_{\alpha(p)})\} : \Sigma)$$
$$\mathfrak{R} = (R = \alpha(p), C^{\mathfrak{R}} = \{(q, \alpha(r))\} : \Sigma).$$

As mentioned, each specification entailment is defined over a given context. Once a specification entailment is defined, the context can be embedded into other specifications by means of specification morphisms. As it will be clear in Section 4.4, this embedding can be used to formulate "universal constraints" on specifications.

Proposition 3 (Embedding) *For any specification entailment* $\mathfrak{L} \vdash \mathfrak{R}$, *with* $\mathfrak{L} = (L, C^{\mathfrak{L}} : \Sigma)$ *and* $\mathfrak{R} = (R, C^{\mathfrak{R}} : \Sigma)$, *and any specification morphism* $\phi : \mathfrak{L} \to \mathfrak{S}$, *with* $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$, *we obtain an induced specification entailment* $\mathfrak{S} \vdash (\mathfrak{S} \cup \phi(\mathfrak{R}))$. *For a given semantic interpretation* $[\![..]\!]^{\Sigma}$, *the induced entailment* $\mathfrak{S} \vdash (\mathfrak{S} \cup \phi(\mathfrak{R}))$ *is valid as long as* $\mathfrak{L} \vdash \mathfrak{R}$ *is valid.*

$$\begin{tikzcd}
\mathfrak{L} \arrow[r, |-] \arrow[d, "\phi"'] & \mathfrak{R} \arrow[d, "\phi"] \\
\mathfrak{S} \arrow[r, |-] & \mathfrak{S} \cup \phi(\mathfrak{R})
\end{tikzcd}$$

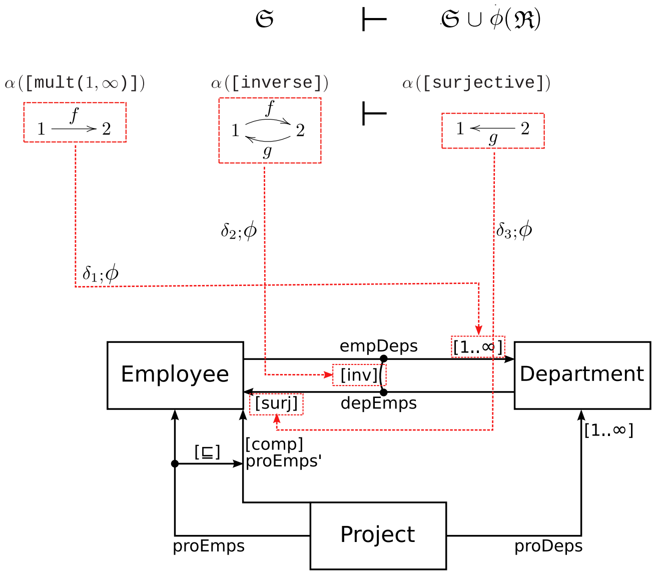


Figure 3.7: A sample specification entailment $\mathfrak{S} \vdash (\mathfrak{S} \cup \phi(\mathfrak{R}))$

Proof.

We have to show $\mathsf{Inst}(\mathfrak{S}) \subseteq \mathsf{Inst}(\mathfrak{S} \cup \phi(\mathfrak{R}))$.
Since $\mathsf{Inst}(\mathfrak{S} \cup \phi(\mathfrak{R})) = \mathsf{Inst}(\mathfrak{S}) \cap \mathsf{Inst}(\phi(\mathfrak{R}))$,
the inclusion $\mathsf{Inst}(\mathfrak{S}) \subseteq \mathsf{Inst}(\mathfrak{S} \cup \phi(\mathfrak{R}))$
is equivalent to $\mathsf{Inst}(\mathfrak{S}) \subseteq \mathsf{Inst}(\phi(\mathfrak{R}))$.
Due to Remark 10, $\mathsf{Inst}(\mathfrak{S}) \subseteq \mathsf{Inst}(\phi(\mathfrak{R}))$
is equivalent to $\phi^{\bullet}(\mathsf{Inst}(\mathfrak{S})) \subseteq \mathsf{Inst}(\mathfrak{R})$.
This inclusion follows from the assumptions
by Proposition 2 $\phi^{\bullet}(\mathsf{Inst}(\mathfrak{L})) \subseteq \mathsf{Inst}(\mathfrak{L})$
and Definition 16 $\mathsf{Inst}(\mathfrak{L}) \subseteq \mathsf{Inst}(\mathfrak{R})$

◊

Example 14 (Embedded Entailment) Building on Examples 13 and 10, Fig. 3.7 shows a specification entailment $\mathfrak{S} \vdash (\mathfrak{S} \cup \phi(\mathfrak{R}))$ which is given by the embedding $\phi : \mathfrak{L} \to \mathfrak{S}$ of the specification entailment in Example 13 into the specification $\mathfrak{S}$ from Example 10. ◊

CHAPTER 4

Metamodelling in DPF

So far, we have formalised the concepts *model* and *instances of a model* in terms of DPF. In this chapter, we use these concepts to formalise the relation between models in a metamodelling hierarchy. In addition, we discuss the usage of DPF in the formalisation of MOF-based modelling languages.

4.1 Introduction

In the context of MDE, metamodelling is the *de facto* standard technique used to define the abstract syntax of modelling languages. Any model which is defined by the modelling language should conform to the language's corresponding metamodel; or put another way, all models which conform to the corresponding metamodel are considered syntactic correct models defined by the modelling language.

In Chapter 3, we formalised the concepts *model* and *instances of a model* by defining the syntax and the semantics of diagrammatic specifications, respectively. Considering the case in which the instances of a model are also models, i.e. relating these concepts to a metamodelling hierarchy, the syntax of a model is treated as a semantic entity which is specified and constrained by another model at a higher level of the hierarchy. In terms of DPF, metamodelling refers to this recursive shift from syntax to semantics.

As a first step, we look at the relation between models and metamodels. In order to be more generic, we consider the relation between models located at any two adjacent levels of a metamodelling hierarchy. We call this relation for *conformance*. Afterwards, we show how modelling languages are formalised as modelling formalisms in DPF, and how metamodels and modelling formalisms are related.

4.2 Conformance Relation

TYPED BY VS CONFORMS TO

This section deals with the DPF based formalisation of the conformance relation between models in a metamodelling hierarchy (see Fig.1.5). In DPF, we use specifications to represent models at any level of a metamodelling hierarchy. Moreover, we distinguish between two types of conformance relations: *typed by* and *conforms to*. A specification $\mathfrak{S}_n$ at level n is typed by a specification $\mathfrak{S}_{n+1}$ at level $n+1$ if there exists a typing morphism $\iota^{S_n} : S_n \to S_{n+1}$ between the underlying graphs of the specifications. This corresponds to the relation between a model and its metamodel in the graph-based formalisation of the metamodelling hierarchy (see Section 2.2). In contrast, a specification $\mathfrak{S}_n$ at level n is said to conform to a specification $\mathfrak{S}_{n+1}$ at level $n+1$ if there exists a typing morphism $\iota^{S_n} : S_n \to S_{n+1}$ such that (S_n, ι^{S_n}) is an instance of $\mathfrak{S}_{n+1}$ [106]. That is, in addition to the existence of the typing morphism ι^{S_n}, the constraints $C^{\mathfrak{S}_{n+1}}$ are satisfied by (S_n, ι^{S_n}).

TYPED SPECIFICATION

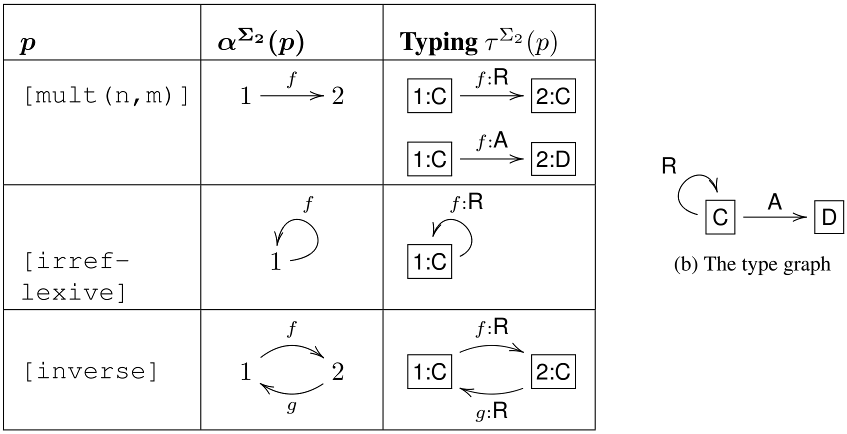
A *typed* specification may be typed by any graph, i.e. not only the underlying graph of another specification. A specification typed by a graph G is a structure which consists of an underlying graph that is typed by G, together with a set of typed atomic constraints. Typed atomic constraints are formulated with help of typed signatures. The “untyped” versions of signatures, atomic constraints and specifications are already defined in Chapter 3. The typed versions are defined as follows.

Definition 17 (Typed Signature) *A signature typed by a graph G is a signature $\Sigma = (P^\Sigma, \alpha^\Sigma)$ together with a map τ^Σ assigning to each predicate $p \in P^\Sigma$ a graph homomorphism $\tau^\Sigma(p) : \alpha^\Sigma(p) \to G$. $\tau^\Sigma(p)$ is called the typing of p. We use $\Sigma \rhd G$ or $((P^\Sigma, \alpha^\Sigma) \rhd_{\tau^\Sigma} G)$ to denote a signature Σ typed by G.*

$$\begin{tikzcd} \alpha^\Sigma(p) \arrow[r, "\tau^\Sigma(p)"] & G \end{tikzcd} \qquad \begin{tikzcd} \Sigma \arrow[r, dotted, "\tau^\Sigma"] & G \end{tikzcd}$$

Example 15 (Typed Signature) Fig. 4.1a shows a part of the signature Σ_2 from Table 3.1 as a typed signature. The typing graph G chosen for this purpose is shown in Fig. 4.1b. The column **Typing** $\tau^{\Sigma_2}(p)$ of the signature represents the possible types for the predicates. According to the typing, the predicate `[mult(n,m)]` may be used to add constraints to arrows which are typed by any of the arrows A or R, as shown in the **Typing** column. Moreover, the predicate `[irreflexive]` may only be used to add constraints to model elements which are typed by the loop structure with the node C and the arrow R. ♢

Remark 12 (Predicate Names and Typing) *Recall the observation in Remark 3 about predicate names. Some of the predicate names in Σ_2 in Table 3.1 refer to unique predicates, while some others refer to a family of predicates. For typed signatures, these names will have yet another dimension based on the typing map. For example, the predicate `[mult(n,m)]` in Fig. 4.1a may be added to arrows of type A or R.*



(b) The type graph

(a) Typed signature

Figure 4.1: A sample typed signature $\Sigma_2 \triangleright G$ and the type graph G

Definition 18 (Typed Atomic Constraint) *Given a typed signature $((P^\Sigma, \alpha^\Sigma) \triangleright_{\tau^\Sigma} G)$, a typed atomic constraint (p, δ) added to a typed graph (S, ι^S) with $\iota^S : S \to G$ is given by a predicate symbol p and a graph homomorphism $\delta : \alpha^\Sigma(p) \to S$ such that $\delta; \iota^S = \tau^\Sigma(p)$.*

$$\begin{tikzcd}
\alpha^\Sigma(p) \arrow[r, "\tau^\Sigma(p)"] \arrow[dr, "\delta"'] \arrow[dr, phantom, "=", shift left=3] & G \\
& S \arrow[u, "\iota^S"']
\end{tikzcd}$$

Definition 19 (Typed Specification) *Given a graph G and a typed signature $\Sigma \triangleright G$, a specification typed by the graph G is a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ together with a typing graph homomorphism $\iota^S : S \to G$ assigning to each element of S a type in G such that $\forall (p, \delta) \in C^{\mathfrak{S}} : \delta; \iota^S = \tau^\Sigma(p)$. We write $\mathfrak{S} \triangleright G$ or $((S, C^{\mathfrak{S}} : \Sigma) \triangleright_{\iota^S} G)$ to denote a specification $\mathfrak{S}$ typed by G.*

$$\begin{tikzcd}
\alpha^\Sigma(p) \arrow[r, "\tau^\Sigma(p)"] \arrow[dr, "\delta"'] \arrow[dr, phantom, "=", shift left=3] & G \\
& S \arrow[u, "\iota^S"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\Sigma \arrow[r, dotted, "\tau^\Sigma"] \arrow[dr, dashed, "C^{\mathfrak{S}}"'] \arrow[dr, phantom, "=", shift left=3] & G \\
& S \arrow[u, "\iota^S"']
\end{tikzcd}$$

Remark 13 (Semantics of Typed Specifications) *The semantics of the typed versions of predicates and specifications is defined in the same way as for the corresponding untyped versions. This is because, whether a specification is typed by a graph will only affect the relation between the specification and the graph, not the instances of the specification.*

CONFORMANT SPECIFICATION

Now we can define conformant specifications as typed specifications which satisfy the constraints of another specification. In the following definition, we use the indices 1, 2 and 3 in order to reflect the levels in the metamodelling hierarchy.

Definition 20 (Conformant Specification) *Let $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2}\!:\!\Sigma_3)$ be a specification with $\Sigma_3 = (P^{\Sigma_3}, \alpha^{\Sigma_3})$, and $\Sigma_2 \rhd S_2 = ((P^{\Sigma_2}, \alpha^{\Sigma_2}) \rhd_{\tau^{\Sigma_2}} S_2)$ a signature. A typed specification $\mathfrak{S}_1 = ((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \rhd_{\iota^{S_1}} S_2)$ conforms to $\mathfrak{S}_2$ iff $(S_1, \iota^{S_1}) \in \textsf{\textbf{Inst}}(\mathfrak{S}_2)$. We write $\mathfrak{S}_1 \blacktriangleright \mathfrak{S}_2$ or $((S_1, C^{\mathfrak{S}_1}\!:\Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$ to denote a specification $\mathfrak{S}_1$ conformant to $\mathfrak{S}_2$.*

$$\begin{tikzcd}
\Sigma_3 \arrow[dr, dashed, "C^{\mathfrak{S}_2}"] & \\
\Sigma_2 \arrow[r, dotted, "\tau^{\Sigma_2}"] \arrow[dr, dashed, "C^{\mathfrak{S}}1"'] & S_2 \\
& S_1 \arrow[u, "\iota^{S_1}"']
\end{tikzcd}$$

Recall that we defined specification morphisms as relations between the untyped version of specifications (see Definition 13). Now we generalise this definition for the typed version of specifications.

Definition 21 (Typed Specification Morphism) *Given a graph G and a typed signature $\Sigma \rhd G$, a typed specification morphism between two typed specifications $\phi : ((S, C^{\mathfrak{S}} : \Sigma) \rhd_{\iota^S} G) \to ((S', C^{\mathfrak{S}'} : \Sigma) \rhd_{\iota^{S'}} G)$ is a specification morphism $\phi : (S, C^{\mathfrak{S}}\!:\!\Sigma) \to (S', C^{\mathfrak{S}'}\!:\!\Sigma)$ such that $\phi; \iota^{S'} = \iota^S$.*

$$\begin{tikzcd}
& G & \\
S \arrow[ur, "\iota^S"] \arrow[rr, "\phi"] & = & S' \arrow[ul, "\iota^{S'}"']
\end{tikzcd}$$

Having defined the concepts of typed specification and typed specification morphism, the category of typed specifications is defined as follows.

Definition 22 (Category of Typed Specifications) *For any graph G and any typed signature $\Sigma \rhd G$ we obtain the category $\textsf{\textbf{TSpec}}(\Sigma \rhd G)$ of all typed specifications $((S, C^{\mathfrak{S}}\!:\!\Sigma) \rhd_{\iota^S} G)$ and all typed specification morphisms $\phi : \mathfrak{S} \rhd G \to \mathfrak{S}' \rhd G$.*

Moreover, according to the definitions of (typed) specifications and (typed) specification morphisms, there exists a functor $U^G : \textsf{\textbf{TSpec}}(\Sigma \rhd G) \to \textsf{\textbf{Spec}}(\Sigma)$ with $U^G((S, C^{\mathfrak{S}} : \Sigma) \rhd_{\iota^S} G) = (S, C^{\mathfrak{S}} : \Sigma)$ for all specifications $\mathfrak{S}$ typed by G, and with $U^G(\phi) = \phi : (S, C^{\mathfrak{S}} : \Sigma) \to (S', C^{\mathfrak{S}'} : \Sigma)$ for all typed specification morphisms $\phi : ((S, C^{\mathfrak{S}}\!:\!\Sigma) \rhd_{\iota^S} G) \to ((S', C^{\mathfrak{S}'}\!:\!\Sigma) \rhd_{\iota^{S'}} G)$.

$$\begin{tikzcd}
\textsf{\textbf{TSpec}}(\Sigma \rhd G) \arrow[r, "U^G"] & \textsf{\textbf{Spec}}(\Sigma)
\end{tikzcd}$$

Analogously, having defined the concepts of conformant specification and typed specification morphism, the category of conformant specifications is defined as follows.

Definition 23 (Category of Conformant Specifications) *For any specification* $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2}\colon \Sigma_3)$ *and any signature* $\Sigma_2 \rhd S_2$ *we obtain a category* $\mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2)$ *of all conformant specifications* $((S_1, C^{\mathfrak{S}_1}\colon \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$ *and all typed specification morphisms* $\phi : \mathfrak{S}_1 \rhd S_2 \to \mathfrak{S}'_1 \rhd S_2$*.*

Moreover, $\mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2)$ *is a full subcategory of* $\mathsf{TSpec}(\Sigma_2 \rhd S_2)$*. That is, we have an inclusion functor* $inc^{\mathfrak{S}_2} : \mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2) \to \mathsf{TSpec}(\Sigma_2 \rhd S_2)$*. Putting this together with Definition 22, we obtain the following diagram*

$$\begin{tikzcd}
\mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2) \arrow[r, "inc^{\mathfrak{S}_2}"] \arrow[dr, "inc^{\mathfrak{S}_2};U^{S_2}"'] \arrow[dr, phantom, "=", shift left=3] & \mathsf{TSpec}(\Sigma_2 \rhd S_2) \arrow[d, "U^{S_2}"] \\
 & \mathsf{Spec}(\Sigma)
\end{tikzcd}$$

The objects in the category of typed specifications $\mathsf{TSpec}(\Sigma_2 \rhd S_2)$ are specifications $\mathfrak{S}_1$ which are typed by S_2. That is, we have $S_1 \in \mathsf{Inst}(S_2)$. Analogously, the objects in the category of conformant specifications $\mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2)$ are specifications $\mathfrak{S}_1$ which conform to $\mathfrak{S}_2$. That is, we have $S_1 \in \mathsf{Inst}(\mathfrak{S}_2)$. Putting this together with Definition 12, we obtain the following remark.

Remark 14 (Categories $\mathsf{TSpec}(\Sigma_2 \rhd S_2)$, $\mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2)$, $\mathsf{Inst}(S_2)$, $\mathsf{Inst}(\mathfrak{S}_2)$) *Given the inclusion functor* $inc^{\mathfrak{S}} : \mathsf{Inst}(\mathfrak{S}) \to \mathsf{Inst}(S)$ *from Definition 12 and the inclusion functor* $inc^{\mathfrak{S}_2} : \mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2) \to \mathsf{TSpec}(\Sigma_2 \rhd S_2)$ *from Definition 23, we obtain the following diagram*

$$\begin{tikzcd}
\mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2) \arrow[r, "inc^{\mathfrak{S}_2}"] \arrow[d, "inst^{\mathfrak{S}_2}"'] \arrow[dr, phantom, "="] & \mathsf{TSpec}(\Sigma_2 \rhd S_2) \arrow[d, "inst^{S_2}"] \\
\mathsf{Inst}(\mathfrak{S}_2) \arrow[r, "inc^{\mathfrak{S}_2}"'] & \mathsf{Inst}(S_2)
\end{tikzcd}$$

Recall that we defined specification entailments as relations between the untyped version of specifications (see Definition 15). Now we generalise this definition for the typed version of specifications.

Definition 24 (Typed Specification Entailment) *Given a graph* G *and a signature* $\Sigma \rhd G$*, a typed specification entailment* $\mathfrak{L} \rhd G \vdash \mathfrak{R} \rhd G$ *is given by two typed specifications* $\mathfrak{L} \rhd G = ((L, C^{\mathfrak{L}}\colon \Sigma) \rhd_{\iota^L} G)$ *and* $\mathfrak{R} \rhd G = ((R, C^{\mathfrak{R}}\colon \Sigma) \rhd_{\iota^R} G)$ *with the same context graph* $L = R$*.*

$$\begin{tikzcd}
 & \mathfrak{L}\rhd G & & \vdash & & \mathfrak{R}\rhd G & \\
\alpha(p_1) \arrow[drrr, "\delta_{p_1}"'] & \cdots & \alpha(p_n) \arrow[dr, "\delta_{p_n}"] & \vdash & \alpha(q_1) \arrow[dl, "\delta_{q_1}"'] & \cdots & \alpha(q_m) \arrow[dlll, "\delta_{q_m}"] \\
 & & & L=R \arrow[d, "\iota^L=\iota^R"] & & & \\
 & & & G & & &
\end{tikzcd}$$

Figure 4.2: Typed specification entailment

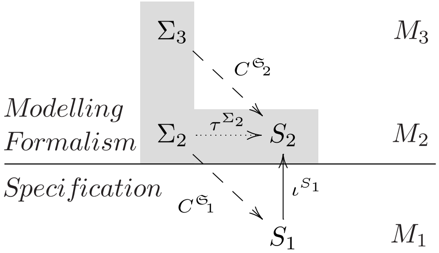


Figure 4.3: Modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ together with a specification $((S_1, C^{\mathfrak{S}_1}\colon \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$

4.3 Modelling Formalisms

In DPF, each modelling language is formalised as a modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$. The corresponding metamodel of the modelling language is represented by the specification $\mathfrak{S}_2$ which has its constraints formulated by predicates from the signature Σ_3. The atomic constraining constructs which are available for the users of the modelling language are located in the typed signature $\Sigma_2 \rhd S_2$. Fig. 4.3 shows a modelling formalism and its alignment with the metamodelling hierarchy from OMG. In addition, the figure shows the relation between the modelling formalism and a specification $\mathfrak{S}_1$ specified by the formalism.

Definition 25 (Modelling Formalism) *A modelling formalism* $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ *is given by two signatures* $\Sigma_2 = ((P^{\Sigma_2}, \alpha^{\Sigma_2}) \rhd_{\tau^{\Sigma_2}} S_2)$ *and* $\Sigma_3 = (P^{\Sigma_3}, \alpha^{\Sigma_3})$ *and a specification* $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2}\colon \Sigma_3)$ *which is called the corresponding metamodel of the modelling formalism.*

Remark 15 (Modelling Formalisms and the Metamodelling Hierarchy) *We have chosen the indexes 1,2 and 3 in order to reflect OMG's metamodelling hierarchy*

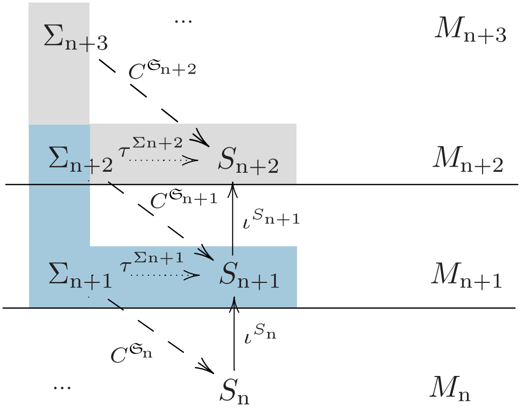


Figure 4.4: Modelling formalisms and the metamodelling hierarchy

and create an intuitive alignment as shown in Fig. 4.3. However, modelling formalisms may be used to represent modelling languages at any level of the hierarchy, as demonstrated in Fig. 4.4.

In a modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$, predicates from the signature Σ_3 are used to add atomic constraints to the metamodel $\mathfrak{S}_2$. This corresponds to metamodel definition. These constraints should be satisfied by all specifications $((S_1, C^{\mathfrak{S}_1} \colon \Sigma_2) \blacktriangleright_{\iota_{S_1}} \mathfrak{S}_2)$. With regard to the classification of constraints in Section 1.5, these atomic constraints may represent both structural constraints SC_3 and additional constraints AC_3.

Moreover, predicates from the signature $\Sigma_2 \rhd S_2$ are used to add constraints to typed and conformant specifications; i.e. $((S_1, C^{\mathfrak{S}_1} \colon \Sigma_2) \rhd_{\iota_{S_1}} S_2)$ and $((S_1, C^{\mathfrak{S}_1} \colon \Sigma_2) \blacktriangleright_{\iota_{S_1}} \mathfrak{S}_2)$, respectively. This corresponds to model definition. These constraints should be satisfied by instances of these specifications. With regard to the classification of constraints in Section 1.5, these atomic constraints may represent both structural constraints SC_2 and additional constraints AC_2.

Any of the signatures in a modelling formalism may be empty. However, if both signatures are empty; that is $(\emptyset, S_2, \emptyset)$, the modelling formalism will only be able to specify the underlying graphs of specifications, without constraints added to them. This resembles the graph-based formalisation of the metamodelling hierarchy as discussed in Section 2.2.

In theory, a metamodelling hierarchy may have an infinite number of levels [87]. However, in practise, this hierarchy has a natural end point. The model at the top level is usually a reflexive metamodel. The reflexive metamodel corresponds to a reflexive modelling language able to define its own metamodel. Reflexive metamodel and reflexive modelling formalism are defined as follows (see Fig. 4.5).

REFLEXIVE METAMODEL

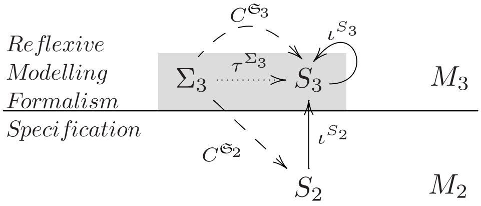


Figure 4.5: A reflexive modelling formalism $(\Sigma_3, \mathfrak{S}_3, \Sigma_3)$ together with a specification $((S_2, C^{\mathfrak{S}_2}\colon \Sigma_3) \blacktriangleright_{\iota^{S_2}} \mathfrak{S}_3)$

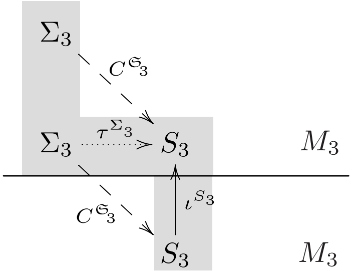


Figure 4.6: A reflexive modelling formalism $(\Sigma_3, \mathfrak{S}_3, \Sigma_3)$ specifying its own metamodel $((S_3, C^{\mathfrak{S}_3}\colon \Sigma_3) \blacktriangleright_{\iota^{S_3}} \mathfrak{S}_3)$

Definition 26 (Reflexive Metamodel) *A specification $\mathfrak{S}$ is reflexive if there exists a graph homomorphism $\iota^S : S \to S$ such that $\mathfrak{S} = ((S, C^{\mathfrak{S}}\colon \Sigma) \blacktriangleright_{\iota^S} \mathfrak{S})$; i.e. such that $\mathfrak{S}$ conforms to itself.*

Definition 27 (Reflexive Modelling Formalism) *A reflexive modelling formalism $(\Sigma_3, \mathfrak{S}_3, \Sigma_3)$ is given by a signature $\Sigma_3 = ((P^{\Sigma_3}, \alpha^{\Sigma_3}) \triangleright_{\tau^{\Sigma_3}} S_3)$ and a reflexive metamodel $\mathfrak{S}_3 = ((S_3, C^{\mathfrak{S}_3}\colon \Sigma_3) \blacktriangleright_{\iota^{S_3}} \mathfrak{S}_3)$.*

For a reflexive modelling formalism $(\Sigma_3, \mathfrak{S}_3, \Sigma_3)$, the same signature Σ_3 may be used both to add constraints to the reflexive metamodel $\mathfrak{S}_3$ and to add constraints to specifications $\mathfrak{S}_2$ specified by $(\Sigma_3, \mathfrak{S}_3, \Sigma_3)$. In fact, Σ_3 may appear in two forms to play these two roles: as a typed signature $\Sigma_3 \triangleright S_3$ and as an untyped signature Σ_3 (see Fig. 4.6).

As mentioned in the introduction, we use a running example to illustrate our approach to model transformation. The example presents transformation of a structural object-oriented model to a relational data model. In the following, we present the two modelling formalisms which are used to specify these kinds of models.

4.3.1 Object-Oriented Modelling Formalism

Building upon Examples 8 and 10, a modelling language for specifying structural object-oriented models is represented by a modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ as shown in the shaded area of Fig. 4.7. Details of an untyped version of the signature Σ_2 are shown in Table 3.1, while parts of its typed version are shown in Fig 4.1a. The types chosen for the signature in Fig. 4.1b correspond to the (first letters of the) types in the metamodel $\mathfrak{S}_2$ in Fig. 4.7b; i.e. (C)lass, (R)eference,(A)ttribute and (D)ataType.

The signature Σ_3 in Fig. 4.7 is not used and hence left empty since there is no need to add atomic constraints to the metamodel $\mathfrak{S}_2$. The metamodel $\mathfrak{S}_2$ declares some basic concepts in object-oriented modelling; such as Class, Reference, Attribute and DataType. We have chosen this subset of concepts based on the basic concepts which are chosen for explaining Ecore in [115]. A sample specification $\mathfrak{S}_1 = ((S_1, C^{\mathfrak{S}_1}\colon \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$ is also shown in Fig. 4.7c in order to demonstrate the usage of the predicates of Σ_2 and the typing morphism $\iota^{S_1} : S_1 \to S_2$. This specification was also shown in Fig. 3.1a.

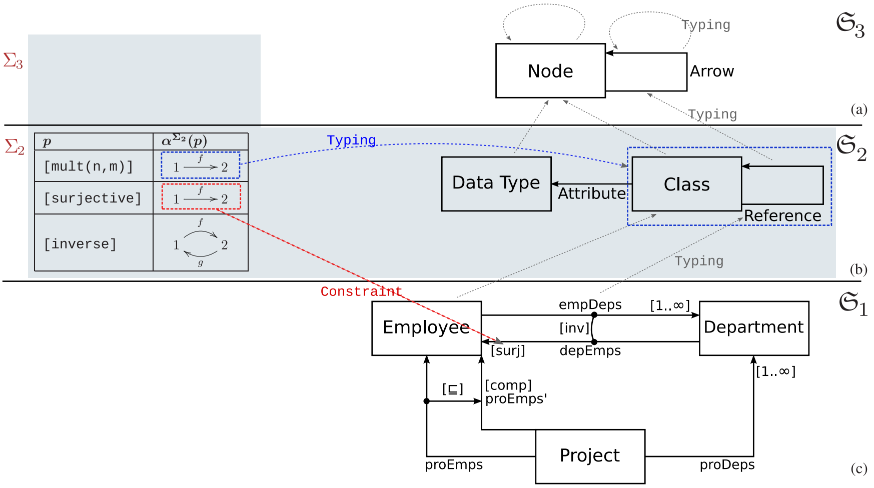
Remark 16 (Typing Notation) *We write* $\boxed{\textsf{Employee:Class}}$ *to denote the typing map* $\iota^{S_1}(Employee) = Class$. *As seen from* $\mathfrak{S}_1$ *in Fig. 4.7c, we may also write* $\boxed{\textsf{Employee}}$ *if the typing is obvious from the context. With regard to arrows, we use* empDeps:Ref *to denote* $\iota^{S_1}(empDeps) = Reference$. *Likewise, we may also write* empDeps *if the typing is obvious from the context.*

For the sake of completeness, the reflexive modelling formalism $(\Sigma_3, \mathfrak{S}_3, \Sigma_3)$ is also shown in the figure. Later, we will compare this modelling formalism with the modelling formalism used for specifying the metamodel of relational data models.

Semantics of nodes and arrows

Note that since $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ is used for specification of structural object-oriented models, the nodes and arrows in the arities of predicates in $\Sigma_2 \rhd S_2$ are interpreted as sets and multi-valued functions, respectively. For the same reason, the nodes and arrows in the underlying graphs of specifications which are specified by $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ are interpreted as sets and multi-valued functions, respectively.

Remark 17 (Modelling Formalism vs Graph-based Formalisation) *We can compare the stack of modelling formalisms in Fig. 4.7 to the metamodelling hierarchy of OMG in Fig. 1.6 and the graph-based formalisation of the hierarchy shown in Fig. 2.1. Fig. 4.7a shows a specification* $\mathfrak{S}_3$ *representing a generic meta-metamodel. Fig. 4.7b shows a specification* $\mathfrak{S}_2$ *representing a metamodel for the specification of structural object-oriented models. Fig. 4.7c shows a sample specification* $\mathfrak{S}_1 = (S_1, C^{\mathfrak{S}_1}\colon \Sigma_2)$ *which represents the same structural object-oriented model shown in Example 10. In these specifications, nodes and arrows are interpreted as sets and multi-valued functions. Moreover, the underlying graphs* S_3*,* S_2 *and* S_1 *of these specifications are exactly the same as the graphs* G_3*,* G_2 *and* G_1



p	$\alpha^{\Sigma_2}(p)$
[mult(n,m)]	$1 \xrightarrow{f} 2$
[surjective]	$1 \xrightarrow{f} 2$
[inverse]	$1 \underset{g}{\overset{f}{\rightleftarrows}} 2$

Figure 4.7: A modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ together with the reflexive modelling formalism $(\Sigma_3, \mathfrak{S}_3, \Sigma_3)$ and a sample specification $((S_1, C^{\mathfrak{S}_1} \colon \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$; not all constraints and typing morphisms are explicitly shown

in Fig. 2.1, respectively. Furthermore, for the lowest level M_0 in Fig. 2.1, Fig. 3.2 showed a graph I representing an instance of the model $\mathfrak{S}_1$.

Similar to the graph-based formalisation of the metamodelling hierarchy in Example 8, $\mathfrak{S}_3$ is typed by itself (reflexive), $\mathfrak{S}_2$ is typed by $\mathfrak{S}_3$, $\mathfrak{S}_1$ is typed by $\mathfrak{S}_2$ and I is typed by $\mathfrak{S}_1$. In addition to the typing relations between the underlying graphs of the specifications, constraints added at any level is required to be satisfied by the underlying graphs of specifications at the level below.

Hierarchical Relations

Most object-oriented modelling languages contain a special construct to express hierarchical relations of the kind *inheritance* [118]. This relation is used to provide a hierarchy in object-oriented models and is especially useful considering reuse and structuring. It is inspired by the inheritance feature of object-oriented programming languages. This feature is all about factoring out the commonality of a collection of similar types and putting them in a new type; then members of the collection inherit these common parts from the new type [110]. In UML, this relation is called *generalisation* [91], while in EMF it is called *eSuperType* [115]. In this thesis, inheritance relations are represented by *inheritance arrows* and the way we define and visualise these arrows is inspired by EMF.

When a class X' inherits from a class X, X' gets access to all attributes and references from X by composing the inheritance arrow with the attributes and references arrows, respectively (see Fig. 4.8). In this relation, the class X is called

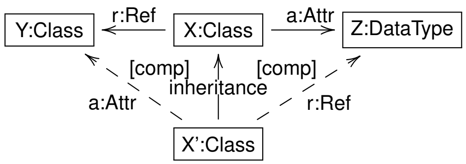


Figure 4.8: The subclass X' inherits all attributes and references from the superclass X

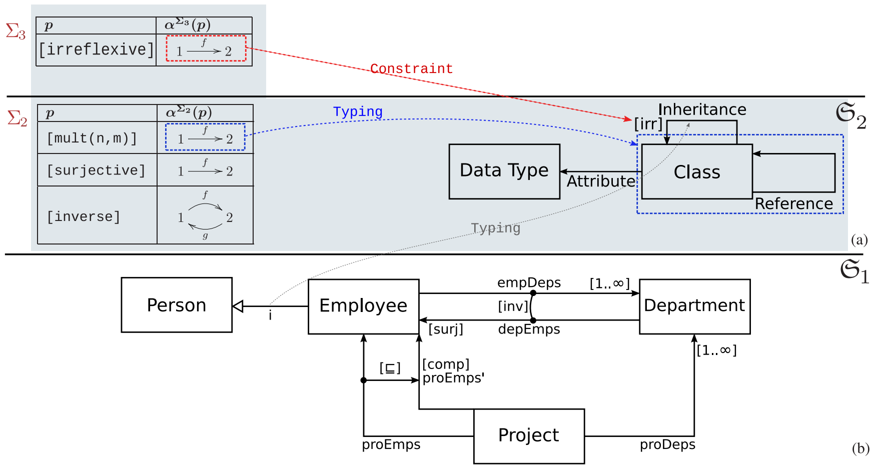


Figure 4.9: The modified modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ from Fig. 4.7; the metamodel $\mathfrak{S}_2$ is extended with Inheritance arrow type

the superclass while the class X' is called the subclass. Moreover, the inheritance relation is transitive; that is, if a class X'' inherits from X', then X'' inherits also from X. Furthermore, the inheritance relation can not be circular; that is, the class X can not inherit from itself or its subclasses.

In order to express inheritance in the object-oriented modelling formalism in Fig. 4.7, we extend the metamodel $\mathfrak{S}_2$ with an arrow type Inheritance from Class to Class, as shown in Fig 4.9a. In addition, we add the predicate `[irreflexive]`, shown originally in Table 3.1, to the signature Σ_3. This predicate is used to add the constraint (`[irreflexive]`, δ) to the arrow Inheritance. This constraint forbids circularity of arrows which are typed by Inheritance.

Inheritance arrows $X' \xrightarrow{i} X$ will be visualised as $\boxed{\text{X':Class}} \xrightarrow{\text{i}} \boxed{\text{X:Class}}$ (see Fig 4.9b).

Example 16 (Inheritance) For the modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ in Fig 4.9a, Fig 4.9b shows a modified version of the typed specification $((S_1, C^{\mathfrak{S}_1}:$

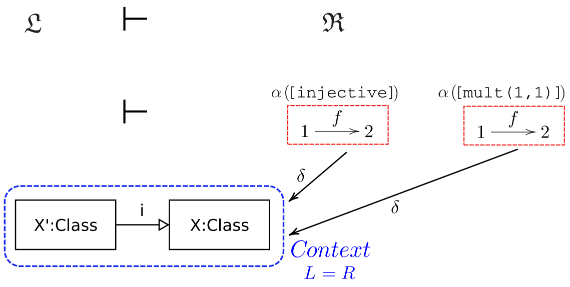


Figure 4.10: A typed specification entailment $\mathfrak{L} \rhd S_2 \vdash \mathfrak{R} \rhd S_2$ related to inheritance arrows

$\Sigma_2) \rhd_{\iota S_1} S_2)$ in Fig. 4.7c with an inheritance arrow $Employee \xrightarrow{i} Person$. ◇

Remark 18 (Inheritance and Typed Specification Entailment) *From an inheritance arrow we can entail that the arrow is injective, total and single-valued*[1]*. That is, for the modelling formalism in Fig. 4.9a, we have the following typed specification entailment $\mathfrak{L} \rhd S_2 \vdash \mathfrak{R} \rhd S_2$, shown in Fig. 4.10:*

$$\begin{aligned} C^{\mathfrak{L}} &= \emptyset \\ C^{\mathfrak{R}} &= \{(\texttt{[injective]}, \delta), \\ &\quad (\texttt{[mult(1,1)]}, \delta)\} \\ L = R &= \boxed{X'{:}Class} \xrightarrow{i} \boxed{X{:}Class} \end{aligned}$$

That is, we have the typing maps:

$$\begin{aligned} \iota^L(X) = \quad & \iota^L(X') = \quad \iota^R(X) \\ = \quad & \iota^R(X') = \quad \mathit{Class} \\ \iota^L(c) = \quad & \iota^R(c) = \quad \mathit{Inheritance} \end{aligned}$$

Containment Relation

In object-oriented structural models, it is sometimes desirable to specify that the existence of instances of a class are dependent on instances of other classes. This is useful when modelling the life cycle of the instances. In UML, this relation is called *composition* [91], while in EMF it is called *containment* [115]. In this thesis, containment relations are represented by *containment arrows* and the way we define and visualise these arrows is inspired by EMF.

[1] Note that according to the semantic interpretation $[\![..]\!]^{\Sigma_2}$ of Σ_2 from Table 3.1, if an arrow is injective and single-valued, it is also non-overlapping.

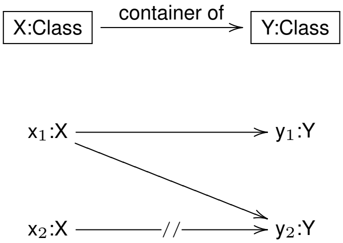
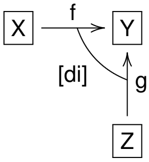


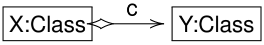
Figure 4.11: Instances of the class Y can be contained in only one instance of X

Table 4.1: Additions to the signature Σ_2 to enable definition of properties related to containment arrows

p	$\alpha^{\Sigma_2}(p)$	**Proposed vis.**	**Semantic interpret.**
`[disjoint-image]`	$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \\ & 3 \arrow[u, "g"'] \end{tikzcd}$		$f(X) \cap g(Z) = \emptyset$

When a class X is said to have a containment relation to a class Y, instances of Y do only exist if they are related to some instances of X. In this relation, the class X is called the *container* while the class Y is called the *contained* class. Moreover, an instance of Y cannot be related to more than one instance of X (see Fig. 4.11). Although it is possible to to have two or more containment relations from one or more classes to the same class Y, an instance of Y is allowed to be related to only one of the instances of the container classes.

In order to express this kind of containment in the object-oriented modelling formalism in Fig. 4.9, we extend the metamodel $\mathfrak{S}_2$ with an arrow type Containment from Class to Class, as shown in Fig 4.12a. In addition, we add the predicate `[disjoint-image]`, shown in Table 4.1, to the signature Σ_2. The corresponding constraint of this predicate is used to forbid that an insatnce of a class has two containers.

Containment arrows $X \overset{c}{\to} Y$ will be visualised as .

Example 17 (Containment) For the modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ in Fig 4.12a, Fig 4.12b shows a modified version of the typed specification $((S_1, C^{\mathfrak{S}_1}\colon \Sigma_2) \rhd_{\iota S_1} S_2)$ in Fig. 4.9c with a containment arrow $\mathit{Faculty} \overset{c}{\to} \mathit{Department}$. $\diamondsuit$

Remark 19 (Containment and Typed Specification Entailment) *From a containment arrow we can entail that the arrow is non-overlapping. If there is only one*

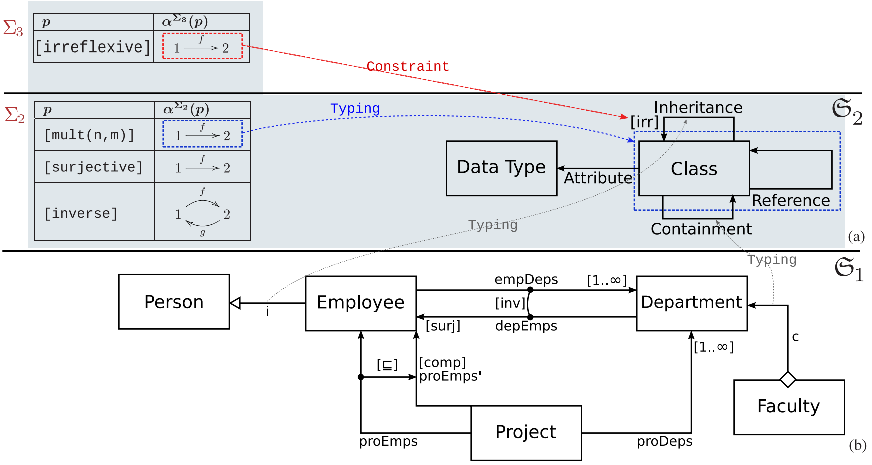


Figure 4.12: The modified modelling formalism $(\Sigma_2 \triangleright S_2, \mathfrak{S}_2, \Sigma_3)$ from Fig. 4.9; the metamodel $\mathfrak{S}_2$ is extended with Containment arrow type

containment arrow into a class, we can entail that the containment arrow is also surjective. Moreover, if there are exactly two containment arrows into the same class, we can entail that the arrows together are jointly-surjective and have disjoint images. The latter ensures that each instance of the target class has exactly one container. That is, for the modelling formalism in Fig. 4.12a, we have the following typed specification entailments, shown in Fig. 4.13, Fig. 4.14 and 4.15, respectively:

1. $\mathfrak{L} \triangleright S_2 \vdash_1 \mathfrak{R} \triangleright S_2$ *with:*

$$\begin{aligned} C^{\mathfrak{L}} &= \emptyset \\ C^{\mathfrak{R}} &= \{(\texttt{[non-overlapping]}, \delta)\} \\ L = R &= \boxed{X{:}Class} \diamond\!\!\xrightarrow{c} \boxed{Y{:}Class} \end{aligned}$$

That is, we have the typing maps:

$$\begin{aligned} \iota^L(X) = \quad \iota^L(Y) &= \quad \iota^R(X) \\ = \quad \iota^R(Y) &= \quad \textit{Class} \\ \iota^L(c) = \quad \iota^R(c) &= \quad \textit{Containment} \end{aligned}$$

2. $\mathfrak{L} \triangleright S_2 \vdash_2 \mathfrak{R} \triangleright S_2$ *with:*

$$\begin{aligned} C^{\mathfrak{L}} &= \emptyset \\ C^{\mathfrak{R}} &= \{(\texttt{[surjective]}, \delta)\} \\ L = R &= \boxed{X{:}Class} \diamond\!\!\xrightarrow{c} \boxed{Y{:}Class} \end{aligned}$$

That is, we have the typing maps:

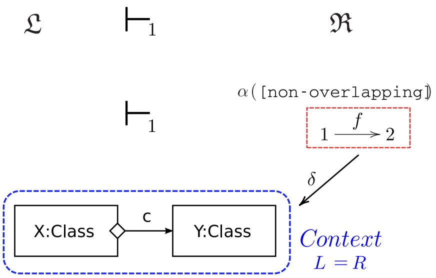


Figure 4.13: A typed specification entailment $\mathfrak{L} \triangleright S_2 \vdash_1 \mathfrak{R} \triangleright S_2$ related to every arrow typed by Containment

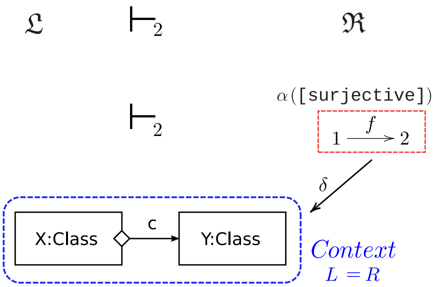


Figure 4.14: A typed specification entailment $\mathfrak{L} \triangleright S_2 \vdash_2 \mathfrak{R} \triangleright S_2$ which applies only for the case of exactly one arrow typed by Containment

$$\begin{array}{rrl} \iota^L(X) = & \iota^L(Y) = & \iota^R(X) \\ = & \iota^R(Y) = & \textsf{Class} \\ \iota^L(c) = & \iota^R(c) = & \textsf{Containment} \end{array}$$

3. *$\mathfrak{L} \triangleright S_2 \vdash_3 \mathfrak{R} \triangleright S_2$ with*

$$\begin{array}{rl} C^{\mathfrak{L}} = & \emptyset \\ C^{\mathfrak{R}} = & \{(\texttt{[disjoint-image]}, \delta), \\ & (\texttt{[jointly-surjective\_2]}, \delta)\} \end{array}$$

$$L = R = \boxed{\textsf{X:Class}} \diamond\!\!\xrightarrow{c_1} \boxed{\textsf{Y:Class}} \xleftarrow{c_2}\!\!\diamond \boxed{\textsf{Z:Class}}$$

That is, we have the typing maps:

$$\begin{array}{rrrrl} \iota^L(X) = & \iota^L(Y) = & \iota^L(Z) = & \iota^R(X) = & \\ & & \iota^R(Y) = & \iota^R(Z) = & \textsf{Class} \\ \iota^L(c_1) = & \iota^L(c_2) = & \iota^R(c_1) = & \iota^R(c_2) = & \textsf{Containment} \end{array}$$

Note that if we have n containment arrows with the same target class, we can entail that each two of these arrows have disjoint-images. In addition, we can entail `jointly-surjective_n` on the n arrows together.

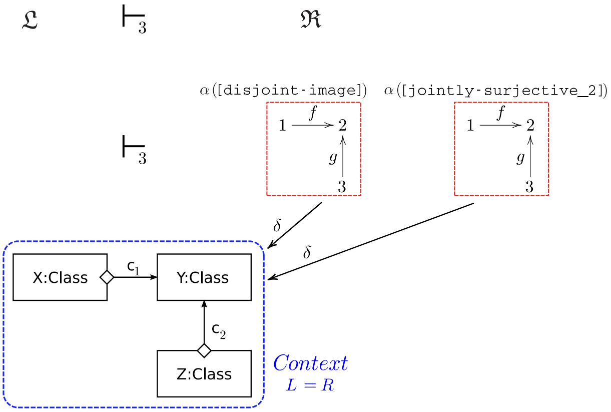


Figure 4.15: A typed specification entailment $\mathfrak{L}\rhd S_2 \vdash_3 \mathfrak{R}\rhd S_2$ which applies only in the case of exactly two arrows typed by Containment

4.3.2 RELATIONAL MODELLING FORMALISM

In DPF, a modelling language for specifying relational data models is represented by a modelling formalism $(\Theta_2\rhd T_2, \mathfrak{T}_2, \Theta_3)$ as shown in the shaded area of Fig. 4.16. The signature $\Theta_2\rhd T_2$ is shown in 4.2, while the signature Θ_3 consists of the predicate [mult(n,m)]. The predicates in Table 4.2 are generalisations, or general patterns, for constraints that are used in relational data modelling.

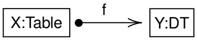
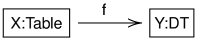
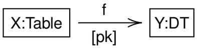
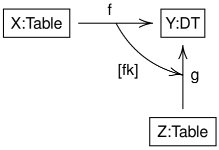
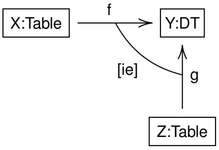
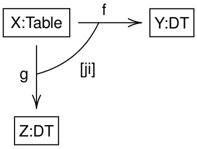
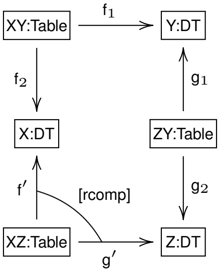
The metamodel $\mathfrak{T}_2$ declares some basic concepts in relational data modelling; such as Table, Column and DataType. A sample specification $\mathfrak{T}_1 = ((T_1, C^{\mathfrak{T}_1} : \Theta_2) \blacktriangleright_{\iota^{T_1}} \mathfrak{T}_2)$ is also shown in Fig. 4.16c in order to demonstrate the usage of the predicates of $\Theta_2 \rhd T_2$ and the typing morphism $\iota^{T_1} : T_1 \rightarrow T_2$. It will be clear later that this specification corresponds to the object-oriented specification which was shown in Fig. 3.1a.

SEMANTICS OF NODES AND ARROWS

Note that since $(\Theta_2 \rhd T_2, \mathfrak{T}_2, \Theta_3)$ is used for specification of relational data models, the nodes and arrows in the arities of predicates in Θ_2 are interpreted as sets and (single-valued) functions, respectively. For the same reason, the nodes and arrows in the underlying graph of specifications which are specified by $(\Theta_2\rhd T_2, \mathfrak{T}_2, \Theta_3)$ are interpreted as sets and (single-valued) functions, respectively.

The reflexive modelling formalism $(\Theta_3, \mathfrak{T}_3, \Theta_3)$ is shown on top of Fig. 4.16. It is noticeable that this modelling formalism is similar to the one used to specify the metamodel $\mathfrak{S}_2$. In particular, the metamodels $\mathfrak{S}_3$ and $\mathfrak{T}_3$ are identical. This follows the same strategy of OMG in which the top most modelling language in the hierarchy – MOF – is unified in order to facilitate exchange and transformation of models.

Table 4.2: The signature $\Theta_2 \triangleright T_2$

$\boldsymbol{p}$	$\boldsymbol{\alpha^{\Theta_2}(p)}$	**Proposed vis. and typing**	**Semantic interpret.**
`[total]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$$		$\forall x \in X : f(x)$ is defined
`[injective]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$$		$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[primary-key]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$$		f is `[total]` and `[injective]`
`[foreign-key]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \\ & 3 \arrow[u, "g"'] \end{tikzcd}$$		$f(X) \subseteq g(Z)$ and g is `[primary-key]`
`[image-equal]`	$$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \\ & 3 \arrow[u, "g"'] \end{tikzcd}$$		$f(X) = g(Z)$
`[jointly-injective]`	$$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[d, "g"'] & 2 \\ 3 & \end{tikzcd}$$		$\forall x, x' \in X : f(x) = f(x')$ and $g(x) = g(x')$ implies $x = x'$
`[rcomp]`	$$\begin{tikzcd} 1 \arrow[r, "f_1"] \arrow[d, "f_2"'] & 2 \\ 3 & 4 \arrow[u, "g_1"'] \arrow[d, "g_2"] \\ 5 \arrow[u, "f'"] \arrow[r, "g'"'] & 6 \end{tikzcd}$$	 XY:Table f₁ Y:DT f₂ g₁ X:DT ZY:Table f′ [rcomp] g₂ XZ:Table Z:DT g′	$XZ = \{(x, z) \mid \exists y \in Y : (x, y) \in XY \wedge (z, y) \in ZY\}$

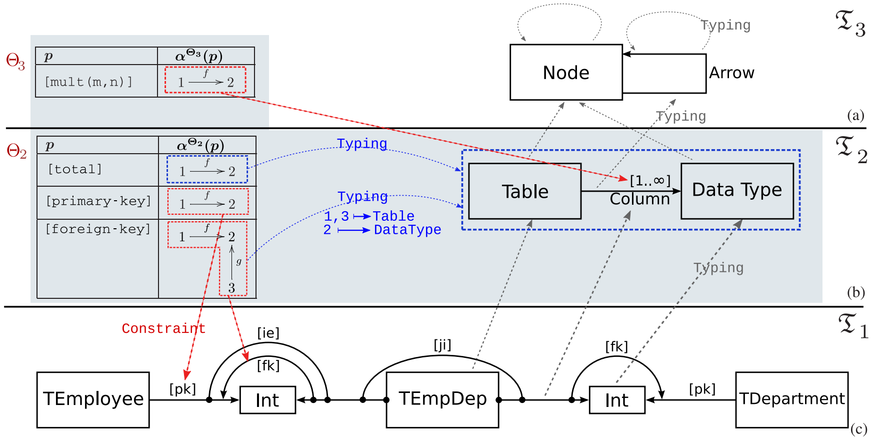


Figure 4.16: A modelling formalism $(\Theta_2 \triangleright T_2, \mathfrak{T}_2, \Theta_3)$ together with the reflexive modelling formalism $(\Theta_3, \mathfrak{T}_3, \Theta_3)$ and a sample specification $((T_1, C^{\mathfrak{T}_1}:\Theta_2) \blacktriangleright_{\iota^{T_1}} \mathfrak{T}_2)$; not all constraints and typing morphisms are explicitly shown

4.4 *Universal Constraints*

So far we have discussed two concepts for constraining specifications: existence of a typing morphism to the metamodel of the modelling formalism and satisfaction of the atomic constraints which are added to the metamodel. These concepts are used to define the conformance relation between specifications and the metamodel of the modelling formalism. In addition to the conformance requirement, there are other constraints concerning the overall structure of specifications defined by a modelling formalism. An example is if one wants to formulate that in EMF models "every model must have a root class" and "every class in a model must have the root class as its container, directly or transitively". We call constraints which have an overall impact on specifications for *universal constraints*. As the name "universal constraint" suggests, it is universally quantified over elements (nodes, arrows and constraints) of a specification. In addition, each universal constraint should hold for all specifications which are specified by the modelling formalism.

Universal constraints can be defined both for (untyped) specifications, typed specifications and conformant specifications. However, in the following we define these constraints for the case of typed specifications since according to Definitions 22 and 23 the definitions can be applied to untyped and conformant specifications also.

Definition 28 (Universal Constraint) *Given a signature $\Sigma \triangleright G = ((P^\Sigma, \alpha^\Sigma) \triangleright_{\tau^\Sigma} G)$, a universal constraint is a typed specification morphism $c : \mathfrak{L} \triangleright G \to \mathfrak{R} \triangleright G$ with $\mathfrak{L} \triangleright G = ((L, C^{\mathfrak{L}}:\Sigma) \triangleright_{\iota^L} G)$ and $\mathfrak{R} \triangleright G = ((R, C^{\mathfrak{R}}:\Sigma) \triangleright_{\iota^R} G)$.*

The universal constraints related to a modelling formalism $(\Sigma_2 \triangleright S_2, \mathfrak{S}_2, \Sigma_3)$ explicate requirements which have to be satisfied by all specifications $\mathfrak{S}_1 \triangleright S_2$ and $\mathfrak{S}_1 \blacktriangleright \mathfrak{S}_2$; i.e. all specifications that can be specified by the modelling formalism. The satisfaction of a universal constraint by a specification is defined as follows.

Definition 29 (Satisfaction of Universal Constraints) *A typed specification $\mathfrak{S} \triangleright G = ((S, C^{\mathfrak{S}} : \Sigma) \triangleright_{\iota^S} G)$ satisfies a universal constraint $c : \mathfrak{L} \triangleright G \to \mathfrak{R} \triangleright G$ iff for any typed specification morphism $m : \mathfrak{L} \triangleright G \to \mathfrak{S} \triangleright G$ there is a typed specification morphism $n : \mathfrak{R} \triangleright G \to \mathfrak{S} \triangleright G$ such that $c; n = m$.*

$$\begin{tikzcd}
 & G & \\
 & S \arrow[u, "\iota^S"'] \arrow[d, phantom, "="] & \\
L \arrow[ur, "m"] \arrow[uur, bend left, "\iota^L"] \arrow[rr, "c"] & & R \arrow[ul, "n"'] \arrow[uul, bend right, "\iota^R"']
\end{tikzcd}$$

We call $\mathfrak{L} \triangleright G$ and $\mathfrak{R} \triangleright G$ input and output patterns of the constraint c, respectively; and we call m and n matches of the patterns $\mathfrak{L} \triangleright G$ and $\mathfrak{R} \triangleright G$ in $\mathfrak{S} \triangleright G$, respectively.

Some examples of universal constraints are presented Sections 4.4.1 and 4.4.2.

Remark 20 (Universal Constraints and Transformation Rules) *Each universal constraint $c : \mathfrak{L} \triangleright G \to \mathfrak{R} \triangleright G$ may be understood as a transformation rule (see Definition 36) in the sense that, given a typed specification $\mathfrak{S} \triangleright G = ((S, C^{\mathfrak{S}} : \Sigma) \triangleright_{\iota^S} G)$, if there is a match $m : \mathfrak{L} \triangleright G \to \mathfrak{S} \triangleright G$, then a match $n : \mathfrak{R} \triangleright G \to \mathfrak{S} \triangleright G$ will be created such that $c; n = m$. This can be used to derive information which is left implicit in specifications.*

Remark 21 (Universal Constraints vs Graph Constraints) *The concept of universal constraints is a generalisation of the concept of graph constraints which is detailed in [41; 118]. It is a generalisation in the sense that a universal constraint $c : \mathfrak{L} \to \mathfrak{R}$ with empty sets $C^{\mathfrak{L}}$ and $C^{\mathfrak{R}}$ can be seen as a (typed) graph constraint [41]. In [77], a variant of universal constraints, which is without typing, is called "sketch axioms" or "sketch entailment". In this variant, a full hierarchy is proposed for these axioms with graph constraints on the lowest level.*

The universal constraints we have discussed so far have a positive nature; that is, they require the existence of some property. However, in some cases it is easier to think of and to define negative universal constraints; for example, in relational data models "every table must have exactly one primary key column". This sort of constraint can be expressed by negative universal constraints (NUC).

Definition 30 (Negative Universal Constraints) *Given a typed signature $\Sigma \triangleright G = ((P^{\Sigma}, \alpha^{\Sigma}) \triangleright_{\tau^{\Sigma}} G)$, a negative universal constraint is a typed specification morphism $n : \mathfrak{L} \triangleright G \rightarrow \mathfrak{N} \triangleright G$ with $\mathfrak{L} \triangleright G = ((L, C^{\mathfrak{L}} : \Sigma) \triangleright_{\iota^L} G)$ and $\mathfrak{N} \triangleright G = ((N, C^{\mathfrak{N}} : \Sigma) \triangleright_{\iota^N} G)$.*

Analogous to (positive) universal constraints, the satisfaction of a NUC by a specification is defined as follows.

Definition 31 (Satisfaction of Negative Universal Constraints) *A typed specification $\mathfrak{S} \triangleright G = ((S, C^{\mathfrak{S}} : \Sigma) \triangleright_{\iota^S} G)$ satisfies a negative universal constraint $n : \mathfrak{L} \triangleright G \rightarrow \mathfrak{N} \triangleright G$ iff for any injective typed specification morphism $m : \mathfrak{L} \triangleright G \rightarrow \mathfrak{S} \triangleright G$ there does not exist an injective typed specification morphism $q : \mathfrak{N} \triangleright G \rightarrow \mathfrak{S} \triangleright G$ such that $n; q = m$.*

$$\begin{tikzcd}
 & G & \\
 & S \arrow[u, "\iota^S"'] \arrow[d, phantom, "\neq"] & \\
L \arrow[uur, bend left, "\iota^L"] \arrow[ur, "m"] \arrow[rr, "n"] & & N \arrow[uul, bend right, "\iota^N"'] \arrow[ul, "q"']
\end{tikzcd}$$

Next, we show some examples of universal constraints for the two modelling formalisms which we defined in Sections 4.3.1 and 4.3.2.

4.4.1 UNIVERSAL CONSTRAINTS FOR OBJECT-ORIENTED MODELLING FORMALISM

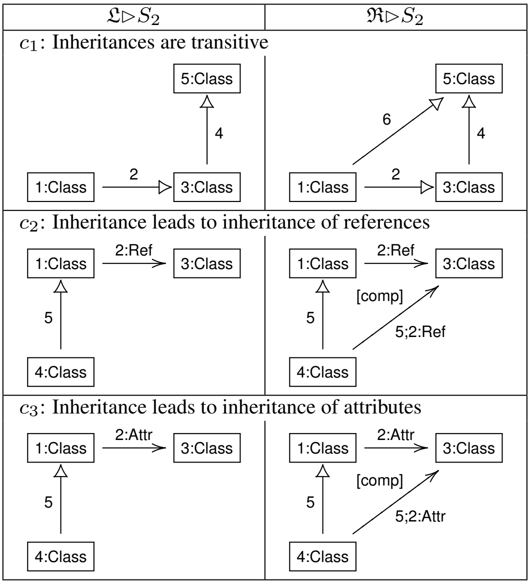
Universal constraints related to object-oriented structural modelling are mostly related to requirements that hierarchical and containment relations should satisfy. In order to express the properties of hierarchical and containment relations, we add the universal constraints shown in Table 4.3 to the modelling formalism $(\Sigma_2 \triangleright S_2, \mathfrak{S}_2, \Sigma_3)$ from Section 4.3.1. These constraints are defined as typed specification morphisms $c : \mathfrak{L} \triangleright S_2 \rightarrow \mathfrak{R} \triangleright S_2$ with $\mathfrak{L} \triangleright S_2 = ((L, C^{\mathfrak{L}} : \Sigma_2) \triangleright_{\iota^L} S_2)$ and $\mathfrak{R} \triangleright S_2 = ((R, C^{\mathfrak{R}} : \Sigma_2) \triangleright_{\iota^R} S_2)$.

The constraint c_1 in Table 4.3 describe that inheritance arrows are transitive. Combining c_1 with the irreflexivity constraint on the arrow type Inheritance in Fig. 4.9 will ensure that specifications do not contain circular inheritance relations. The universal constraints c_2, c_3 express the property of inheritance arrows which we mentioned in Section 4.3.1. That is, a class 4:Class which inherits from another class 1:Class will have access to all of the references and attributes of 1:Class.

4.4.2 UNIVERSAL CONSTRAINTS FOR RELATIONAL MODELLING FORMALISM

For the modelling formalism $(\Theta_2 \triangleright T_2, \mathfrak{T}_2, \Theta_3)$ in Section 4.3.2, some of the universal constraints which have to be satisfied by any specification $\mathfrak{T}_1$ in $\textbf{TSpec}(\Theta_2 \triangleright T_2)$

Table 4.3: Some universal constraints for object-oriented structural models


$\mathfrak{L}\triangleright S_2$
$\mathfrak{R}\triangleright S_2$
c_1: Inheritances are transitive
5:Class
4
1:Class
2
3:Class
5:Class
6
4
1:Class
2
3:Class
c_2: Inheritance leads to inheritance of references
1:Class
2:Ref
3:Class
5
4:Class
1:Class
2:Ref
3:Class
[comp]
5
5;2:Ref
4:Class
c_3: Inheritance leads to inheritance of attributes
1:Class
2:Attr
3:Class
5
4:Class
1:Class
2:Attr
3:Class
[comp]
5
5;2:Attr
4:Class

are shown in Table 4.4. These constraints are defined as typed specification morphisms $c : \mathfrak{L} \triangleright T_2 \to \mathfrak{R} \triangleright T_2$ with $\mathfrak{L} \triangleright T_2 = ((L, C^{\mathfrak{L}} : \Theta_2) \triangleright_{\iota^L} T_2)$ and $\mathfrak{R} \triangleright T_2 = ((R, C^{\mathfrak{R}} : \Theta_2) \triangleright_{\iota^R} T_2)$. Note that we only show mappings for nodes and arrows whose names do not match; the others are omitted.

There are some differences between the universal constraints in Table 4.4. While the constraints c_1 and c_3 require the existence of some structures in $\mathfrak{T}_1$, the other two constraints prohibit some structures. The constraint c_2 is a uniqueness condition. It ensures that each table has at most one primary key column. If a table would have two different primary key columns we can find a match m of the input pattern with $m(2) \neq m(4)$. But then we can not find a match n of the output pattern such that $m = c_2; n$. The constraint c_4 works analogously. Since the constraints c_2 and c_4 have a negative nature, we can also define them as the NUCs n_1 and n_2 in Table 4.5, respectively.

4.4.3 Specification Entailment as Universal Constraint

Specification entailments, especially the typed version (see Definitions 15 and 24), are closely related to universal constraints (see Fig.4.17). Each typed specification entailment $\mathfrak{L} \triangleright G \vdash \mathfrak{R} \triangleright G$ gives rise to a corresponding universal constraint $c : \mathfrak{L} \triangleright G \to \mathfrak{R} \triangleright G$ with an identity graph homomorphism $id_L : L \to R$. Hence, the underlying graphs L and R are identical, while the atomic constraints $C^{\mathfrak{R}} = C^{\mathfrak{L}} \cup C^{\mathfrak{R}}$.

Table 4.4: Some universal constraints for relational data models

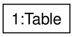
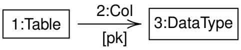
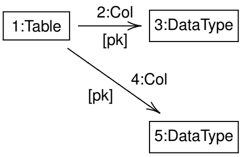
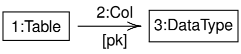
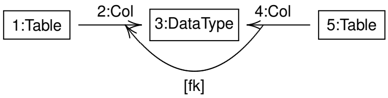
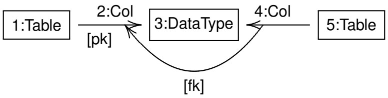
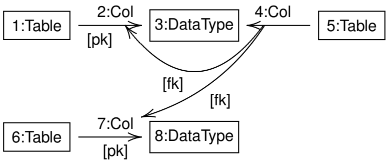
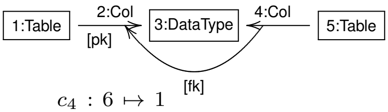
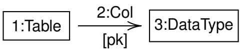
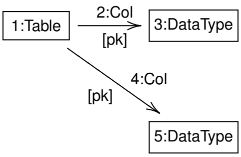
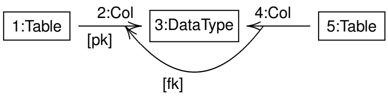
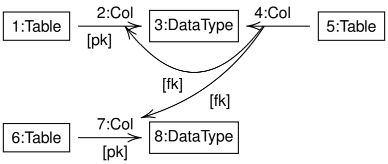
$\mathfrak{L}\triangleright T_2$	$\mathfrak{R}\triangleright T_2$
c_1: Every table must have a primary key column	
	
c_2: Every table must have exactly one primary key column	
 1:Table 2:Col [pk] 3:DataType 4:Col [pk] 5:DataType	 $c_2\ :\ 2,4\ \mapsto\ 2$ $c_2\ :\ 3,5\ \mapsto\ 3$
c_3: A foreign key column should only refer to a primary key column	
	 1:Table 2:Col [pk] 3:DataType 4:Col 5:Table [fk]
c_4: A foreign key column should only refer to exactly one primary key column	
 1:Table 2:Col [pk] 3:DataType 4:Col 5:Table [fk] [fk] 6:Table 7:Col [pk] 8:DataType	 1:Table 2:Col [pk] 3:DataType 4:Col 5:Table [fk] $c_4\ :\ 6\ \mapsto\ 1$ $c_4\ :\ 7\ \mapsto\ 2$ $c_4\ :\ 8\ \mapsto\ 3$

Table 4.5: Some negative universal constraints for relational data models

$\mathfrak{L}\triangleright T_2$	$\mathfrak{N}\triangleright T_2$
n_1: Every table must have exactly one primary key column	
	 1:Table 2:Col [pk] 3:DataType 4:Col [pk] 5:DataType
n_2: A foreign key column should only refer to exactly one primary key column	
 1:Table 2:Col [pk] 3:DataType 4:Col 5:Table [fk]	 1:Table 2:Col [pk] 3:DataType 4:Col 5:Table [fk] [fk] 6:Table 7:Col [pk] 8:DataType

$$\begin{tikzcd}
 & & \mathfrak{L} \rhd G & \vdash & & & \mathfrak{R} \rhd G & \\
 & & & & & \alpha(p_1) \arrow[ddl, "\delta_{p_1}"'] & \cdots & \alpha(p_n) \arrow[ddlll, "\delta_{p_n}"'] \\
\alpha(p_1) \arrow[drr, "\delta_{p_1}"'] & \cdots & \alpha(p_n) \arrow[d, "\delta_{p_n}"] & & \alpha(q_1) \arrow[d, "\delta_{q_1}"'] & & \cdots & \alpha(q_m) \arrow[dlll, "\delta_{q_m}"] \\
 & & L \arrow[rr, "id_L"] \arrow[dr, "\iota^L"'] & & R \arrow[dl, "\iota^R"] & & & \\
 & & & G & & & &
\end{tikzcd}$$

Figure 4.17: A typed specification entailment gives rise to a universal constraint

Example 18 (Specification Entailments and Universal Constraints) Table 4.6 shows the universal constraints c_1, c_2, c_3 and c_4. The universal constraints c_1, c_2 and c_4 correspond to the specification entailments in Fig. 3.6, Fig. 4.10 and Fig. 4.15, respectively. Moreover, c_3 corresponds to both entailments in Figs. 4.13 and 4.14 $\diamondsuit$

Recall that specification entailments are used to describe (or to require) properties of the semantic interpretation of predicates (see Remark 4 and Section 3.4.3). In contrast, universal constraints describe (or require) specific overall structures in specifications. A specification satisfies a universal constraint, corresponding to a certain specification entailment, if the corresponding semantic properties are fully syntactically presented in the specification.

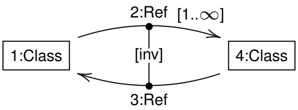
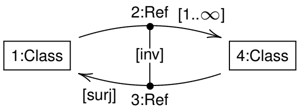
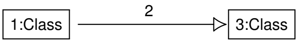
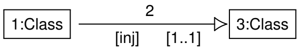
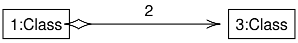
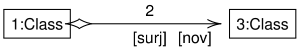
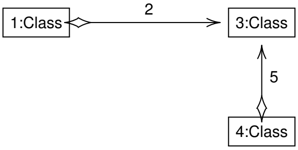
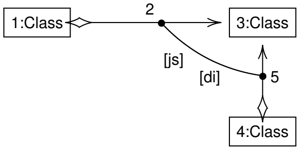
Remark 22 (Revisiting Constraint Classification) *In Section 1.5 we proposed a classification of constraints based on their origins and their affect, and organised them into the following categories: structural and attached constraints SC_{n+1}, AC_{n+1} which affect models at level M_n; and structural and attached constraints SC_n, AC_n which affect models at level M_{n-1}. In this regard, given a modelling formalism $(\Sigma_{n+1} \rhd S_{n+1}, \mathfrak{S}_{n+1}, \Sigma_{n+2})$, the constraints AC_{n+1} and SC_{n+1} are covered by*

- *the typing restrictions defined by S_{n+1},*
- *the atomic constraints $C^{\mathfrak{S}_{n+1}}$, and*
- *the universal constraints $r : \mathfrak{L} \rhd S_{n+1} \to \mathfrak{R} \rhd S_{n+1}$.*

Likewise, given a specification $((S_n, C^{\mathfrak{S}_n} : \Sigma_{n+1}) \blacktriangleright_{\iota^{S_n}} \mathfrak{S}_{n+1})$ at level M_n, the constraints AC_n and SC_n are covered by

- *the typing restrictions defined by S_n, and*
- *the atomic constraints $C^{\mathfrak{S}_n}$.*

Table 4.6: The universal constraints c_1, c_2, c_3 and c_4 corresponding to the specification entailments in Fig. 3.6, Fig. 4.10, (Fig. 4.13 and 4.14) and Fig. 4.15, respectively

$\mathfrak{L} \triangleright S_2$	$\mathfrak{R} \triangleright S_2$
c_1: `[mult(1,∞)]` and `[inverse]` entail `[surjective]`	
	 2:Ref [1..∞] 1:Class [inv] 4:Class [surj] 3:Ref
c_2: Inheritance entails `[injective]` and `[mult(1,1)]`	
	
c_3: Containment entails `[surjective]` and `[non-overlapping]`	
	
c_4: Two containments entail `[disjoint-image]` and `[jointly-surjective_2]`	
	

CHAPTER 5

Constraint-Aware Model Transformation

In the previous chapters, we formalised (meta)models as diagrammatic specifications and (meta)modelling languages as modelling formalisms. In this formalisation, we integrated structural and attached constraints in diagrammatic specifications. In constraint-aware model transformation, these constraints are taken into consideration both during definition and during application of model transformations. In this chapter, constraint-aware model transformation is discussed as an approach to model transformation in MDE.

5.1 Introduction

In the DPF based formalisation of MOF-based modelling languages, structural constraints and attached constraints are integrated in diagrammatic specifications. In more details, models are formalised as diagrammatic specifications which consist of an underlying graph together with a set of atomic constraints (see Chapter 3). Moreover, modelling languages are formalised as modelling formalisms $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$. The specification $\mathfrak{S}_2$ represents the metamodel of the language; the signature Σ_3 contains predicates which are used to add constraints to the metamodel $\mathfrak{S}_2$; while the typed signature $\Sigma_2 \rhd S_2$ contains predicates which are used to add constraints to specifications $\mathfrak{S}_1$ that are specified by the modelling formalism. The specifications $\mathfrak{S}_1$ need to conform to $\mathfrak{S}_2$; i.e. they need to satisfy all typing and atomic constraints which are specified by $\mathfrak{S}_2$. In addition to these constraints, one may also define universal constraints which have to be satisfied by all specifications $\mathfrak{S}_1$ that are specified by the modelling formalism (see Chapter 4).

In the DPF based approach to model transformation, these constraints are taken into account by introducing the concept of "constraint-aware model transforma-

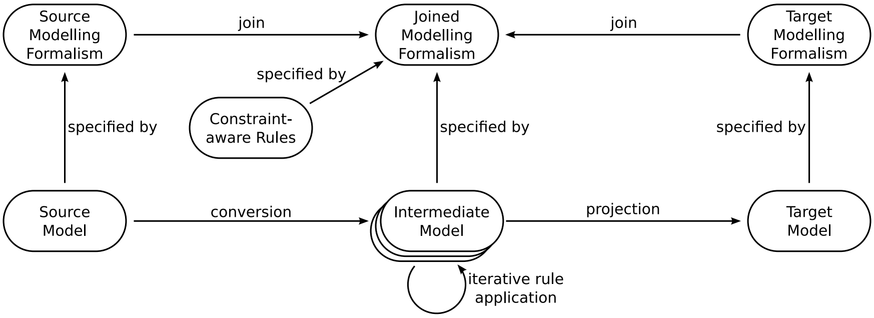


Figure 5.1: Overview of the DPF based approach to model transformation

tion" [107; 108]. Constraint-aware model transformation is a technique which supports specifying constraints in input and output patterns and uses these constraints to control

- which structure to create in the target specification and;
- which constraints to add to the created structure.

In this respect, it can be considered as an enhancement of the formal framework of graph transformations [41] in the sense that it can be used to transform the structure of models as well as constraints. That is, it offers more fine-grained means to describe, control and execute model transformations. Similar to graph transformations, the DPF based approach may be used to solve different transformation tasks. However, in this thesis the approach is demonstrated by a heterogeneous, out-place model transformation.

Recall that a model transformation consists of the automatic generation of target models from source models, according to a transformation definition [31; 81; 126]. Thus, model transformation consists basically of two tasks: *definition* of transformations, and, *application* of these transformations.

Recall also that source and target models may be specified by the same modelling language or by different modelling languages. In order to cover the general case, we consider here the transformation between models defined by different modelling languages. In this regard, the definition task is performed in two steps. The first step consists of relating the source and the target modelling languages to each other; that is, constructing an appropriate *joined modelling language*. An appropriate joined modelling language is a language which can be used to define both the source and the target models (see Fig. 5.1). The second step consists of using the joined modelling language to define model transformation rules. The definition task is usually performed only once for each pair of modelling languages, and reused during the application task.

The application task is performed in three steps. The first step consists of converting each source model to an intermediate model; i.e. a model which is defined

$$\begin{tikzcd}
\mathfrak{S}_2 \arrow[r, "join"] & \mathfrak{J}_2 & \mathfrak{T}_2 \arrow[l, "join"'] \\
\mathfrak{S}_1 \arrow[u] \arrow[r, "conversion"] & \mathfrak{J}_1 \arrow[u] \arrow[r, "projection"] & \mathfrak{T}_1 \arrow[u]
\end{tikzcd}$$

Figure 5.2: Join, conversion and projection

by the joined modelling language. In the second step, the transformation rules are applied iteratively to the intermediate models. In the third step, target models will be projected out from the final intermediate models.

In DPF terms, the definition task of constraint-aware model transformation is carried out as follows (see Fig. 5.2):

1. Define an appropriate joined modelling formalism $(\Gamma_2 \triangleright J_2, \mathfrak{J}_2, \Gamma_3)$ together with morphisms from the source $(\Sigma_2 \triangleright S_2, \mathfrak{S}_2, \Sigma_3)$ and target $(\Theta_2 \triangleright T_2, \mathfrak{T}_2, \Theta_3)$ modelling formalisms into $(\Gamma_2 \triangleright J_2, \mathfrak{J}_2, \Gamma_3)$.
2. Define (non-deleting) transformation rules as typed specification morphisms $r : \mathfrak{L} \triangleright J_2 \to \mathfrak{R} \triangleright J_2$.

Given a conformant source specification $\mathfrak{S}_1 \blacktriangleright \mathfrak{S}_2$, the application task is performed as follows:

1. Convert $\mathfrak{S}_1 \blacktriangleright \mathfrak{S}_2$ to a typed specification $\mathfrak{J}_1 \triangleright J_2$.
2. Apply the transformation rules iteratively to $\mathfrak{J}_1 \triangleright J_2$, and obtain a conformant specification $\mathfrak{J}'_1 \blacktriangleright \mathfrak{J}_2$.
3. Project out a conformant target specification $\mathfrak{T}_1 \blacktriangleright \mathfrak{T}_2$ from $\mathfrak{J}'_1 \blacktriangleright \mathfrak{J}_2$.

A running example is used to illustrate constraint-aware model transformations. It presents a transformation of a structural object-oriented model to a relational data model. The modelling formalisms used to define these models are introduced in Sections 4.3.1 and 4.3.2. In this example, the syntax used for the definition of the transformation rules is the same as the syntax used to specify the models themselves (see [12; 24; 55; 56] for references to and arguments for the usage of this kind of "concrete syntax" for the definition of model transformation rules).

5.2 Relating Modelling Formalisms

The first step of the definition task, as mentioned, is based on morphisms between modelling formalisms. Since both the source and the target modelling formalisms will be related to the joined modelling formalism in the same way, we describe the morphism between modelling formalisms in a generic way; i.e. from $(\Sigma_2 \triangleright S_2, \mathfrak{S}_2, \Sigma_3)$ to $(\Sigma'_2 \triangleright S'_2, \mathfrak{S}'_2, \Sigma'_3)$. A modelling formalism morphism should satisfy the following properties:

$$\begin{tikzcd}
\Sigma_3 \arrow[dr, dashed, "C^{\mathfrak{S}_2}"] & & & & & \Sigma'_3 \arrow[dl, dashed, "C^{\mathfrak{S}'_2}"'] \\
\Sigma_2 \arrow[r, dotted, "\tau^{\Sigma_2}"'] & S_2 & & & S'_2 & \Sigma'_2 \arrow[l, dotted, "\tau^{\Sigma'_2}"]
\end{tikzcd}$$

Figure 5.3: Two unrelated modelling formalisms

- It should enable the conversion of any conformant specification $((S_1, C^{\mathfrak{S}_1}\colon \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$ to a typed specification $((S'_1, C^{\mathfrak{S}'_1}\colon \Sigma'_2) \triangleright_{\iota^{S'_1}} S'_2)$. That is, it should define a *conversion functor* from $\mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2)$ to $\mathsf{TSpec}(\Sigma'_2 \triangleright S'_2)$.
- It should enable the projection of a conformant specification $((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$ from any conformant specification $((S'_1, C^{\mathfrak{S}'_1} : \Sigma'_2) \blacktriangleright_{\iota^{S'_1}} \mathfrak{S}'_2)$. That is, it should define a *projection functor* from $\mathsf{CSpec}(\Sigma'_2 \blacktriangleright \mathfrak{S}'_2)$ to $\mathsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2)$

Recall that a modelling formalism $(\Sigma_2 \triangleright S_2, \mathfrak{S}_2, \Sigma_3)$ consists of a signature $\Sigma_3 = (P^{\Sigma_3}, \alpha^{\Sigma_3})$, a metamodel $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2}\colon \Sigma_3)$ and a typed signature $\Sigma_2 \triangleright S_2 = ((P^{\Sigma_2}, \alpha^{\Sigma_2}) \triangleright_{\tau^{\Sigma_2}} S_2)$. Hence, in order to relate modelling formalisms $(\Sigma_2 \triangleright S_2, \mathfrak{S}_2, \Sigma_3)$ and $(\Sigma'_2 \triangleright S'_2, \mathfrak{S}'_2, \Sigma'_3)$ (see Fig. 5.3), we need to define

1. a signature morphism from Σ_3 to Σ'_3,
2. a specification morphism from $\mathfrak{S}_2$ to $\mathfrak{S}'_2$, and
3. a typed signature morphism from $\Sigma_2 \triangleright S_2$ to $\Sigma'_2 \triangleright S'_2$.

First, we define signature morphisms (see Fig. 5.4).

Definition 32 (Signature Morphism) *Given signatures $\Sigma_3 = (P^{\Sigma_3}, \alpha^{\Sigma_3})$ and $\Sigma'_3 = (P^{\Sigma'_3}, \alpha^{\Sigma'_3})$, a signature morphism $\sigma_3 : \Sigma_3 \to \Sigma'_3$ is a mapping $\sigma_3 : P^{\Sigma_3} \to P^{\Sigma'_3}$ such that for any $p \in P^{\Sigma_3}$ it holds that $\alpha^{\Sigma_3}(p) = \alpha^{\Sigma'_3}(\sigma_3(p))$.*

A semantically compatible signature morphism is a signature morphism where $[\![\sigma_3(p)]\!]^{\Sigma'_3} \subseteq [\![p]\!]^{\Sigma_3}$.

Each signature morphism $\sigma_3 : \Sigma_3 \to \Sigma'_3$ gives rise to a functor σ_3^* between the corresponding categories of specifications $\mathsf{Spec}(\Sigma_3)$ and $\mathsf{Spec}(\Sigma'_3)$.

Proposition 4 (Signature Morphism and Translation Functor) *Given signatures $\Sigma_3 = (P^{\Sigma_3}, \alpha^{\Sigma_3})$ and $\Sigma'_3 = (P^{\Sigma'_3}, \alpha^{\Sigma'_3})$, a signature morphism $\sigma_3 : \Sigma_3 \to \Sigma'_3$ induces a functor $\sigma_3^* : \mathsf{Spec}(\Sigma_3) \to \mathsf{Spec}(\Sigma'_3)$ between the corresponding categories of specifications. That is,*

$$\begin{array}{rll}
\textit{given a specification} & \mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2}\colon \Sigma_3) & \textit{in } \mathsf{Spec}(\Sigma_3), \\
\textit{we define} & \sigma_3^*(\mathfrak{S}_2) = (S_2, \sigma_3^*(C^{\mathfrak{S}_2})\colon \Sigma'_3) & \textit{in } \mathsf{Spec}(\Sigma'_3) \\
\textit{where} & \sigma_3^*(C^{\mathfrak{S}_2}) = \{(\sigma_3(p), \delta) \mid (p, \delta) \in C^{\mathfrak{S}_2}\}. &
\end{array}$$

$$\begin{tikzcd}
\Sigma_3 \arrow[rrrr, "\sigma_3"] \arrow[dr, dashed, "C^{\mathfrak{S}_2}"] & & & & \Sigma'_3 \arrow[dll, dashed, "\sigma_3^*(C^{\mathfrak{S}_2})"'] \arrow[dl, dashed, "C^{\mathfrak{S}'_2}"] \\
\Sigma_2 \arrow[r, dotted, "\tau^{\Sigma_2}"'] & S_2 \arrow[r, equal] & \sigma_3^*(S_2) \arrow[r, "\phi_2"] & S'_2 & \Sigma'_2 \arrow[l, dotted, "\tau^{\Sigma'_2}"]
\end{tikzcd}$$

Figure 5.4: Signature morphism and heterogeneous specification morphism

$$\begin{tikzcd}
\Sigma_3 \arrow[rr, "\sigma_3"] & & \Sigma'_3 \\
\alpha^{\Sigma_3}(p) \arrow[rr, bend left, equal] \arrow[rr, phantom, "="] \arrow[dr, bend right, "\delta"'] & & \alpha^{\Sigma'_3}(\sigma_3(p)) \arrow[dl, bend left, "\delta"] \\
& S_2 &
\end{tikzcd}$$

for any specification morphism $\psi : \mathfrak{S}_2 \to \mathfrak{P}_2$ *in* $\mathsf{Spec}(\Sigma_3)$
we get a specification morphism $\sigma_3^*(\psi) := \psi : \sigma_3^*(\mathfrak{S}_2) \to \sigma_3^*(\mathfrak{P}_2)$ *in* $\mathsf{Spec}(\Sigma'_3)$.

Proof. We have to validate that $\sigma_3^*(\psi)$ preserves constraints. This is ensured by the following:

$$\begin{aligned}
(\sigma_3(p), \delta) \in \sigma_3^*(C^{\mathfrak{S}_2}) &\Leftrightarrow (p, \delta) \in C^{\mathfrak{S}_2} && \text{(Definition of } \sigma_3^*(C^{\mathfrak{S}_2})\text{)} \\
&\Rightarrow (p, \delta; \psi) \in C^{\mathfrak{P}_2} && (\psi : \mathfrak{S}_2 \to \mathfrak{P}_2 \text{ is specification morphism)} \\
&\Leftrightarrow (\sigma_3(p), \delta; \psi) \in \sigma_3^*(C^{\mathfrak{P}_2}) && \text{(Definition of } \sigma_3^*(C^{\mathfrak{P}_2})\text{)}
\end{aligned}$$

$\diamondsuit$

Now considering specification morphism from $\mathfrak{S}_2$ to $\mathfrak{S}'_2$. For the modelling formalisms $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$, $(\Sigma'_2 \rhd S'_2, \mathfrak{S}'_2, \Sigma'_3)$, a signature morphism $\sigma_3 : \Sigma_3 \to \Sigma'_3$ allows us to define a *heterogeneous specification morphism* between the metamodels $\mathfrak{S}_2$ and $\mathfrak{S}'_2$ (see Fig. 5.4). Since these two specifications belong to two different categories, i.e. $\mathsf{Spec}(\Sigma_3)$ and $\mathsf{Spec}(\Sigma'_3)$, respectively, the morphism $(\phi_2, \sigma_3) : \mathfrak{S}_2 \to \mathfrak{S}'_2$ is defined in two steps. Firstly, the induced functor σ_3^* maps the specification $\mathfrak{S}_2$ to a specification $\sigma_3^*(\mathfrak{S}_2)$ in $\mathsf{Spec}(\Sigma'_3)$. Then, a (homogeneous) specification morphism ϕ_2 (see Definition 13) is defined between $\sigma_3^*(\mathfrak{S}_2)$ and $\mathfrak{S}'_2$.

Definition 33 (Heterogeneous Specification Morphism) *Given two specifications* $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2}\colon \Sigma_3)$ *and* $\mathfrak{S}'_2 = (S'_2, C^{\mathfrak{S}'_2}\colon \Sigma'_3)$, *a heterogeneous specification*

$$\begin{tikzcd}
\Sigma_2 \arrow[r, dotted, "\tau^{\Sigma_2}"'] \arrow[rrr, bend left, "\sigma_2"] \arrow[dr, dashed, "C^{\mathfrak{S}_1}"'] & S_2 \arrow[r, "\phi_2"'] & S'_2 & \Sigma'_2 \arrow[l, dotted, "\tau^{\Sigma'_2}"] \arrow[dl, dashed, "C^{(\sigma_2,\phi_2)^*(\mathfrak{S}_1)}"] \\
 & S_1 \arrow[u, "\iota^{S_1}"'] \arrow[r, equal] & (\sigma_2,\phi_2)^*(S_1) \arrow[u, "\iota^{(\sigma_2,\phi_2)^*(S_1)}"] &
\end{tikzcd}$$

Figure 5.5: Typed signature morphism

morphism $(\phi_2, \sigma_3) : \mathfrak{S}_2 \to \mathfrak{S}'_2$ is given by a signature morphism $\sigma_3 : \Sigma_3 \to \Sigma'_3$ together with a specification morphism $\phi_2 : \sigma_3^*(\mathfrak{S}_2) \to \mathfrak{S}'_2$

$$\begin{tikzcd}
\Sigma_3 \arrow[r, "\sigma_3"] & \Sigma'_3
\end{tikzcd}$$

$$\begin{tikzcd}
\textsf{Spec}(\Sigma_3) \arrow[rrr, "\sigma_3^*"] & & & \textsf{Spec}(\Sigma'_3) & \\
\mathfrak{S}_2 \arrow[u, "\in"] \arrow[rr, mapsto] & & \sigma_3^*(\mathfrak{S}_2) \arrow[ur, "\in"] \arrow[rr, "\phi_2"] & & \mathfrak{S}'_2 \arrow[ul, "\in"']
\end{tikzcd}$$

After defining morphisms between the signatures Σ_3 and Σ'_3 and the specifications $\mathfrak{S}_2$ and $\mathfrak{S}'_2$, the next step is to define a *typed signature morphism* between the typed signatures $\Sigma_2 \rhd S_2$ and $\Sigma'_2 \rhd S'_2$ (see Fig. 5.5).

Definition 34 (Typed Signature Morphism) *Given typed signatures $\Sigma_2 \rhd S_2 = ((P^{\Sigma_2}, \alpha^{\Sigma_2}) \rhd_{\tau^{\Sigma_2}} S_2)$ and $\Sigma'_2 \rhd S'_2 = ((P^{\Sigma'_2}, \alpha^{\Sigma'_2}) \rhd_{\tau^{\Sigma'_2}} S'_2)$, a typed signature morphism $(\sigma_2, \phi_2) : \Sigma_2 \rhd S_2 \to \Sigma'_2 \rhd S'_2$ is given by a signature morphism $\sigma_2 : \Sigma_2 \to \Sigma'_2$ together with a graph homomorphism $\phi_2 : S_2 \to S'_2$ such that $\tau^{\Sigma_2}(p); \phi_2 = \tau^{\Sigma'_2}(\sigma_2(p))$ for all $p \in P^{\Sigma_2}$.*

A typed signature morphism (σ_2, ϕ_2) is semantically compatible if σ_2 is semantically compatible.

$$\begin{tikzcd}
\Sigma_2 \arrow[rrr, "\sigma_2"] & & & \Sigma'_2 \\
\alpha^{\Sigma_2}(p) \arrow[r, "\tau^{\Sigma_2}(p)"'] \arrow[rrr, equal, bend left, "="] & S_2 \arrow[r, "\phi_2"'] & S'_2 & \alpha^{\Sigma'_2}(\sigma_2(p)) \arrow[l, "\tau^{\Sigma'_2}(\sigma_2(p))"]
\end{tikzcd}$$

Similar to (untyped) signature morphisms, each typed signature morphism $(\sigma_2, \phi_2) : \Sigma_2 \rhd S_2 \to \Sigma'_2 \rhd S'_2$ gives rise to a functor $(\sigma_2, \phi_2)^*$ between the corresponding categories of typed specifications $\textsf{TSpec}(\Sigma_2 \rhd S_2)$ and $\textsf{TSpec}(\Sigma'_2 \rhd S'_2)$.

Proposition 5 (Typed Signature Morphism and Translation Functor) *Given typed signatures $\Sigma_2 \triangleright S_2$ and ${\Sigma'}_2 \triangleright {S'}_2$, a typed signature morphism $(\sigma_2, \phi_2) : \Sigma_2 \triangleright S_2 \rightarrow {\Sigma'}_2 \triangleright {S'}_2$ induces a functor $(\sigma_2, \phi_2)^* : \mathsf{TSpec}(\Sigma_2 \triangleright S_2) \rightarrow \mathsf{TSpec}({\Sigma'}_2 \triangleright {S'}_2)$ between the corresponding categories of typed specifications. That is,*

given a typed specification $((S_1, C^{\mathfrak{S}_1}\colon \Sigma_2) \triangleright_{\iota^{S_1}} S_2)$ *in* $\mathsf{TSpec}(\Sigma_2 \triangleright S_2)$*,*
we define $(\sigma_2, \phi_2)^*((S_1, C^{\mathfrak{S}_1}\colon \Sigma_2) \triangleright_{\iota^{S_1}} S_2) =$
$((S_1, (\sigma_2, \phi_2)^*(C^{\mathfrak{S}_1})\colon {\Sigma'}_2) \triangleright_{\iota^{S_1};\phi_2} {S'}_2)$
in $\mathsf{TSpec}({\Sigma'}_2 \triangleright {S'}_2)$
where $(\sigma_2, \phi_2)^*(C^{\mathfrak{S}_1}) = \{(\sigma_2(p), \delta) \mid (p, \delta) \in C^{\mathfrak{S}_1}\}$.

$$\begin{tikzcd}
\Sigma_2 \arrow[rrrr, "\sigma_2"] & & & & {\Sigma'}_2 \\
\alpha^{\Sigma_2}(p) \arrow[r, dotted, "\tau^{\Sigma_2}(p)"'] \arrow[rrrr, bend left, equal, "="] \arrow[rrd, bend right, "\delta"'] & S_2 \arrow[rr, "\phi_2"'] \arrow[d, phantom, "="] & \arrow[d, phantom, "="] & {S'}_2 \arrow[d, phantom, "="] & \alpha^{{\Sigma'}_2}(\sigma_2(p)) \arrow[l, dotted, "\tau^{{\Sigma'}_2}(\sigma_2(p))"] \arrow[lld, bend left, "\delta"] \\
& & S_1 \arrow[lu, "\iota^{S_1}"] \arrow[ru, "\iota^{S_1};\phi"'] & &
\end{tikzcd}$$

for any typed specification morphism
$\psi : ((S_1, C^{\mathfrak{S}_1}\colon \Sigma_2) \triangleright_{\iota^{S_1}} S_2) \rightarrow ((P_1, C^{\mathfrak{P}_1}\colon \Sigma_2) \triangleright_{\iota^{P_1}} S_2)$ *in* $\mathsf{TSpec}(\Sigma_2 \triangleright S_2)$*,*
we get a specification morphism
$(\sigma_2, \phi_2)^*(\psi) := \psi : ((S_2, (\sigma_2, \phi_2)^*(C^{\mathfrak{S}_1})\colon {\Sigma'}_2) \triangleright_{\iota^{S_1};\phi_2} {S'}_2) \rightarrow$
$((P_1, (\sigma_2, \phi_2)^*(C^{\mathfrak{P}_1})\colon {\Sigma'}_2) \triangleright_{\iota^{P_1};\phi_2} {S'}_2)$ *in* $\mathsf{TSpec}({\Sigma'}_2 \triangleright {S'}_2)$*.*

Proof. In addition to the proof of Proposition 4, we have to validate that $(\sigma_2, \phi_2)^*(C^{\mathfrak{S}_1})$ is compatible with typing, that is, $\forall (\sigma_2(p), \delta) \in (\sigma_2, \phi_2)^*(C^{\mathfrak{S}_1}) : \delta; (\iota^{S_1}; \phi_2) = \tau^{{\Sigma'}_2}(\sigma_2(p))$. This is ensured by the following:

$$\begin{aligned} \delta; (\iota^{S_1}; \phi_2) &= \tau^{\Sigma_2}(p); \phi_2 \qquad (C^{\mathfrak{S}_1} \text{ is type compatible}) \\ &= \tau^{{\Sigma'}_2}(\sigma_2(p)) \qquad (\phi_2 \text{ is type compatible}) \end{aligned}$$

We can validate that $(\sigma_2, \phi_2)^*(\psi)$ preserves constraints in the same way as in Proposition 4. It remains only to validate that $(\sigma_2, \phi_2)^*(\psi)$ is also compatible with typing. This is ensured by definition; i.e. $\psi; \iota^{P_1} = \iota^{S_1}$ implies $\psi; (\iota^{P_1}; \phi_2) = \iota^{S_1}; \phi_2$. $\diamondsuit$

Immediately from Propositions 2, 4 and 5 we obtain the following corollary.

Corollary 1 *Given a semantically compatible signature morphism $\sigma_3 : \Sigma_3 \rightarrow {\Sigma'}_3$, for any specification $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2}\colon \Sigma_3)$ in $\mathsf{Spec}(\Sigma_3)$, we have $\mathsf{Inst}(\sigma_3^*(\mathfrak{S}_2)) \subseteq \mathsf{Inst}(\mathfrak{S}_2)$.*

Similarly, given a semantically compatible typed signature morphism $(\sigma_2, \phi_2) : (\Sigma_2 \rhd S_2) \to (\Sigma'_2 \rhd S'_2)$*, for any typed specification* $\mathfrak{S}_1 \rhd S_2 = ((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \rhd_{\iota^{S_1}} S_2)$ *in* $\textsf{TSpec}(\Sigma_2 \rhd S_2)$*, we have* $\textsf{Inst}((\sigma_2, \phi_2)^*(\mathfrak{S}_1 \rhd S_2)) \subseteq \textsf{Inst}(\mathfrak{S}_1 \rhd S_2)$.

Remark 23 (Commutative Functors) *Based on Definition 22 (see Section 4.2) and Propositions 4 and 5, we obtain the following commutative functors between categories of (typed) specifications.*

$$\begin{tikzcd}
\textsf{TSpec}(\Sigma_2 \rhd S_2) \arrow[r, "U^{S_2}"] \arrow[d, "(\sigma_2,\phi_2)^*"'] \arrow[dr, phantom, "="] & \textsf{Spec}(\Sigma_2) \arrow[d, "\sigma_2^*"] \\
\textsf{TSpec}(\Sigma'_2 \rhd S'_2) \arrow[r, "U^{S'_2}"] & \textsf{Spec}(\Sigma'_2)
\end{tikzcd}$$

Proposition 6 (Signature Morphism and Category of Instances) *Given a semantically compatible signature morphism* $\sigma_3 : \Sigma_3 \to \Sigma'_3$*, a heterogeneous specification morphism* $(\phi_2, \sigma_3) : \mathfrak{S}_2 \to \mathfrak{S}'_2$ *with* $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2} : \Sigma_3)$ *and* $\mathfrak{S}'_2 = (S'_2, C^{\mathfrak{S}'_2} : \Sigma'_3)$ *induces a forgetful functor* $(\phi_2, \sigma_3)^\bullet : \textsf{Inst}(\mathfrak{S}'_2) \to \textsf{Inst}(\mathfrak{S}_2)$.

Proof. Follows from Proposition 2 and Corollary 1. That is,

$$(\phi_2, \sigma_3)^\bullet : \textsf{Inst}(\mathfrak{S}'_2) \to \textsf{Inst}(\sigma_3^*(\mathfrak{S}_2)) \subseteq \textsf{Inst}(\mathfrak{S}_2)$$

$\diamondsuit$

We are now able to define morphisms between modelling formalisms in such a way that the requirements ensuring the conversion and projection steps in our approach are fulfilled.

Definition 35 (Modelling Formalism Morphism) *Given modelling formalisms* $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ *and* $(\Sigma'_2 \rhd S'_2, \mathfrak{S}'_2, \Sigma'_3)$*, a modelling formalism morphism* $(\sigma_3, \phi_2, \sigma_2) : (\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3) \to (\Sigma'_2 \rhd S'_2, \mathfrak{S}'_2, \Sigma'_3)$ *is given by:*

- *a semantically compatible signature morphism* $\sigma_3 : \Sigma_3 \to \Sigma'_3$ *and*
- *a graph homomorphism* $\phi_2 : S_2 \to S'_2$ *such that*
  - $(\sigma_2, \phi_2) : \Sigma_2 \rhd S_2 \to \Sigma'_2 \rhd S'_2$ *is a typed signature morphism and*
  - $(\phi_2, \sigma_3) : \mathfrak{S}_2 \to \mathfrak{S}'_2$ *is a heterogeneous specification morphism.*

The conversion step is ensured by the existence of a conversion functor.

Proposition 7 (Conversion Functor) *Given modelling formalisms* $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ *and* $(\Sigma'_2 \rhd S'_2, \mathfrak{S}'_2, \Sigma'_3)$*, a typed signature morphism* $(\sigma_2, \phi_2) : \Sigma_2 \rhd S_2 \to \Sigma'_2 \rhd S'_2$ *provides a conversion functor*

$$\begin{tikzcd}
\textsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2) \arrow[r, "(\sigma_2,\phi_2)^*"] & \textsf{TSpec}(\Sigma'_2 \rhd S'_2)
\end{tikzcd}$$

For any conformant specification $((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$ *we define a typed specification* $(\sigma_2, \phi_2)^*((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2) = ((S'_1, C^{\mathfrak{S}'_1} : \Sigma'_2) \rhd_{\iota^{S'_1}} S'_2)$.

Proof. Follows immediately from Proposition 5 and the inclusion $\textsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2) \subseteq \textsf{TSpec}(\Sigma_2 \rhd S_2)$ in Definition 23. $\diamondsuit$

Also, the projection step is ensured by the existence of a functor.

Proposition 8 (Projection Functor) *Given modelling formalisms* $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ *and* $(\Sigma'_2 \rhd S'_2, \mathfrak{S}'_2, \Sigma'_3)$*, a modelling formalism morphism* $(\sigma_3, \phi_2, \sigma_2) : (\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3) \to (\Sigma'_2 \rhd S'_2, \mathfrak{S}'_2, \Sigma'_3)$ *provides a projection functor*

$$\textsf{CSpec}(\Sigma'_2 \blacktriangleright \mathfrak{S}'_2) \xrightarrow{(\sigma_3, \phi_2, \sigma_2)^\bullet} \textsf{CSpec}(\Sigma_2 \blacktriangleright \mathfrak{S}_2)$$

For any conformant specification $((S'_1, C^{\mathfrak{S}'_1} : \Sigma'_2) \blacktriangleright_{\iota^{S'_1}} \mathfrak{S}'_2)$ *we define a conformant specification* $(\sigma_3, \phi_2, \sigma_2)^\bullet((S'_1, C^{\mathfrak{S}'_1} : \Sigma'_2) \blacktriangleright_{\iota^{S'_1}} \mathfrak{S}'_2) = ((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$.

Proof. The underlying graph S_1 together with the graph homomorphism $\iota^{S_1} : S_1 \to S_2$ are constructed by a pullback of $S_2 \xrightarrow{\phi_2} S'_2 \xleftarrow{\iota^{S'_1}} S'_1$. Proposition 6 ensures $(S_1, \iota^{S_1}) \in \textsf{Inst}(\mathfrak{S}_2)$.

$$\begin{tikzcd}
\Sigma_3 \arrow[rrr, "\sigma_3"] \arrow[dr, dashed, "C^{\mathfrak{S}_2}"] & & & \Sigma'_3 \arrow[dl, dashed, "C^{\mathfrak{S}'_2}"'] \\
\Sigma_2 \arrow[r, dotted, "\tau^{\Sigma_2}"'] \arrow[rrr, bend left, "\sigma_2"] \arrow[dr, dashed, "C^{\mathfrak{S}_1}"'] & S_2 \arrow[r, "\phi_2"'] & S'_2 & \Sigma'_2 \arrow[l, dotted, "\tau^{\Sigma'_2}"] \arrow[dl, dashed, "C^{\mathfrak{S}'_1}"] \\
& S_1 \arrow[u, "\iota^{S_1}"'] \arrow[r, "\phi_1"'] & S'_1 \arrow[u, "\iota^{S'_1}"] &
\end{tikzcd}$$

It remains to construct the set $C^{\mathfrak{S}_1}$ of constraints. For any $p \in P^{\Sigma_2}$ and any constraint $(\sigma_2(p), \delta') \in C^{\mathfrak{S}'_1}$ we construct a constraint $(p, \delta) \in C^{\mathfrak{S}_1}$ as follows

$$\begin{tikzcd}
\Sigma_2 \arrow[rrr, "\sigma_2"] & & & \Sigma'_2 \\
\alpha^{\Sigma_2}(p) \arrow[r, dotted, "\tau^{\Sigma_2}(p)"'] \arrow[rrr, bend left, equal, "="] \arrow[dr, "\delta"'] \arrow[dr, phantom, bend left, "="] & S_2 \arrow[r, "\phi_2"] \arrow[dr, phantom, "P.B."] & S'_2 & \alpha^{\Sigma'_2}(\sigma_2(p)) \arrow[l, dotted, "\tau^{\Sigma'_2}(\sigma(p))"'] \arrow[dl, "\delta'"] \arrow[dl, phantom, bend right, "="] \\
& S_1 \arrow[u, "\iota^{S_1}"'] \arrow[r, "\phi_1"'] & S'_1 \arrow[u, "\iota^{S'_1}"] &
\end{tikzcd}$$

Due to the definition of typed signature morphisms we have $\alpha^{\Sigma_2}(p) = \alpha^{\Sigma'_2}(\sigma_2(p))$ and $\tau^{\Sigma_2}(p); \phi_2 = \tau^{\Sigma'_2}(\sigma_2(p))$. Since $C^{\mathfrak{S}'_1}$ is type compatible we obtain, in such a way, $\tau^{\Sigma_2}(p); \phi_2 = \delta'; \iota^{S'_1}$ thus exists, due to the universal property of the pullback, a unique $\delta : \alpha^{\Sigma_2}(p) \to S_1$ such that $\delta; \phi_1 = \delta'$ and $\delta; \iota^{S_1} = \tau^{\Sigma_2}(p)$. The second equation means that the constructed constraint $(p, \delta) \in C^{\mathfrak{S}_1}$ is also type compatible. $\diamondsuit$

5.3 Joined Modelling Formalism

We have explained how modelling formalism morphisms are defined and which properties these morphisms should possess. In this section, we explain the first step of the definition task; i.e. how a joined modelling formalism and the corresponding morphisms may be defined in practice. A possible way to obtain a joined modelling formalism is to construct the disjoint union of the components of the source and target modelling formalisms and to add additional auxiliary components to this disjoint union as shown in [107]. In this case, the morphisms from the source and target modelling formalisms to the joined modelling formalism are given by the injections which we obtain according to the disjoint union construction.

Roughly speaking, given the source $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ and the target $(\Theta_2 \rhd T_2, \mathfrak{T}_2, \Theta_3)$ modelling formalisms (see Fig. 5.6a and Fig. 5.6c, respectively), a joined modelling formalism $(\Gamma_2 \rhd J_2, \mathfrak{J}_2, \Gamma_3)$ is defined by the disjoint union construction (see Fig. 5.6b). In more detail, the source and target metamodels are joined together to $\mathfrak{J}_2 := \mathfrak{S}_2 \uplus \mathfrak{K}_2 \uplus \mathfrak{T}_2$, and the source and target signatures are joined together to $\Gamma_2 \rhd J_2 := \Sigma_2 \rhd S_2 \uplus \Theta_2 \rhd T_2$ and $\Gamma_3 := \Sigma_3 \uplus \Xi_3 \uplus \Theta_3$, where $\uplus$ denotes the disjoint union operation (see Example 19). In $\mathfrak{J}_2$, the component $\mathfrak{K}_2$ represents the correspondence between $\mathfrak{S}_2$ and $\mathfrak{T}_2$. In most cases, the elements in $\mathfrak{K}_2$ will be arrows connecting nodes in $\mathfrak{S}_2$ and $\mathfrak{T}_2$. However, in some cases it may be convenient to also have auxiliary nodes in $\mathfrak{K}_2$ and arrows connecting these nodes with elements in $\mathfrak{S}_2$ and/or $\mathfrak{T}_2$. In Γ_3, the component Ξ_3 contains predicates which are used to add constraints, and thus additional requirements, to $\mathfrak{J}_2$. That is, $\mathfrak{K}_2$ is not a specification independently, similarly, Ξ_3 is not a signature independently.

In practise, the definitions of $\mathfrak{K}_2$ and Ξ_3 are often done manually by transformation designers during the definition task of model transformation. This is the essential and creative part which normally cannot be performed fully automatically unless the source and target modelling formalisms are very similar.

Example 19 (Joined Modelling Formalism) Recall Sections 4.3.1 and 4.3.2 in which we introduced the source modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ for specifying structural object-oriented models and the target modelling formalism $(\Theta_2 \rhd T_2, \mathfrak{T}_2, \Theta_3)$ for specifying relational data models. Fig. 5.7 shows a joined modelling formalism $(\Gamma_2 \rhd J_2, \mathfrak{J}_2, \Gamma_3)$. Moreover, Table 5.1 shows the component Ξ_3 of Γ_3. Note that the node DataType in $\mathfrak{S}_2$ and $\mathfrak{T}_2$ is renamed to DataType^s and DataType^t in $\mathfrak{J}_2$ by the disjoint union operation. $\diamondsuit$

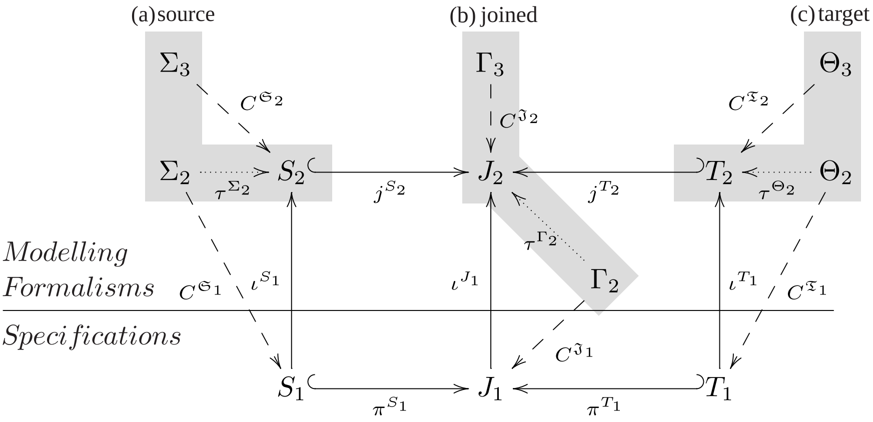


Figure 5.6: Source, target and joined modelling formalisms

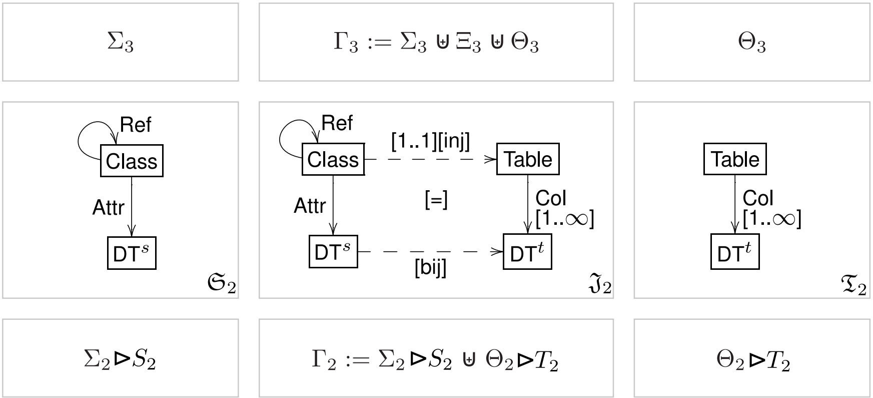


Figure 5.7: A joined modelling formalism for structural object-oriented models and relational data models

Table 5.1: A sample signature Ξ_3

p	$\alpha^{\Xi_3}(p)$	**Proposed vis.**	**Semantic Interpret.**
`[commutative]`	$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[d, "g"'] & 2 \arrow[d, "g'"] \\ 3 \arrow[r, "f'"'] & 4 \end{tikzcd}$	$\begin{tikzcd} \boxed{\mathsf{X}} \arrow[r, "\mathsf{f}"] \arrow[d, "\mathsf{g}"'] \arrow[dr, phantom, "{[=]}"] & \boxed{\mathsf{Y}} \arrow[d, "\mathsf{g}'"] \\ \boxed{\mathsf{Z}} \arrow[r, "\mathsf{f}'"'] & \boxed{\mathsf{Æ}} \end{tikzcd}$	$\forall x \in X :$ $g'(f(x)) = f'(g(x))$
`[bijective]`	$\begin{tikzcd} 1 \arrow[r, "f"] & 2 \end{tikzcd}$	$\begin{tikzcd} \boxed{\mathsf{X}} \arrow[r, "\mathsf{f}", "{[\mathsf{bij}]}"'] & \boxed{\mathsf{Y}} \end{tikzcd}$	*f* is `[mult(1,1)]`, `[injective]` and `[surjective]`

Remark 24 (Universal Constraints in Joined Modelling Formalisms) *The universal constraints from the source modelling formalism will be checked to validate source specifications. Moreover, after the projection of target models, the universal constraints of the target modelling formalism will be checked to validate the target specifications. The universal constraints of the target modelling formalism may also be translated to the joined modelling formalism by the conversion functor (see Proposition 7) and applied as transformation rules (see Remark 20).*

5.4 Constraint-Aware Transformation Rules

The second step in the definition task of model transformation consists of the definition of constraint-aware transformation rules. These rules are defined as typed specification morphisms; that is, the input and output patterns are typed specifications. Moreover, we use none-deleting (or monotonic) transformation rules in our approach. As a consequence, in each transformation rule the input pattern is included in the output pattern.

Definition 36 (Transformation Rule) *Given a modelling formalism* $(\Sigma_2 \triangleright S_2, \mathfrak{S}_2, \Sigma_3)$*, a transformation rule is a typed specification morphism* $r : \mathfrak{L} \triangleright S_2 \hookrightarrow \mathfrak{R} \triangleright S_2$ *between the input and output patterns* $\mathfrak{L} \triangleright S_2$ *and* $\mathfrak{R} \triangleright S_2$*, with r being an inclusion.*

$$\begin{tikzcd} & S_2 \arrow[d, phantom, "="] & \\ L \arrow[ur, "\iota^L"] \arrow[rr, hook, "r"] & & R \arrow[ul, "\iota^R"'] \end{tikzcd}$$

$$\begin{tikzcd} \mathfrak{L} \triangleright S_2 \arrow[r, hook, "r"] & \mathfrak{R} \triangleright S_2 \end{tikzcd}$$

Recall that a model transformation definition consists of a set of transformation rules and its application consists of the iterative application of these rules. In our approach, given a modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$, the application of a transformation rule is given by a pushout construction in the category $\mathsf{TSpec}(\Sigma_2 \rhd S_2)$ (see Proposition 11 in Appendix A.1.2).

Definition 37 (Application of Transformation Rule) *Let* $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ *be a modelling formalism,* $\mathfrak{S}_1 \rhd S_2$ *a typed specification, and* $r : \mathfrak{L} \rhd S_2 \hookrightarrow \mathfrak{R} \rhd S_2$ *a transformation rule. An application* $\langle r, m \rangle$ *of* r *via a match* $m : \mathfrak{L} \rhd S_2 \to \mathfrak{S}_1 \rhd S_2$*, where* m *is a typed specification morphism, is given by a pushout of* $\mathfrak{S}_1 \rhd S_2 \xleftarrow{m} \mathfrak{L} \rhd S_2 \xrightarrow{r} \mathfrak{R} \rhd S_2$ *in* $\mathsf{TSpec}(\Sigma_2 \rhd S_2)$

$$\begin{tikzcd}
\mathfrak{L} \rhd S_2 \arrow[r, hook, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "PO"] & \mathfrak{R} \rhd S_2 \arrow[d, "m^*"] \\
\mathfrak{S}_1 \rhd S_2 \arrow[r, hook, "\langle r{,}m \rangle"] & \mathfrak{S^*}_1 \rhd S_2
\end{tikzcd}$$

Note that in constraint-aware transformation rules, the output patterns are not only dependent on the structure of the input patterns, but also on the constraints. That is, the input patterns $\mathfrak{L}_1$ and $\mathfrak{L}_2$ of two rules $r_1 : \mathfrak{L}_1 \to \mathfrak{R}_1$ and $r_2 : \mathfrak{L}_2 \to \mathfrak{R}_2$ may have the same underlying graphs $L_1 = L_2$, however, depending on differences between $C^{\mathfrak{L}_1}$ and $C^{\mathfrak{L}_2}$, the output patterns $\mathfrak{R}_1$ and $\mathfrak{R}_2$, and especially the underlying graphs R_1 and R_2, may be different. This makes constraint-aware transformation rules more fine-grained and expressive in the sense that one can consider more transformation cases based on constraints. This feature of expressiveness comes in addition to the capability of transforming constraints.

Recall that the modelling formalisms in our running example, defined in Sections 4.3.1 and 4.3.2, are designed for multi-valued and single-valued semantic environments, respectively. In order to define transformation rules between models specified by these formalisms, one needs to determine how functions and predicates from a multi-valued semantic environment are represented in a single-valued environment. We discuss now the representation of multi-valued functions, as well as the composition and image-inclusion of these functions in a single-valued modelling environment. In Example 20, the representation of multi-valued functions is reflected in rules r_4, r_5 and r_6; while composition and image-inclusion are reflected in rule r_7.

For a multi-valued function $f : X \to Y$, the graph of f is defined by $gr(f) = \{(x, y) | y \in f(x)\}$. The set of tuples $gr(f)$ together with the projections π_1, π_2 and the predicate `[jointly-injective]` represent the multi-valued function f in a single-valued environment (see Fig. 5.8).

For two multi-valued functions $f : X \to Y$ and $g : Y \to Z$, the composition $f;g$ is defined as $(f;g)(x) = g(f(x)) = \{z \in Z \mid \exists y \in Y : y \in f(x) \wedge z \in g(y)\}$. This is indicated by the predicate `[composition]` in Fig. 5.9a. In a single-valued environment, this composition is obtained in two steps. In the first

$$\begin{tikzcd}
X \arrow[r, "f"] & Y
\end{tikzcd}$$

(a) Multi-valued function

$$\begin{tikzcd}
X \arrow[rr, dashed, "f"] & & Y \\
& gr(f) \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] \arrow[u, phantom, "{[ji]}"] &
\end{tikzcd}$$

(b) Its representation in a single-valued environment

Figure 5.8: Representation of multi-valued function in a single-valued environment

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[rr, bend left, "f;g", "{[comp]}"'] & Y \arrow[r, "g"] & Z
\end{tikzcd}$$

(a) Composition in multi-valued environment

$$\begin{tikzcd}
X \arrow[rr, dashed, "f"] \arrow[rrrr, dashed, bend left, "f;g"] & & Y \arrow[rr, dashed, "g"] & & Z \\
& gr(f) \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] & P.B. & gr(g) \arrow[ul, "\pi_3"] \arrow[ur, "\pi_4"'] & \\
& & gr(f) \bowtie gr(g) \arrow[ul, "\pi_5"] \arrow[ur, "\pi_6"'] \arrow[d, "Pr_{X,Z}"', "{[surj]}"] & & \\
& & gr(f;g) \arrow[uuull, bend left] \arrow[uuurr, bend right] \arrow[d, phantom, "{[ji]}"] & & \\
& & & &
\end{tikzcd}$$

(b) Its representation in a single-valued environment

Figure 5.9: Representation of composition of multi-valued functions in a single-valued environment

step, we use the pullback construction $gr(f) \bowtie gr(g) = \{(x, y, z) \mid (x, y) \in gr(f) \wedge (y, z) \in gr(g)\}$. In the second step, we use surjective-jointly-injective factorisation $gr(f; g) = Pr_{X,Z}(gr(f) \bowtie gr(g)) = \{(x, z) \mid \exists y \in Y : (x, y) \in gr(f) \wedge (y, z) \in gr(g)\}$ (see Fig. 5.9b). The predicate `[rcomp]` in the signature Θ_2 (see Table 4.2) can be seen as an abbreviation of the pullback construction followed by the surjective-jointly-injective factorisation.

$$\begin{tikzcd} X \arrow[r, bend left, "f"] \arrow[r, bend right, "g"'] \arrow[r, phantom, "{[\sqsubseteq]}"] & Y \end{tikzcd}$$

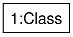
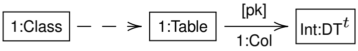
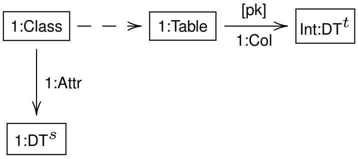
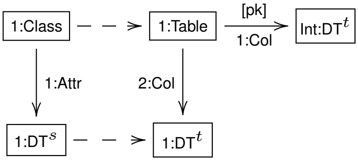
(a) Image-inclusion in multi-valued environment

$$\begin{tikzcd} & gr(f) \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] \arrow[dd, "inj"] & \\ X \arrow[r, phantom, "{[comp]}"] & & Y \arrow[l, phantom, "{[comp]}"'] \\ & gr(g) \arrow[ul, "\pi_3"] \arrow[ur, "\pi_4"'] & \end{tikzcd}$$

(b) Its representation in a single-valued environment

Figure 5.10: Representation of image-inclusion on multi-valued functions in a single-valued environment

Table 5.2: Rules r_1 and r_2 for the transformation of structural object-oriented models to relational data models

$\mathfrak{L}\rhd J_2$	$\mathfrak{R}\rhd J_2$
Rule r_1. Class to table	
	
Rule r_2. Attribute to column	
	

Now we consider image-inclusion. For the multi-valued version, if we have $f : X \to Y$ and $g : X \to Y$, then $f \sqsubseteq g$ means $\forall x \in X : f(x) \subseteq g(x)$. In a single-valued environment, this will be represented as the function $inj : gr(f) \to gr(g)$ together with the predicates `[injective]` and `[composition]`, with the meaning that $(x, y) \in gr(f)$ implies $(x, y) \in gr(g)$ (see Fig. 5.10). Note that for readability reasons the arrows f and g are omitted in Fig 5.10b.

Example 20 (Definition of Transformation Rules) Building upon Example 19, Tables 5.2 and 5.3 shows some of the transformation rules which are needed to transform structural object-oriented models to relational data models. Moreover, Table 5.4 shows some transformation rules which enable transformation of more complex constraints such as requirement 5 in Example 10. These rules are typed by the joined metamodel $\mathfrak{J}_2$ shown in Fig. 5.7 (see also Appendix A.2 for a longer list of transformation rules).

In rule r_1, each class is transformed to a corresponding table. In rule r_2, for each attribute a column is created. The rules r_3 and r_4 are used to transform bidirectional references (or a pair of inverse functions) between two classes to foreign keys between two tables. Notice that the difference between the input patterns of the rules r_3 and r_4 is the constraints forced by the predicates [injective] and [mult(0,1)] on the arrows 1:Ref and 2:Ref, respectively. This constraint affects the way in which a match of the input pattern is transformed to a match of the output pattern. More precisely, since in r_3 each 2:Class is related to at most *one* 1:Class, a foreign key column 3:Col will be created which will refer to 1:Col. However, in r_4 each 2:Class may be related to many 1:Class and vice versa. Therefore, a link table 3:Table is created with two foreign key columns 3:Col and 4:Col.

As seen from Table 5.3, applying rule r_5 will have the same effect as applying rule r_4. The difference between the input patterns is that in r_4 both classes 1:Class and 2:Class have access to each other, while in r_5 only 1:Class "knows about" (or has access to) 2:Class. In contrast, in the output patterns of both rules, the link table 3:Table has access to both of the corresponding tables 1:Table and 2:Table.

The rule r_6 resembles r_3 and r_4, however, the main difference is the constraints which are forced by the predicates [surjective] and [mult(1,∞)] on the arrows 1:Ref and 2:Ref, respectively. According to these constraints, each 2:Class must be related to at least *one* 1:Class. This is reflected in the output pattern by the predicate [image-equal] on the arrows 2:Col and 4:Col.

In rule r_7, the constraints which are forced by the predicates [composition] and [image-inclusion] on the arrows 1;2:Ref and 3:Ref are mapped to [rcomp], [injective] and [total]. More precisely, the link tables 4:Table, 5:Table and 6:Table with their [jointlyinjective] and [foreign-key] predicates correspond to the references 1:Ref, 2:Ref and 3:Ref, respectively, according to rule r_5. Moreover, the join table 7:Table and its [rcomp] predicate correspond to the reference 1;2:Ref and the [composition] predicate. Furthermore, the predicates [total] and [injective] on the arrow from 6:Table to 7:Table correspond to the [image-inclusion] predicate on the references 3:Ref and 1;2:Ref. ♢

Table 5.3: Rules r_3, r_4, r_5 and r_6 for the transformation of structural object-oriented models to relational data models

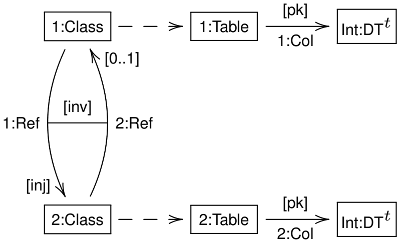
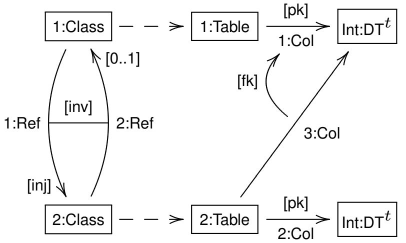
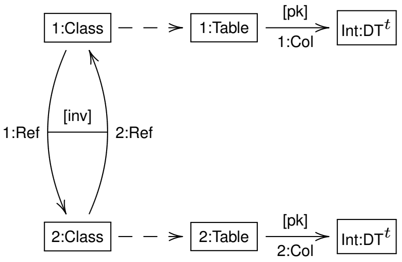
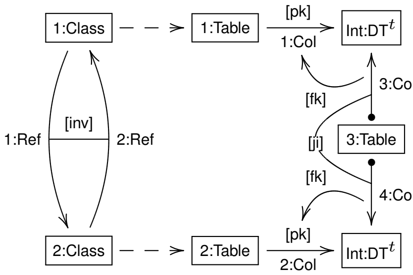
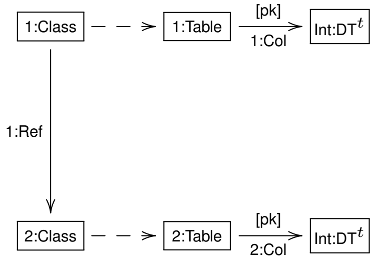
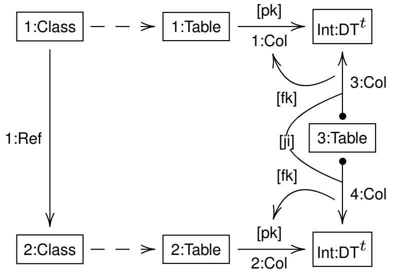
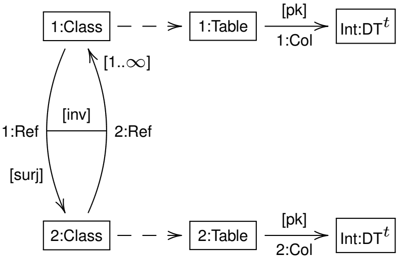
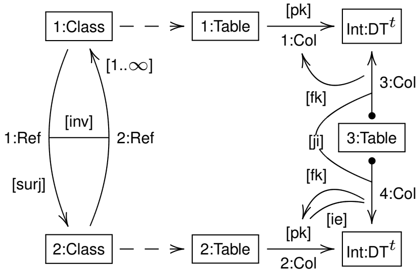
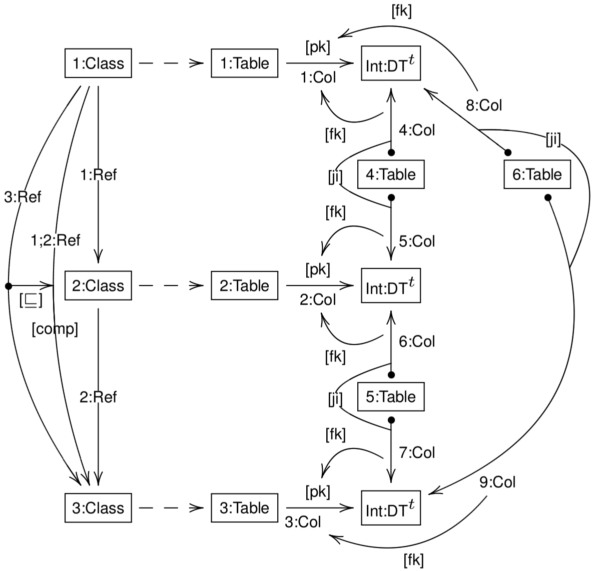
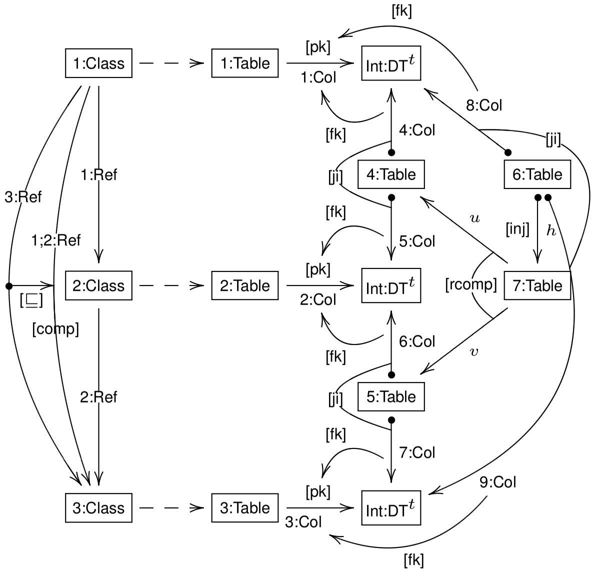
$\mathfrak{L} \triangleright J_2$	$\mathfrak{R} \triangleright J_2$
Rule r_3. Many-to-one references to foreign key	
 1:Class 1:Table [pk] 1:Col Int:DT^t [0..1] [inv] 1:Ref 2:Ref [inj] 2:Class 2:Table [pk] 2:Col Int:DT^t	 1:Class 1:Table [pk] 1:Col Int:DT^t [0..1] [fk] [inv] 1:Ref 2:Ref 3:Col [inj] 2:Class 2:Table [pk] 2:Col Int:DT^t
Rule r_4. Many-to-many references to link table and foreign keys	
 1:Class 1:Table [pk] 1:Col Int:DT^t [inv] 1:Ref 2:Ref 2:Class 2:Table [pk] 2:Col Int:DT^t	 1:Class 1:Table [pk] 1:Col Int:DT^t 3:Col [fk] [inv] 1:Ref 2:Ref [ji] 3:Table [fk] 4:Col [pk] 2:Class 2:Table 2:Col Int:DT^t
Rule r_5. Many-to-many reference to link table and foreign keys	
 1:Class 1:Table [pk] 1:Col Int:DT^t 1:Ref 2:Class 2:Table [pk] 2:Col Int:DT^t	 1:Class 1:Table [pk] 1:Col Int:DT^t 3:Col [fk] 1:Ref [ji] 3:Table [fk] 4:Col [pk] 2:Class 2:Table 2:Col Int:DT^t
Rule r_6. `[inverse]` and `[surjective]` to `[foreign-key]`, `[image-equal]`, `[total]` and `[jointly-injective]`	
 1:Class 1:Table [pk] 1:Col Int:DT^t [1..∞] [inv] 1:Ref 2:Ref [surj] 2:Class 2:Table [pk] 2:Col Int:DT^t	 1:Class 1:Table [pk] 1:Col Int:DT^t [1..∞] 3:Col [fk] [inv] 1:Ref 2:Ref [ji] 3:Table [surj] [fk] 4:Col [ie] [pk] 2:Class 2:Table 2:Col Int:DT^t

Table 5.4: Rule r_7 for the transformation of structural object-oriented models to relational data models

<table>
<tr><th>$\mathfrak{L} \triangleright J_2$</th><th>$\mathfrak{R} \triangleright J_2$</th></tr>
<tr><td colspan="2">Rule r_7. <code>[composition]</code> and <code>[image-inclusion]</code> to <code>[rcomp]</code>, <code>[injective]</code> and <code>[total]</code></td></tr>
<tr><td>
</td><td>
</td></tr>
</table>

5.5 Application of Model Transformation

In this section, we discuss the application task of model transformation. That is, for the following source, target and joined modelling formalisms as well as the morphisms between them,

$$(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3) \xrightarrow{(\sigma_2,\phi_2,\sigma_3)} (\Gamma_2 \rhd J_2, \mathfrak{J}_2, \Gamma_3) \xleftarrow{(\psi_2,\mu_2,\psi_3)} (\Theta_2 \rhd T_2, \mathfrak{T}_2, \Theta_3)$$

we outline the procedure for transforming a source specification $\mathfrak{S}_1 \blacktriangleright \mathfrak{S}_2$ to a target model $\mathfrak{T}_1 \blacktriangleright \mathfrak{T}_2$. We explain the procedure later by applying it to our running example.

1. *Conversion of the source specification.* The source specification $\mathfrak{S}_1 \blacktriangleright \mathfrak{S}_2$ is converted to an intermediate specification $\mathfrak{J}_1 \rhd J_2$. This conversion is given by the conversion functor according to Proposition 7 which leads to $\mathfrak{J}_1 \rhd J_2 = (\sigma_2, \phi_2)^*(\mathfrak{S}_1 \rhd S_2)$.
2. *Iterative application of the transformation rules.* Upon the application of a rule $r : \mathfrak{L} \rhd J_2 \hookrightarrow \mathfrak{R} \rhd J_2$, for a match of the input pattern $\mathfrak{L} \rhd J_2$ in $\mathfrak{J}_1 \rhd J_2$, the specification $\mathfrak{J}_1 \rhd J_2$ will be extended by an appropriate copy of the new elements in $\mathfrak{R} \rhd J_2$, i.e. by those elements in $\mathfrak{R} \rhd J_2$ that are not already in $\mathfrak{L} \rhd J_2$. This step is repeated as long as there are rules which are applicable and the intermediate specification is not conformant to the joined metamodel.
3. *Obtaining the target model.* Once a conformant specification $\mathfrak{J}'_1 \blacktriangleright \mathfrak{J}_2$ is constructed and there are no more applicable transformation rules, the projection functor (see Proposition 8) ensures that we can construct a specification $\mathfrak{T}_1 \blacktriangleright \mathfrak{T}_2 = (\psi_2, \mu_2, \psi_3)^\bullet(\mathfrak{J}'_1 \blacktriangleright \mathfrak{J}_2)$ which can be considered the target model.

Remark 25 (Cases During Rule Application) *The transformation rules are applied iteratively to the intermediate specification $\mathfrak{J}_1 \rhd J_2$. While applying these rules, depending on whether there are still rules which are applicable and whether the intermediate specification is conformant to the joined metamodel, we encounter one of the following states:*

- *There are still rules which are applicable, and the intermediate specification is not conformant to the joined metamodel. In this case, we continue applying the rules.*
- *No more rules are applicable and the intermediate specification is conformant to the joined metamodel. This is the desired case in which we stop applying the rules and project out the target model.*
- *No more rules are applicable, but the constructed specification is not conformant to the joined metamodel. This may mean that the rules are not complete; i.e. the rules do not cover all possible cases in the source models. Alternatively, this may mean that the joined metamodel is not satisfiable.*
- *There are still rules which are applicable, but the specification is already conformant. This may mean that the joined metamodel is underspecified, or loosely specified.*

- *There are always some rules which are applicable. This may mean that the rules are non-terminating, e.g. because the NACs are not defined properly (see below).*

FUNCTIONAL BEHAVIOUR

Model transformations need to possess certain properties in order to be useful. Among these properties is *functional behaviour* [41]. If there are no restrictions on the application of rules, functional behaviour can be achieved if the set of transformation rules is *terminating* and *locally confluent*. The former means that we always come to a point where there are no more rules applicable. The latter means that if two different transformation rules are applied to the same source specification, the two results can be transformed further, leading finally to isomorphic specifications. Another way to achieve functional behaviour is to control the application of rules in an appropriate way. In this section, we shortly outline some strategies or mechanisms for controlling the application of constraint-aware transformation rules in order to achieve functional behaviour. These strategies consist of *negative application conditions* (NAC) and *layering of rules* [41].

NEGATIVE APPLICATION CONDITION

In general, an arbitrary set of transformation rules may be non-terminating. However, if each rule deletes one or more model elements in each application via a match, then the rule will be applicable only once via that particular match, leading to termination. Since our rules are non-deleting rules, we need to require that the output pattern of each rule defines always a NAC for the rule itself. This is to force that a rule is applied only once via a certain match.

Definition 38 (Negative Application Condition) *Given a modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ and a transformation rule $r : \mathfrak{L} \rhd S_2 \hookrightarrow \mathfrak{R} \rhd S_2$, a negative application condition for r is a typed specification morphism $n : \mathfrak{L} \rhd S_2 \to \mathfrak{N} \rhd S_2$.*

Each transformation rule r may be accompanied with a set $NAC(r)$ of NACs. Based on the existence of a match for the input pattern of the rule, and the none-existence of matches for the rule's NACs, the application of the rule can be controlled as defined in the following.

Definition 39 (Application of Transformation Rules with NACs) *Let $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ be a modelling formalism, $\mathfrak{S}_1 \rhd S_2$ a typed specification, and $r : \mathfrak{L} \rhd S_2 \hookrightarrow \mathfrak{R} \rhd S_2$ a transformation rule with a set of negative application conditions $NAC(r) = \{n_i : \mathfrak{L} \rhd S_2 \to \mathfrak{N}_i \rhd S_2\}$ with $i \in I$ for some index set I. The rule r is applicable via a match $m : \mathfrak{L} \rhd S_2 \to \mathfrak{S}_1 \rhd S_2$ if there does not exist any injective match $q_i : \mathfrak{N}_i \rhd S_2 \to \mathfrak{S}_1 \rhd S_2$ such that $m = n_i ; q_i$.*

$$\begin{tikzcd}
\mathfrak{N}_i \rhd S_2 \arrow[dr, "q_i"', "\neq"] & \mathfrak{L} \rhd S_2 \arrow[l, "n_i"'] \arrow[r, hook, "r"] \arrow[d, "m"] & \mathfrak{R} \rhd S_2 \\
 & \mathfrak{S}_1 \rhd S_2 &
\end{tikzcd}$$

Remark 26 (NACs and NUCs) *Negative application conditions are related to negative universal constraints from Definition 30. In particular, each negative universal constraint may be understood as a negative application condition in the sense that, given a transformation rule $r : \mathfrak{L}\rhd S_2 \hookrightarrow \mathfrak{R}\rhd S_2$, a NAC $n : \mathfrak{L}\rhd S_2 \to \mathfrak{N}\rhd S_2$ in $NAC(r)$, and a specification $\mathfrak{S}_1\rhd S_2$, if there is a match $m : \mathfrak{L}\rhd S_2 \to \mathfrak{S}_1\rhd S_2$, then there should not exist a match $q : \mathfrak{N}\rhd S_2 \to \mathfrak{S}_1\rhd S_2$ such that $n; q = m$.*

There are two kinds of non-determinism during application of transformation rules. Firstly, there may be more than one rule which is applicable at the same time. Secondly, for a given rule, there may be more than one match in the source model. A strategy which can be used to achieve some degree of determinism – which together with the termination strategy will lead to functional behaviour – is layering of the rules. In this strategy, one defines a set of numbered layers and assigns each rule to a layer based on the order of its application. In this way, the rules at each layer are applied before applying rules from the layers below. Through the combination of layering and NACs, we will obtain a hierarchy of rules as illustrated in Example 21.

LAYERING RULES

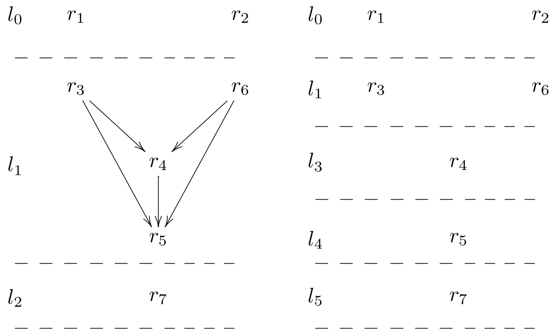
Example 21 (Controlling Rule Application) Recall Example 20. We may use the following NACs and a layering of the rules:

- l_0 : Rules r_1, r_2.
- l_1 : Rules r_3, r_4, r_5, r_6. The input patterns of the rules r_3, r_6 are NACs for the rules r_4, r_5; while the input pattern of the rule r_4 is a NAC for rule r_5.
- l_2 : Rule r_7.

This means that the rules r_1, r_2 will be applied as long as possible. Then, the rules r_3, r_6 are applied as long as possible. Afterwards, the rule r_4, followed by r_5 and r_7. Fig. 5.11a shows the hierarchy which is induced by the strategy above. In the figure, an arrow $r_x \to r_y$ between two rules r_x, r_y denotes that the input pattern of r_x is a NAC for r_y. $\diamondsuit$

Remark 27 (Guarantee of Functional Behaviour) *Since the rules in Example 20 are not exhaustive, i.e. they do not consider every possible case, we will not prove that in general the strategy in Example 21 will guarantee functional behaviour. However, the strategy holds for the object-oriented specification in Fig. 4.7c, which is used as a starting point for the transformation. Note also that this configuration of layering and NACs shown in the example is not unique; that is, one may define other layers and NACs to achieve functional behaviour (see for example Fig. 5.11b).*

Example 22 (Sample Model Transformation) Recall the source $(\Sigma_2\rhd S_2, \mathfrak{S}_2, \Sigma_3)$, target $(\Theta_2\rhd T_2, \mathfrak{T}_2, \Theta_3)$ and joined $(\Gamma_2\rhd J_2, \mathfrak{J}_2, \Gamma_3)$ modelling formalisms introduced in Sections 4.3.1, 4.3.2 and 5.3, respectively. In this example, we use our transformation procedure to apply the model transformation which consists of the rules from Tables 5.2, 5.3 and 5.4 to the source model $((S_1, C^{\mathfrak{S}_1}\!:\Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$ in Fig. 4.7c.



(a) Using NACs and layering (b) Using only layering

Figure 5.11: Possible strategies for controlling application of the rules in Tables 5.2, 5.3 and 5.4

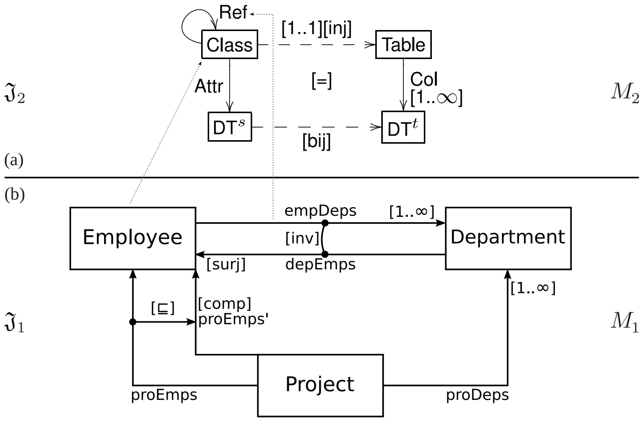


Figure 5.12: The specification $\mathfrak{J}_1 \rhd J_2$ before rule applications

Fig. 5.12b shows the specification $\mathfrak{J}_1 \rhd J_2$ after the conversion step; that is, it shows the first intermediate model before application of the transformation rules. Note that the only difference between $\mathfrak{S}_1 \blacktriangleright \mathfrak{S}_2$ in Fig. 4.7 and $\mathfrak{J}_1 \blacktriangleright \mathfrak{J}_2$ in Fig. 5.12b is that $\mathfrak{S}_1$ is typed by the specification $\mathfrak{S}_2$ while $\mathfrak{J}_1$ is typed by the specification $\mathfrak{J}_2$.

Fig. 5.13 shows an intermediate specification which is created by applying the transformation rules r_1, r_5 and r_6 in Tables 5.2 and 5.3 to the specification in Fig. 5.12b. The effect of applying the rules r_1 and r_5 are hopefully obvious from the figure. The rule r_6 is applied to the arrows empDeps and depEmps, and the predicates [inverse] and [surjective] are transformed to [foreign-key], [image-equal], [total] and [jointly-injective] on the arrows connecting the nodes TEmployee, TEmpDep and TDepartment to Int in Fig. 5.13. The

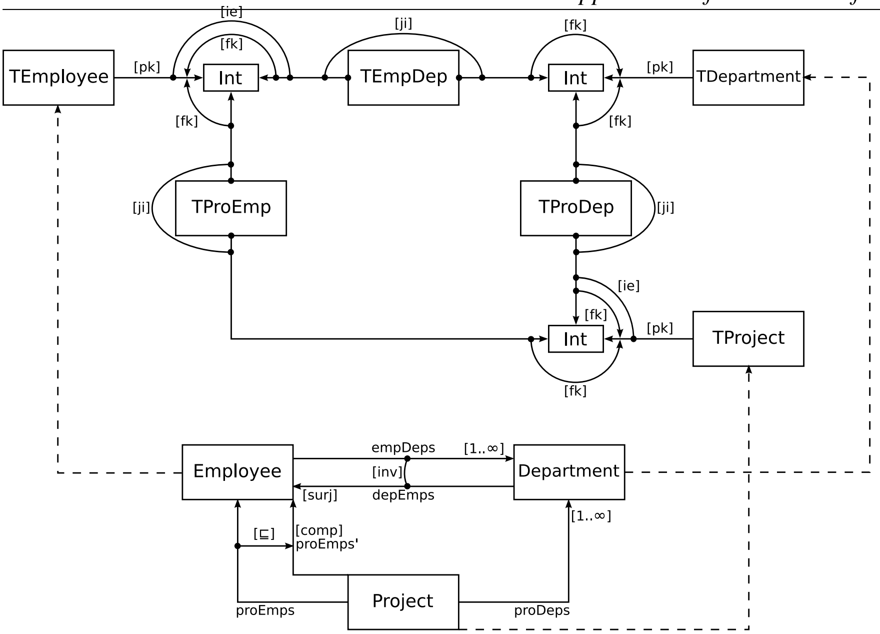


Figure 5.13: The specification $\mathfrak{J}_1 \rhd J_2$ resulting from the application of the transformation rules r_1, r_5 and r_6 from the Tables 5.2 and 5.3

predicate `[image-equal]` is used to force that for any row in the table TEmployee there is a corresponding row in the table TEmpDep.

Fig. 5.14c shows the relational data model, right after the projection step, which is created by applying the transformation rules r_1, r_5, r_6 and r_7. More precisely, this specification is obtained by applying r_1 three times, r_5 three times, r_6 one time, and r_7 one time. Note that the rule r_7 is applied to the arrows proEmps and proEmps', and the predicates `[composition]` and `[image-inclusion]` are transformed to `[rcomp]`, `[injective]` and `[total]` on the arrows connected to the node TProEmp'. Recall that arrows in $(\Theta_2 \rhd T_2, \mathfrak{T}_2, \Theta_3)$ are interpreted as (single-valued) functions (see Section 4.3.2). Hence, we do not need to add constraints to force (single-valued) functions in Fig. 5.14c. However, we use the predicate `[total]` from $\Theta_2 \rhd T_2$ (see Table 4.2) to add constraints which force total functions whenever necessary, for example columns for which a value is required.

$\diamondsuit$

Remark 28 (Queries in Data Models) *For the sake of comparison, we show in Fig. 5.14 a 4-layered metamodelling hierarchy for relational data models. In Fig. 5.14d, we have included the M_0-level which corresponds to the instance in Fig 3.2b. In Fig. 5.14d, we have represented the link tables as tuples. Note that the predicates*

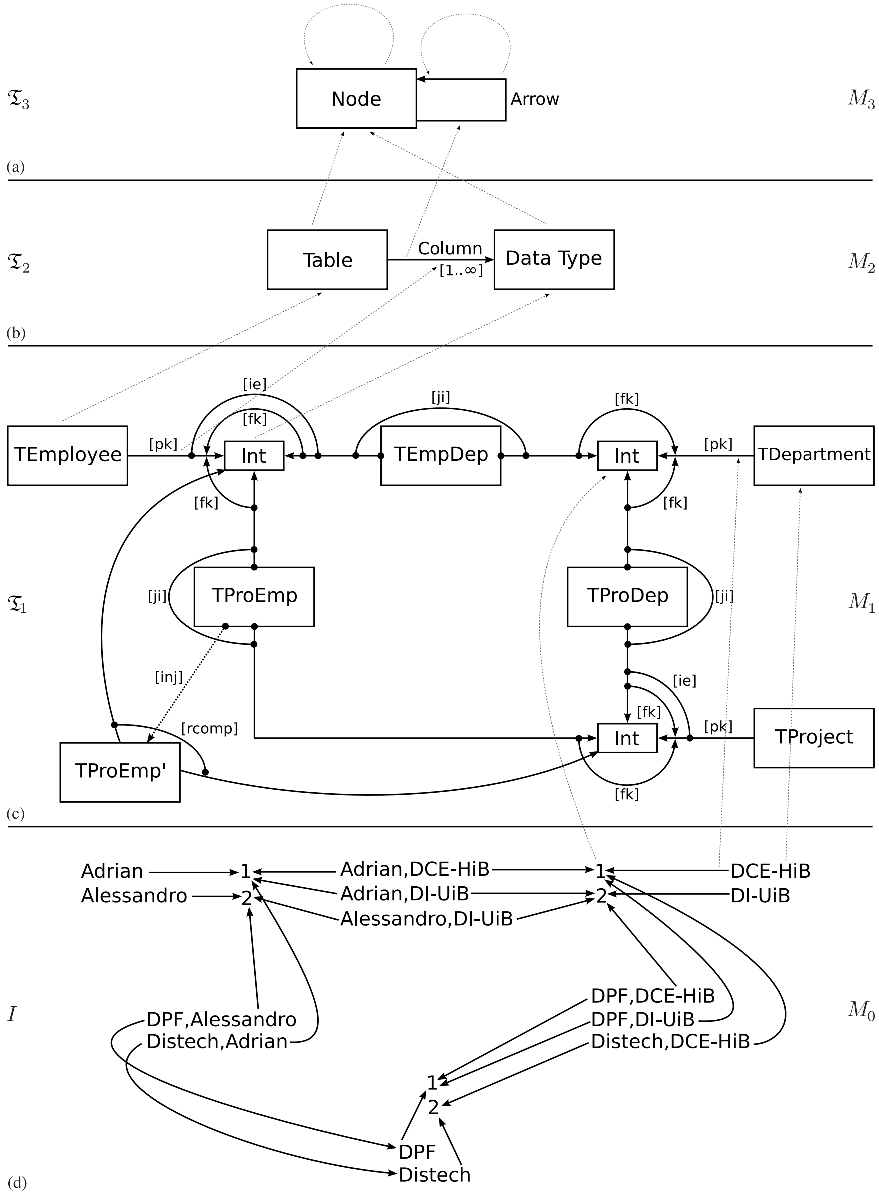


Figure 5.14: The hierarchy of the target modelling formalism $(\Theta_2 \rhd T_2, \mathfrak{T}_2, \Theta_3)$ along with $(\Theta_3, \mathfrak{T}_3, \Theta_3)$; note that the signatures are not shown

`[rcomp]`, `[injective]` *and* `[total]`, *the node* TProEmp'*, and the arrows connected to it in* $\mathfrak{T}_1$*, may be seen as queries or triggers in SQL. That is, in SQL this structure may not be present in the actual database scheme. However, it may be represented as one or another mechanism which will check that the database is in a valid state, i.e. satisfies these constraints, after each update.*

Remark 29 (Cases of Model Transformation) *Recall the classifications of model transformations into homogeneous and heterogeneous on one hand, and into in-place and out-place on the other hand (see Section 1.6). Recall also that these classifications are orthogonal to each other; that is, a homogeneous model transformation may be carried out in-place or out-place, likewise, a heterogeneous model transformation may be carried out in-place or out-place. In this view, we identify the following interesting cases for the joined modelling formalism and its relations to the source and target modelling formalisms:*

$$\begin{tikzcd}
(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3) \arrow[r, "(\sigma_2, \phi_2, \sigma_3)"] & \begin{array}{c} (\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3) \\ \uplus \\ (\Xi_2 \rhd K_2, \mathfrak{K}_2, \Xi_3) \\ \uplus \\ (\Theta_2 \rhd T_2, \mathfrak{T}_2, \Theta_3) \end{array} & (\Theta_2 \rhd T_2, \mathfrak{T}_2, \Theta_3) \arrow[l, "(\mu_2, \psi_2, \mu_3)"']
\end{tikzcd}$$

In case of out-place model transformations, the transformation rules are specified such that no model elements in $\mathfrak{R} \setminus \mathfrak{L}$ *are typed by the source metamodel. That is,* $\iota^{\mathfrak{R} \setminus \mathfrak{L}} \not\subseteq \phi_2(\mathfrak{S}_2)$*. This is necessary to ensure that the transformation will happen out-place, i.e. the original source model is not touched by the model transformation.*

In case of heterogeneous, out-place transformations, i.e. the general case which we have detailed in this thesis, the source and target modelling formalisms are different. In case of homogeneous, out-place model transformations, the source and target modelling formalisms are the same.

In case of in-place model transformations, our approach supports the case where model transformations only extend the source models; i.e. the case where no deleting rules are necessary. Hence heterogeneous, in-place transformations cannot be covered by the our approach since this kind of transformations requires deleting rules.

In case of homogeneous, in-place transformations, the source and target modelling formalisms are the same. Moreover, an appropriate joined modelling formalism may be constructed as follows:

$$\begin{tikzcd}
(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3) \arrow[r, "(\sigma_2, \phi_2, \sigma_3)"] & \begin{array}{c} (\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3) \\ \uplus \\ (\Xi_2 \rhd K_2, \mathfrak{K}_2, \Xi_3) \end{array}
\end{tikzcd}$$

CHAPTER 6

Version Control in MDE

Like other software artefacts, models undergo a complex evolution during their life cycles. Version control is a key technique which enable developers to tackle this complexity. This chapter provides a formalisation of the fundamental concepts and processes of version control in the context of MDE.

6.1 Introduction

The main usage of version control systems (VCSs) consists of enabling users to modify artefacts concurrently, and integrate these modifications as seamless as possible. Thus, a VCS should be able to calculate what has been modified in artefacts at any given time. This feature is called *difference calculation*. In order to integrate these modifications, the VCS should also be able to detect whether two concurrent modifications of the same artefact are in conflict with each other or not. This feature is called *conflict detection*. If conflicts are detected, manual user intervention will be required to resolve the conflicts, otherwise the modifications will be integrated. The integration process is called *merging*.

Traditional VCSs used in software engineering are based on the *copy-modify-merge* approach. This approach is not fully exploited in MDE since current mainstream implementations lack model orientation. In this chapter, we introduce a formalisation of the fundamental concepts of the copy-modify-merge approach based on DPF. The main idea of this approach is to use *common models* to record unmodified model elements and use these common models in difference calculation. These modifications are represented in a *difference model* in which modifications are *annotated* by means of predicates from a signature. This signature is specifically designed for this purpose; that is, it consists of predicates such as `[add]`, `[delete]`, etc. Two concurrent modifications are given by two difference models which can be merged and analysed for conflict detection. The merging process is based on a pushout construction, and (custom) rules are used to detect conflicts.

The usage of signatures for annotation purposes is different from the usage we have discussed in the previous chapters. In Chapter 3, we used predicates from signatures to add constraints to specifications. These predicates had semantic interpretations and affected what could be considered an instance of a specification. In Chapter 5, we used the constraints in declaring model transformation rules and in controlling the application of these rules. In this chapter we use signatures to also annotate modifications in difference models. For example, we may annotate an arrow or a node as added or deleted.

In the remainder of this chapter, all models are diagrammatic specifications as defined in Section 3.3. We assume that these specifications are defined by the object-oriented modelling formalism in Section 4.3.1. However, since the techniques can be used independent on the metamodelling levels, we assume that specifications are untyped and are of the form $(S, C^{\mathfrak{S}}\!:\!\Sigma)$; that is, we drop all subscripts which denote the metamodelling levels.

In this thesis, we only introduce the main concepts and the fundamental processes of the approach which is detailed in [101; 105]. These concepts and processes include:

- calculation of differences
- representation of differences
- merging of differences and conflict detection

We start with an example which explains a copy-modify-merge scenario.

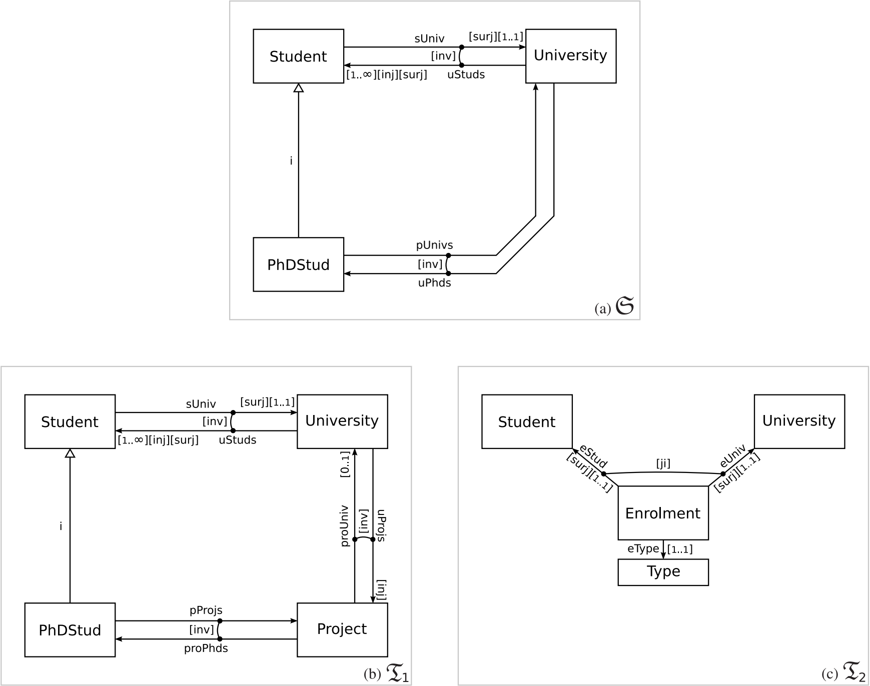
6.2 A Copy-Modify-Merge Scenario in MDE

In this section, we illustrate a usual scenario of concurrent development in MDE, in which an ideal copy-modify-merge VCS is adopted. In this scenario, each developer accesses a repository and creates a personal working copy – a snapshot of the repository's models. Then, the developers modify their working copies simultaneously and independently. Finally, the working copies are merged together in the repository. The VCS assists with the merging by detecting conflicting modifications. When a conflict is detected, the system requires manual intervention by the developer.

Example 23 (A VCS Scenario) This example is kept intentionally simple, retaining only the details which are relevant for our discussion. Suppose that two software developers, Alice and Bob, are working on the development of an information system for universities using a copy-modify-merge VCS. This scenario is depicted in Fig. 6.1.

Alice starts with the model $\mathfrak{S}$ (see Fig. 6.1a). She deletes the arrows `pUnivs` and `uPhds` (see Fig. 6.1b) and adds a node `Project` together with the arrows `pProjs`, `proPhds`, `proUniv` and `uProjs`.

Bob modifies $\mathfrak{S}$ concurrently. He considers other types of students, e.g. Post-doc, teaching assistant, etc; therefore, he deletes the `PhDStud` node and refactors



(a) $\mathfrak{S}$

(b) $\mathfrak{T}_1$

(c) $\mathfrak{T}_2$

Figure 6.1: The models $\mathfrak{S}$, $\mathfrak{T}_1$ and $\mathfrak{T}_2$

the model by adding two nodes, `Enrolment` and `Type`, together with the arrows `eStud`, `eUniv` and `eType` (see Fig. 6.1c).

Merging the modifications done by Alice and Bob will lead to conflicts. This is because Alice has added some arrows to/from the node `PhDStud` which Bob has deleted. The resolution of this conflict requires manual intervention by the developers.

◇

6.3 Calculation and Representation of Differences

VCSs rely on the identification of commonalities between (versions of) artefacts, which is necessary to compute their differences. For example, a solution to the longest common subsequence problem [60] is typically implemented in differencing algorithms for text-based files. In DPF, we introduce a different approach to the identification of common elements. Model elements which are not changed during an evolution step are *recorded* in *common models*; i.e. models which represent

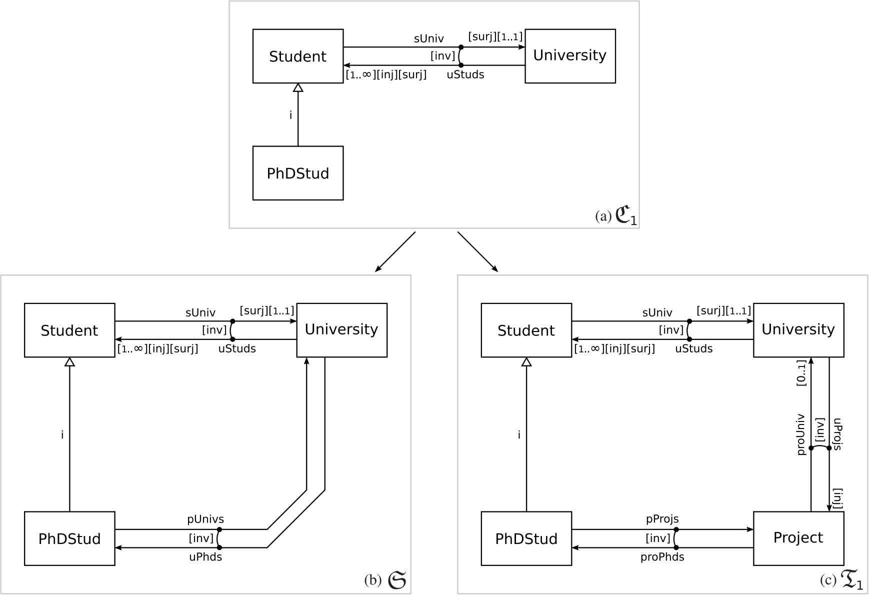


Figure 6.2: The common model $\mathfrak{C}_1$ of the models $\mathfrak{S}$ and $\mathfrak{T}_1$

the commonalities between two subsequent versions of a model. These models are regarded as meta-information about evolution steps.

Example 24 (Common Model) Building on Example 23, Fig. 6.2a shows the common model $\mathfrak{C}_1$ for the models $\mathfrak{S}$ and $\mathfrak{T}_1$.

◇

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 each evolution step. In this thesis, we classify modifications as indicated in Table 6.1. These are just a subset of the modifications presented in [101; 105]. This classification is performed on the level of structural models, meaning that we do not take into account *operations*; i.e. methods or functions. Nor this approach does detect and represent modifications in layout or visualisation since the syntax and the semantics of a structural model is not affected by these changes.

In order to calculate the difference between two subsequent versions of a model, we need to know which modifications have taken place. Identifying these modifications requires analysis of the old and the new versions of the model, together with their common model. For example, all the nodes and arrows which are present in the new model, but not in the common model, are identified as added. Similarly,

Table 6.1: Classification of modifications

Name	**Definition**	**Alternative terms**
add	a node/arrow is added to the underlying graph of a model	*create*, *insert*
delete	a node/arrow is deleted from the underlying graph of a model	*remove*

Table 6.2: The signature Δ

Π_Δ	α_Δ	**Proposed visual.**	**Alternative visual.**
`[add]`^{n}	1	$\boxed{X}^{[A]}$	$\boxed{X}$
`[add]`^{a}	$1 \xrightarrow{x} 2$	$\boxed{X} \xrightarrow{[A]f} \boxed{Y}$	$\boxed{X} \xrightarrow{f} \boxed{Y}$
`[delete]`^{n}	1	$\boxed{X}^{[D]}$	$\boxed{X}$
`[delete]`^{a}	$1 \xrightarrow{x} 2$	$\boxed{X} \xrightarrow{[D]f} \boxed{Y}$	$\boxed{X} \xrightarrow{f} \boxed{Y}$
`[conflict]`^{n}	1	$\boxed{X}^{[C]}$	$\boxed{X}$
`[conflict]`^{a}	$1 \xrightarrow{x} 2$	$\boxed{X} \xrightarrow{[C]f} \boxed{Y}$	$\boxed{X} \xrightarrow{f} \boxed{Y}$

all the nodes and arrows which are present in the old model, but not in the common model, are identified as deleted. This means that in order to calculate the difference we need to distinguish between common elements, elements from the old version and elements from the new version. This capacity is one of the properties of the pushout construction in category **Spec**(Σ) (see Appendix A.1.2). Hence, we adopt pushout construction to calculate differences between models. In particular, pushout constructs a model where all common, added and deleted elements are present at the same time.

The output of this calculation is then presented in a *difference model*. In order to show the modifications, an appropriate language is needed. Due to the graph-based nature of models, the language must be diagrammatic and must make it possible to identify modifications as added and deleted. In DPF, we define a signature Δ for the representation of model differences. The signature $\Delta = (P^\Delta, \alpha^\Delta)$ consists of the predicates `[add]`, `[delete]` and `[conflict]` (see Table 6.2). These predicates are used to present the information "added" and "deleted" locally in the difference model. More precisely, the difference models will be annotated by predicates from Δ in addition to predicates from Σ. The predicates `[add]` and `[delete]` each has two arities: 1 and $1 \xrightarrow{x} 2$. That is, each of these predicates can be used to annotate both nodes and arrows. Note that the predicate `[conflict]` is not used in difference models of two subsequent models; it will be used to annotate conflicting modifications in merge models (see Section 6.4).

$$\begin{tikzcd}
& \mathfrak{C} \arrow[dl, "s"'] \arrow[dr, "t"] & \\
\mathfrak{S} \arrow[dr, "s^*"'] & PO & \mathfrak{T} \arrow[dl, "t^*"] \\
& \mathfrak{P} \arrow[r, hook] & \mathfrak{D}
\end{tikzcd}$$

Figure 6.3: The difference model $\mathfrak{D}$ for the models $\mathfrak{S}$ and $\mathfrak{T}$

Remark 30 (Multiple Visualisations) *We define two visualisations for the Δ predicates. The default visualisation is compatible with black and white printing and the predicates are used to annotate the model elements in difference models. Although this visualisation enables the representation of differences, an alternative visualisation based on colour-coding is proposed. We believe that this colouring technique makes it easier to understand modifications. We have adopted both visualisations so that the examples are intuitive as well as compatible with black and white printing.*

To summarise, given two models $\mathfrak{S}$ and $\mathfrak{T}$ together with their common model $\mathfrak{C}$, we calculate and represent their difference model $\mathfrak{D}$ in two steps: firstly, we construct a pushout $\mathfrak{P}$; secondly, we use a set of rules to add annotations to the resultant pushout object (see Fig. 6.3). The rules are defined in such a way that model elements which are added will be annotated with the predicate `[add]`, and model elements which are deleted will be annotated with the predicate `[delete]`. In more detail, assuming that $\mathfrak{T}$ is newer than $\mathfrak{S}$, all model elements $\mathfrak{T} \setminus t(\mathfrak{C})$ will be labelled as added in $\mathfrak{D}$; and all model elements $\mathfrak{S} \setminus s(\mathfrak{C})$ will be labelled as deleted in $\mathfrak{D}$. We define difference models as follows (see Fig. 6.3):

Definition 40 (Difference Model) *Given a common model $\mathfrak{C}$ of models $\mathfrak{S}$ and $\mathfrak{T}$, the difference model is a specification $\mathfrak{D} = (D, C^{\mathfrak{D}} : \Sigma \cup \Delta)$ which is constructed in the following two steps:*

- *construction of $\mathfrak{P}$ together with the specification morphisms $s^* : \mathfrak{S} \to \mathfrak{P}$ and $t^* : \mathfrak{T} \to \mathfrak{P}$ by pushout in category* $\mathbf{Spec}(\Sigma \cup \Delta)$, *in accordance with Proposition 10*
- *obtaining $\mathfrak{D}$ from $\mathfrak{P}$ by applying the following rules (see Table 6.3):*

$$\begin{array}{rl}
\textit{For each node } X \in (T_0 \setminus t_0(C_0))\textit{:} & (\texttt{[add]}^n, \delta) \in C^{\mathfrak{D}} \\
\textit{where} & \delta(\alpha(\texttt{[add]}^n)) = X \\
\textit{For each arrow } f \in (T_1 \setminus t_1(C_1))\textit{:} & (\texttt{[add]}^a, \delta) \in C^{\mathfrak{D}} \\
\textit{where} & \delta(\alpha(\texttt{[add]}^a)) = f \\
\textit{For each node } X \in (S_0 \setminus s_0(C_0))\textit{:} & (\texttt{[delete]}^n, \delta) \in C^{\mathfrak{D}} \\
\textit{where} & \delta(\alpha(\texttt{[delete]}^n)) = X \\
\textit{For each arrow } f \in (S_1 \setminus s_1(C_1))\textit{:} & (\texttt{[delete]}^a, \delta) \in C^{\mathfrak{D}} \\
\textit{where} & \delta(\alpha(\texttt{[delete]}^a)) = f
\end{array}$$

Table 6.3: Summary of rules for annotating difference model $\mathfrak{D}$

In pushout object $\mathfrak{P}$	In $\mathfrak{D}$
For each node $X \in (T_0 \setminus t_0(C_0))$ $\boxed{\text{X}}$	$\boxed{\text{X}}^{[A]}$
For each arrow $f \in (T_1 \setminus t_1(C_1))$ $\boxed{\text{X}} \xrightarrow{f} \boxed{\text{Y}}$	$\boxed{\text{X}} \xrightarrow{[A] f} \boxed{\text{Y}}$
For each node $X \in (S_0 \setminus s_0(C_0))$ $\boxed{\text{X}}$	$\boxed{\text{X}}^{[D]}$
For each arrow $f \in (S_1 \setminus s_1(C_1))$ $\boxed{\text{X}} \xrightarrow{f} \boxed{\text{Y}}$	$\boxed{\text{X}} \xrightarrow{[D] f} \boxed{\text{Y}}$

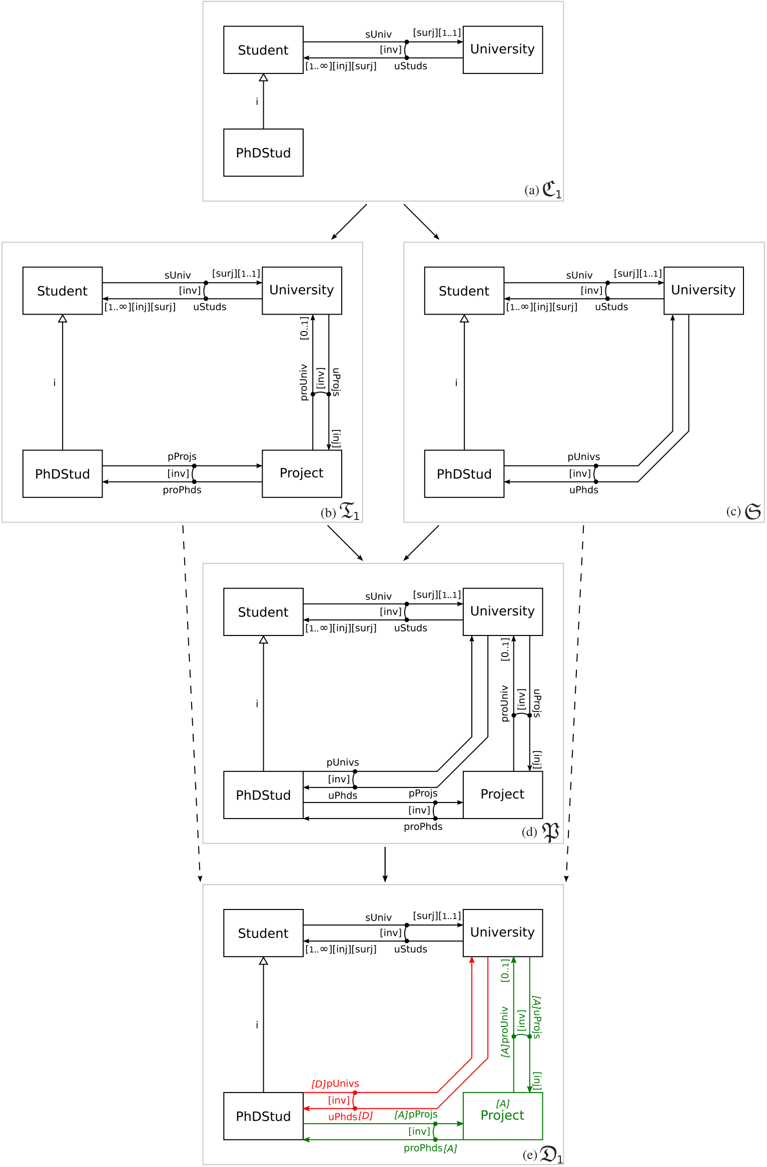
Remark 31 (Annotation of Constraints) *In the definition of difference model, we have that $C^{\mathfrak{D}} := C^{\mathfrak{P}} \cup \{(p, \delta) | p \in P^{\Delta}\}$; however, we annotate only the underlying graph D with predicates from Δ. The possibility to extend the technique in order to also annotate the constraints $C^{\mathfrak{P}}$ is a subject to future work.*

Example 25 (Difference Model) Building on Example 23, Fig. 6.4d shows a pushout object $\mathfrak{P}$ constructed for the models $\mathfrak{C}_1$, $\mathfrak{S}$ and $\mathfrak{T}_1$. Moreover, Fig. 6.4e shows the difference model $\mathfrak{D}_1$ after applying the rules in Table 6.3 to the model in Fig. 6.4d. The node `Project` and the arrows connected to it have been added to the model $\mathfrak{T}_1$. These added elements are annotated as added; i.e. annotated with the predicate `[add]` from Δ in the difference model $\mathfrak{D}$. This predicate is visualised as *`[A]`* in the proposed visualisation, or by green colouring in the alternative visualisation of Δ. Moreover, the arrows `pUnivs` and `uPhds` have been deleted from the model $\mathfrak{S}$. These deleted elements are annotated with the predicate `[delete]` from Δ in the difference model $\mathfrak{D}$. This predicate is visualised as *`[D]`* in the proposed visualisation, or by red colouring in the alternative visualisation of Δ.

◇

6.4 Merging

We have shown the calculation and representation of differences between two models. Now, given two difference models which are calculated based on modifications done to the same model, we want to check whether these modifications can be seamlessly merged together. More precisely, given two difference models $\mathfrak{D}_1$ and $\mathfrak{D}_2$, representing two concurrent modifications of the model $\mathfrak{S}$, we calculate and represent merge of differences in a model $\mathfrak{M}$ in two steps: firstly, we construct a pushout $\mathfrak{I}$; secondly, we apply rules to the resultant pushout object (see Fig. 6.5).



(a) $\mathfrak{C}_1$

(b) $\mathfrak{T}_1$

(c) $\mathfrak{S}$

(d) $\mathfrak{P}$

(e) $\mathfrak{D}_1$

Figure 6.4: An example difference model $\mathfrak{D}_1$ for the models $\mathfrak{S}$ and $\mathfrak{T}_1$

$$\begin{tikzcd}
 & \mathfrak{C}_1 \arrow[dl] \arrow[dr] & & \mathfrak{C}_2 \arrow[dl] \arrow[dr] & \\
\mathfrak{T}_1 \arrow[dr] & & \mathfrak{S} \arrow[dl, "d_1"'] \arrow[dr, "d_2"] & & \mathfrak{T}_2 \arrow[dl] \\
 & \mathfrak{D}_1 \arrow[dr, "d_1^*"'] & PO & \mathfrak{D}_2 \arrow[dl, "d_2^*"] & \\
 & & \mathfrak{I} \arrow[r] & \mathfrak{M} &
\end{tikzcd}$$

Figure 6.5: The merge of differences $\mathfrak{M}$ of the two difference models $\mathfrak{D}_1$ and $\mathfrak{D}_2$

These rules are defined such that model elements which are in conflict according to the conflict detection rules will be annotated with [conflict], and model elements which are annotated with [delete] will be deleted.

By pushout construction, the sets of constraints (and annotations) $C^{\mathfrak{D}_1}$ and $C^{\mathfrak{D}_2}$ are merged together into $C^{\mathfrak{I}}$. While some of the annotations are identified (i.e. determined to be identical) by pushout construction (see Remark 32 in Appendix A.1.2), some model elements may be annotated by two predicates from Δ. Each pair of annotations in $C^{\mathfrak{I}}$ must be analysed to detect conflicting modifications; i.e. to decide on whether adding annotations $(\texttt{[conflict]}, \delta) \in C^{\mathfrak{I}}$ or not. In order to perform this analysis, we will define a set of rules in Definition 41 which are applied to $\mathfrak{I}$. These rules contain all possible combinations of annotations $(p, \delta) \in C^{\mathfrak{I}}$. This is justified as follows:

- It is impossible to annotate the same model element in $\mathfrak{I}$ with [add] twice, or with [add] and [delete] together. This is because we consider added elements to be distinct even if they have the same names, and only elements which are identified in a common model will be treated as identical.
- It is impossible to annotate the same model element in $\mathfrak{I}$ with [delete] twice. This is because they will be identified by the pushout construction.

We define merge of differences as follows (see Fig. 6.6).

Definition 41 (Merge of Differences) *Given a model $\mathfrak{S}$ and two difference models $\mathfrak{D}_1$ and $\mathfrak{D}_2$, the merge of differences is a specification $\mathfrak{M} = (M, C^{\mathfrak{M}} \colon \Sigma \cup \Delta)$ which is constructed in the following two steps:*

- *construction of $\mathfrak{I}$ together with the specification morphisms $d_1^* : \mathfrak{D}_1 \to \mathfrak{I}$ and $d_2^* : \mathfrak{D}_2 \to \mathfrak{I}$ by pushout in category $\textsf{Spec}(\Sigma)$, in accordance with Proposition 10*
- *obtaining $\mathfrak{M}$ from $\mathfrak{I}$ by applying the following rules (see Table 6.4): For each node $X \in I_0$, each arrow $f \in I_1$, and each pair of constraints $(p, \delta; d_1^*), (q, \rho; d_2^*) \in C^{\mathfrak{I}}$ such that:*

Table 6.4: The rules applied to the pushout object $\mathfrak{I}$ in order to obtain merge of differences $\mathfrak{M}$

In pushout object $\mathfrak{I}$	In $\mathfrak{M}$
Source of an added arrow is deleted	
$\boxed{\mathrm{X}} \xleftarrow{[A]f} \boxed{\mathrm{Y}}^{[D]}$	$\boxed{\mathrm{X}} \xleftarrow{[C][A]f} \boxed{\mathrm{Y}}^{[C][D]}$
Target of an added arrow is deleted	
$\boxed{\mathrm{X}} \xrightarrow{[A]f} \boxed{\mathrm{Y}}^{[D]}$	$\boxed{\mathrm{X}} \xrightarrow{[C][A]f} \boxed{\mathrm{Y}}^{[C][D]}$

$$\begin{tikzcd}
 & \mathfrak{S} \arrow[dl, "d_1"'] \arrow[dr, "d_2"] & \\
\mathfrak{D}_1 \arrow[dr, "d_1^*"'] & PO & \mathfrak{D}_2 \arrow[dl, "d_2^*"] \\
 & \mathfrak{I} \arrow[r] & \mathfrak{M}
\end{tikzcd}$$

Figure 6.6: The merge of differences $\mathfrak{M}$

$$\begin{array}{llll}
 & X = (\delta; d_1^*)(\alpha(p)) = & src((\rho; d_2^*)(\alpha(q))) = & src(f) \\
or & X = (\delta; d_1^*)(\alpha(p)) = & trg((\rho; d_2^*)(\alpha(q))) = & trg(f) \\
or & X = (\rho; d_2^*)(\alpha(q)) = & src((\delta; d_1^*)(\alpha(p))) = & src(f) \\
or & X = (\rho; d_2^*)(\alpha(q)) = & trg((\delta; d_1^*)(\alpha(p))) = & trg(f)
\end{array}$$

we have

$$\begin{array}{lll}
1. & (\texttt{[add]}^a, \delta; d_1^*), (\texttt{[delete]}^n, \rho; d_2^*) & \in C^{\mathfrak{I}} \\
implies & (\texttt{[add]}^a, \delta; d_1^*), (\texttt{[delete]}^n, \rho; d_2^*), & \\
 & (\texttt{[conflict]}^a, \delta; d_1^*), (\texttt{[conflict]}^n, \rho; d_2^*) & \in C^{\mathfrak{M}} \\
 & & \\
2. & (\texttt{[delete]}^n, \delta; d_1^*), (\texttt{[add]}^a, \rho; d_2^*) & \in C^{\mathfrak{I}} \\
implies & (\texttt{[delete]}^n, \delta; d_1^*), (\texttt{[add]}^a, \rho; d_2^*), & \\
 & (\texttt{[conflict]}^n, \delta; d_1^*), (\texttt{[conflict]}^a, \rho; d_2^*) & \in C^{\mathfrak{M}}
\end{array}$$

Should the merge of differences $\mathfrak{M}$ contain a constraint $(\texttt{[conflict]}, \delta)$, it is said to be in a state of conflict which has to be resolved manually by the developer. Although conflicts are context-dependent, we have recognised one situation in which syntactic conflicts will arise: the concurrent modification in which arrows are added from/to a node which has been deleted. This is just a subset of the conflicts presented in [101; 105]. In addition, other rules for the detection of custom conflicting modifications can be defined.

However, if there are no conflicts, a synchronised model will be created which includes modifications from both $\mathfrak{D}_1$ and $\mathfrak{D}_2$. Creation of this model is done by deleting the nodes and arrows from $\mathfrak{M}$ which are annotated with the predicate `[delete]`. In addition, the remaining annotations (representing added elements) will be deleted, yielding a specification which has only constraints from the signature Σ.

Example 26 Merge of Differences Building on Example 23, Fig. 6.7c shows the pushout object constructed for the difference models $\mathfrak{D}_1$ and $\mathfrak{D}_2$, while Fig. 6.7d shows the merge of differences $\mathfrak{M}$ after applying the rules from Definition 41. After applying the rules and detecting the conflicts, the merging process stops and manual intervention by the users will be required to resolve the conflicts. That is, the structure in Fig. 6.7e will not be created since it contains dangling edges.

Note that the node `PhDStud` and the arrows `pProjs` and `proPhds` are annotated with `[delete]`, `[add]` and `[add]`, respectively. In $\mathfrak{M}$ these nodes and arrows are additionally annotated with `[conflict]` in accordance with the rules in Definition 41. Moreover, the arrows `pUnivs` and `uPhds` are annotated with the predicate `[delete]` in both $\mathfrak{D}_1$ and $\mathfrak{D}_2$, however, since the two corresponding annotations are identified by pushout construction, there is only one `[delete]` on these arrows in the pushout object in Fig. 6.7c.

$\diamondsuit$

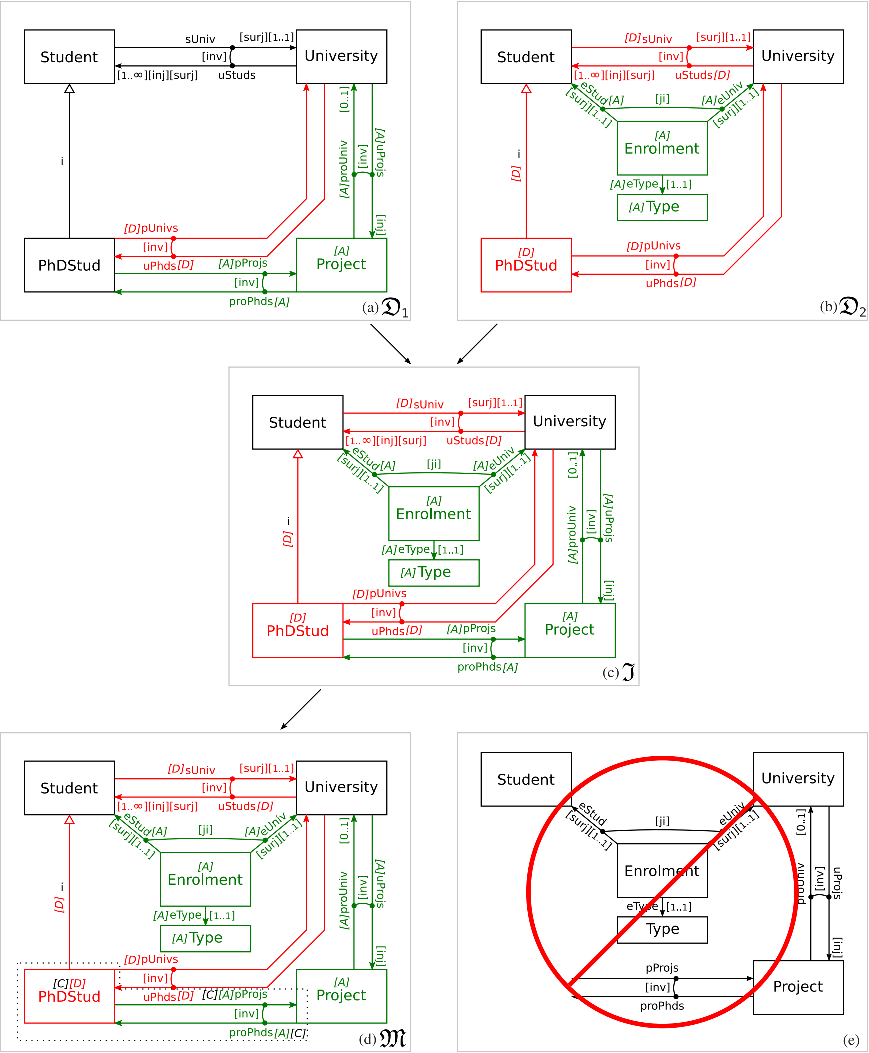


Figure 6.7: An example of merge of differences $\mathfrak{M}$

CHAPTER 7

Discussion

This chapter presents approaches and techniques which we consider relevant to the approach detailed in this thesis. The chapter presents also some open issues and further work regarding the current state of DPF. Finally, a summary of the main points of the thesis along with some concluding remarks are given.

7.1 Related Work

This thesis has dealt with a formalisation approach to (meta)modelling, model transformation and version control of models. Other approaches, techniques and tools related to the topic of this thesis are presented in this section.

7.1.1 (META)MODELLING

Lately, diagrammatic modelling and formalisation of metamodelling have been extensively discussed in the literature. Here we present some of the approaches to the specification of models, metamodels and constraints, as well as the formalisation of MOF-based modelling languages.

ATTRIBUTED GRAPHS

Attributed graphs are often used in modelling, metamodelling and model transformation [40; 41]. For example, the well-known graph transformation framework Attributed Graph Grammar (AGG) uses attributed graphs to represent (meta)models. An attributed graph is a graph with node and edge attribution; that is, it consists of a graph together with a set of attribute nodes, e.g. `String`, `Integer`, `"Person"`, `123`, etc. Assigning attributes to the nodes of the graph is done by creating edges from the nodes to attribute nodes. The assignment of edge attributes can be done in the same way, however, in this case the graph must allow edges from edges to attribute nodes. This kind of graph is called E-graph in [40; 41]. In attributed graphs, node and edge attributions are used to describe properties of nodes and edges. The typical visualisation of node and edge attributions is quite similar to UML models.

This, along with its formal underpinning, have facilitated the adoption of attributed graphs both in practically oriented and theoretically oriented software modelling and model transformation frameworks.

ALGEBRAIC SPECIFICATIONS

The work in [21; 99] uses algebraic specification to give formal semantics to MOF. In [21] MOF-based metamodels are considered to be graphs with attributed objects as nodes and references as arrows. These graphs are represented by specifications in Membership Equational Logic (MEL), i.e. the logical power of MEL is essentially used to define the concept of a finite (multi)set at different places and levels. This formal semantics is made executable by using the Maude language [122], which directly supports MEL specifications. In this formalisation, metamodels are seen to play several roles: as data, as type or as theory; these roles are formally expressed by metamodel definition, model type, and metamodel realisation, respectively. In [99] a metamodelling framework, which is also based on Maude, is presented. In this framework, graphs are represented as terms by taking advantage of the underlying term matching algorithm modulo associativity and commutativity. The ability of Maude to execute the specifications allows the implementation of some key operations on models, such as model subtyping, type inference, and metric evaluation.

CONSTRUCTIVE TYPE THEORY

The work in [96] exploits the higher-order nature of constructive type theory to uniformly treat the syntax of models, metamodels as well as the MOF model itself. Models are formalised as terms (token models) and can also be represented as types (type models) by means of a reflection mechanism. This formalisation ensures that correct typing corresponds to provably correct models and metamodels.

EASIK

Easik (Entity Attribute Sketch Implementation Kit) is a Java based graphical environment for database design, database implementation and data manipulation [63; 121]. Easik implements the Sketch Data Model (SkDM) as an enhancement of the Entity-Relationship Attribute (ERA) design paradigm. The SkDM is a data model related to the ERA model, using the category-theoretic concept of Sketch [63]. The enhancement provides simple and precise expressiveness of constraints, which automatically enforces constraints in models. The visual design of Easik is cleaner since there are fewer graphical element types than in ERA.

The interface of Easik enables users to create, in a diagrammatic fashion, a database design of entities, attributes and constraints. The design can be exported to a database schema in SQL that enforces the diagrammatic constraints. Easik supports connection to some common database management systems such as PostgreSQL and MySQL. With a connection available, it enables data entry and data manipulation via its visual interface. In addition, an overview canvass allows several databases to be edited simultaneously, and provides a simple mechanism for views. Furthermore, Easik provides some reasoning about dependencies or relations between constraints.

EPSILON

Epsilon (Extensible Platform of Integrated Languages for mOdel maNagement) is a family of consistent and interoperable task-specific programming languages which can be used to interact with EMF models [44]. In addition to the core language Epsilon Object Language (EOL) – an imperative language that combines the procedural style of Javascript with OCL's querying capabilities – Epsilon provides several task specific languages for performing code generation, model transformation, model validation, model comparison, etc. One of these task specific languages is 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.

In addition to the languages above, Epsilon provides a mature tool support for management of Ecore-based models. For example, EuGENia: a front-end for Graphical Modeling Framework (GMF) [51]; Exeed: an enhanced version of the built-in EMF reflective tree-based editor; ModeLink: an editor consisting of side-by-side EMF tree-based editors which is convenient for establishing (weaving) links between models; etc.

VISUAL OCL

Visual OCL (VOCL) [125] is an effort to define a graphical visualisation for OCL. It extends the syntax of UML and is used to define constraints on UML models. VOCL allows developers to put models together with constraints without leaving the graphical level of abstraction. VOCL does not extend the formal semantics of OCL. It only visualises constraints which are specifiable in OCL.

ALLOY

Alloy [5] is a structural modelling language which is capable of expressing complex structural constraints and behaviour. 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. Then for a given logical formula F, Alloy attempts to find a model which satisfies F. Alloy models are checked by using the Alloy analyser which attempts to find counterexamples within a limited scope which violates 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.

FUJABA

The Fujaba (From UML to Java And Back Again) Tool Suite is an open source CASE tool supporting MDE and re-engineering [48]. It provides a formal, graphical, object-oriented software system specification language consisting of UML class diagrams and story diagrams. Story diagrams are specialised activity diagrams based on graph transformations that facilitate the specification of complex application-specific object structures [75]. Fujaba facilitates Java code generation based on a formal specification of systems structure and behaviour. In addition, Fujaba is an extensible tool framework supporting plugin development. Several Fujaba plug-ins are available providing support for, among others, modelling and metamodelling with MOF; model transformations specified by TGGs; as well as reverse engineering of source code through creating UML class diagrams, detecting design patterns, idioms, anti patterns, etc.

7.1.2 Model Transformation

Many of the model transformation approaches of today are based on graph transformations due to the graph-based nature of models. In this section, an outline of these approaches and other relevant approaches is given. Since the transformation approach detailed in this thesis is also closely related to graph transformations, this section starts with an introduction of the main concepts of graph transformations [40; 41; 58; 103].

GRAPH TRANSFORMATION

Graph transformation is used as the formal foundation for several model transformation approaches and tools, e.g. PROGRES [97] AGG [119], VIATRA2 [124], etc. The notion of graph transformation comprises the concepts of graph grammars and graph rewriting, and its main idea is rule-based modification of graphs. In graph transformation, models are typically represented by typed (attributed) graphs and metamodels by (attributed) type graphs. Moreover, the conformance relation between models and metamodels is given by typing graph homomorphisms.

One of the main approaches of graph grammar and graph transformation is the algebraic approach, which is based on pushout constructions. This approach may be further divided into *double pushout* (DPO) and *single pushout* (SPO).

DOUBLE PUSHOUT

In the DPO approach, a graph production has the form $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, where L and R are the LHS and the RHS graphs, respectively, K is the common interface (graph) of L and R, and l, r are injective graph homomorphisms. Moreover, given a host (or source) graph G, a direct graph transformation $G \overset{p,m}{\Longrightarrow} H$ with a production p and a match $m : L \rightarrow G$ is given by two pushout constructions in the category **Graph**

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "PO_1"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "PO_2"] & R \arrow[d, "n"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

By constructing a pushout complement PO_1, all elements of $L \setminus K$ which are matched by m will be deleted from G. The remaining structure of G; i.e. after deleting $L \setminus K$ from G, will result in the context graph D. By a second pushout construction PO_2, all elements of $R \setminus K$ will be *glued* together with D to obtain H. That is, R and D are glued together via K, written $R +_K D$, to obtain H.

Since a direct graph transformation $G \overset{p,m}{\Longrightarrow} H$ may lead to dangling edges in H, a *gluing condition* has to be satisfied in the first step (PO_1). The gluing condition requires that $L +_K D = G$; i.e. no dangling edges exist in D. If the gluing condition is not satisfied, p is identified as not applicable via the match m.

SINGLE PUSHOUT

In the SPO approach, a graph production p may be interpreted as a *partial* graph morphism. In this case, a direct graph transformation $G \overset{p,m}{\Longrightarrow} H$ will be given by a pushout in the category **PGraph**, which is the category of graphs and partial graph morphisms. Taking pushout in **PGraph** means to delete dangling edges in H.

NEGATIVE APPLICATION CONDITION

Independent on which of the approaches is used, a direct graph transformation $G \overset{p,m}{\Longrightarrow} H$ represents a one-step transformation from G to H. This step is carried out if p is *applicable*. In addition to the *applicability condition*, sometimes it is desirable to specify the cases in which a direct graph transformation *should not* be applied. This feature is represented by *negative application conditions* (NAC).

Given a production p, a negative application condition for p is given by a graph homomorphism $n : L \to N$. For a given source graph G, a match $m : L \to G$ satisfies n if there does not exist an injective morphism $q : N \to G$ such that $m = n; q$

$$\begin{tikzcd}
N \arrow[dr, "q"'] & {} \arrow[d, phantom, "\neq"] & L \arrow[ll, "n"'] \arrow[dl, "m"] \\
& G &
\end{tikzcd}$$

Triple graph grammar

A graph transformation-based approach which is frequently used in the field of model transformation is Triple Graph Grammar (TGG) [42; 70; 109], e.g. TGG comprises the underlying foundation for model to model transformation in the Fujaba [48] and in other QVT-like approaches such as [32; 54; 69]. Since the transformation approach detailed in this thesis is also related to triple graph grammars, we introduce the main concepts of this transformation approach in this section. This introduction is adopted from [42; 70]

A *triple graph* $TrG = (SG \xleftarrow{sg} CG \xrightarrow{tg} TG)$ consists of a source graph SG and a target graph TG that are related via a correspondence graph CG and two graph homomorphisms sg, tg from the correspondence graph to the source and target graphs, respectively. In this way, the source and target graphs are joined into a single structure thus providing a basis for consistent co-evolution of the graphs [42]. The use of correspondence graphs allows to relate a node (arrow) in the source graph with a node (arrow) in the target graph and to constrain these relations by means of constraint languages such as OCL [117].

Operational rules

Triple rules are declarative descriptions for building the source, target and correspondence graphs simultaneously. In order to achieve a transformation step, *operational graph rewriting rules* have to be derived from these declarative rules. Since filtering or deletion of structure can be done by projection, the generated operational transformation rules can be restricted to *monotonic* rules [42; 70]; that is, the LHS of the rule is included in the RHS thus it may only add structure. These operational rules may be used for forward or backward transformations; i.e. they can be used to facilitate bidirectional model transformations [42].

As mentioned, the DPF based approach to model transformation is closely related to the formal framework of graph transformation, especially TGGs. In this regard, it extends graph transformations by adding support for transformation of constraints which come additional to the graph structure of the models. Similar to TGG, in DPF a joined metamodel is used to describe relations between the source and target metamodels. However, one difference is that we can define and constrain, in a diagrammatic way, arbitrary complex relations between source and target metamodel elements. Another difference is that the three graphs comprising triple graphs are represented by a single graph – the underlying graph of the joined (meta)model.

AGG

AGG is a rule based visual tool which supports the algebraic approach to graph transformation [119]. AGG may be used as a transformation engine in Java applications which use graph transformation methods. Among analysis techniques used in AGG are critical pair analysis and consistency checking, facilitating validation of graph transformations. In addition, rule layering is used to control application of transformation rules. Furthermore, negative application conditions can be specified to express requirements for non-existence of substructures. In AGG, graphs may be attributed by Java objects and types in addition to basic data types. Another feature of AGG is that transformation rules may be attributed by Java expressions which are evaluated during rule applications. In addition, rules may have attribute conditions being boolean Java expressions.

RULE VISUALISATION

Graph productions (or graph transformation rules) are presented differently in different tools. In PROGRESS [97] and AGG [119], the LHS and RHS of a rule are shown in a separated style; i.e. in two graphs. The common interface graph K, which represents preserved elements and exist in both LHS and RHS, is usually represented by numbering the nodes and edges of LHS and RHS. More precisely, numbers are used to identify model elements, and a node or edge which is to be preserved will have the same identifier-number in both LHS and RHS of the rule. In contrast, in Fujaba [48] and Henshin [8; 120], a collapsed, compact style is used to visually present rules. In this style, one graph is shown with annotations that depict which elements will be preserved, added and deleted by applying the rule.

PATTERN-BASED TRANSFORMATION

Pattern-based model-to-model transformation is an algebraic, bidirectional and relational approach to model transformation [32]. This approach is based on triple patterns which express allowed and forbidden relations between two models, where the models are triple graphs [42; 70; 109]. Triple patterns can be seen as graph constraints for triple graphs, which specify both negative and positive constraints. Pattern-based specifications are compiled to operational TGG rules, which perform forward and backward model transformations. In [57], the approach is extended by attribute-handling mechanisms. In particular, attribute computations and conditions are integrated in triple patterns. In [94], correctness, completeness and termination properties of pattern-based model-to-model transformation are analysed. In particular, the authors show that it is possible to prove that the compilation mechanism generates graph grammars that are terminating. In addition, they analyse correctness of the compilation of pattern-based specification into operational rules. They also show completeness in the sense that models which are considered relevant can be built by the generated operational rules.

VMTS

The Visual Modeling and Transformation System (VMTS) is an n-layer metamodelling environment which supports editing models according to their metamodels and allows specifying OCL constraints [74]. VMTS provides a graph transformation-based approach to model transformations in which models are formalised as directed, labelled graphs. Moreover, OCL constraints are used to control the execution of transformations. The input and output patterns of transformation rules use metamodel elements; meaning that an instantiation of the input pattern must be found in the source graph instead of an isomorphic subgraph of the pattern. These patterns are guarded by pre- and post-conditions. Before the execution of each transformation rule, the pre-conditions are checked and used to narrow down the set of matches. After execution of each rule, the post-conditions are checked against the output of the rule. In this way, if a rule executes successfully it can be asserted that the transformation has resulted in the expected output.

ATOM^3

The transformation and simulation tool AToM^3 (A Tool for Multi-formalism and Meta-Modelling) is a tool for multi-paradigm modelling [1; 33]. The two main tasks of AToM^3 are metamodelling and model transformation. In AToM^3, both formalisms and models are described as graphs. For each formalism, a visual modelling tool can be generated from a meta-specification which is specified in ER-formalism. This tool is used to visually manipulate models defined by the formalism. Some of the metamodels currently available are: Entity-Relationship, Deterministic Finite state Automata, Non-Deterministic Finite state Automata, Petri Nets, etc. Moreover, graph rewriting is used to perform model transformations. Thus the transformation definitions are expressed as graph grammar models. Typical applications of AToM^3 in the field of model transformation include model simplification, code generation, as well as behaviour-preserving transformations between models defined by different formalisms.

USING OCL

An approach to the analysis of graph transformation rules based on an intermediate OCL representation is presented in [24]. The semantics of rules together with their properties (such as rule applicability, conflict or independence) are transformed into OCL expressions. While these OCL expressions are combined with structural- and attached OCL constraints during the analysis process, the attached OCL constraints are not shown to be transformed. Another approach proposed in [46] employs transformation rules to preserve the semantics of UML/OCL class diagrams when using refactoring rule `moveAttribute`.

CGT

The concrete syntax-based graph transformation framework (CGT) is another graph transformation-based approach to model transformation developed in [55], and compared to AGG and ATL in [56]. The main idea of this approach is that it employs a concrete syntax for definition of model transformation rules. That is, instead of using the concrete syntax of the modelling languages involved in the model transformation, as done in several other approaches, one can define transformation rules employing the same syntax used for definition of the models themselves. In addition, this approach offers a collection operator which is used for matching and transformation of collections of similar subgraphs.

GREAT

The Graph Rewrite And Transformation (GReAT) is a graph transformation-based framework that supports specification of complex model transformations [13; 52; 83]. Transformation rules are specified using a visual tool. Rule application is controlled by explicitly specifying them in a sequence. In addition, test rules are used to change the control flow of rule application [75]. A test rule consists only of an LHS graph and if it is applicable, the next rule in the sequence is executed. In GReAT, the domains of transformation are specified by means of UML and OCL. The complexity of matching is reduced by specification of an initial context for matching. GReAT facilitates composition of source and target metamodels by defining temporary vertex and edge types that can span across multiple domains. This is used to define larger heterogeneous domains by combining different domains together. This feature plays an important role in verifying transformations.

PROGRESS

PROGRES is a visual programming language which has a graph-oriented data model and a graphical syntax for its most important language constructs [97]. It can be seen as a hybrid transformation approach facilitating declarative and imperative definitions of rules. In PROGRES, application of transformation rules are controlled by means of rule sequencing and constructs for rule firing. Like GReAT, PROGRES have test rules construction used to change control flow during rule execution [75]. Moreover, transformation rules in PROGRES can be used in either a deterministic or a non-deterministic manner.

FUJABA

The Fujaba Tool Suite supports model transformation based on TGGs [48]. As mentioned, in Fujaba story diagrams are used to express systems behaviour. In model transformation by Fujaba, these story diagrams are used to define complex control structures for rule applications.

VIATRA2

The VIATRA2 (VIsual Automated model TRAnsformations) framework aims at providing support for the entire life-cycle of engineering model transformations, including the specification, design, execution, validation and maintenance of transformations within and between various modelling languages and domains [123; 124]. The transformation language provided by VIATRA2 is based on graph transformations and Abstract State Machines (ASM). The latter is used to define control flow for the application of transformation rules. Moreover, the transformation language supports both declarative and imperative styles of rule definitions. VIATRA2 provides also a high performance transformation engine supporting incremental model transformations.

ATL

The ATLAS Transformation Language (ATL) was developed in response to the OMG's QVT Request For Proposal (RFP) [66; 67]. ATL is currently available as an open source project under the Eclipse Modeling subproject. ATL is sometimes described as the "QVT of today" [20] and provides an implementation of a transformation definition language. The ATL framework consists of a transformation language, a virtual machine, and an IDE for writing transformation definitions. ATL is a hybrid language – both declarative and imperative. Like QVT, OCL is used to execute the transformations and MOF and Ecore are employed at the meta-level. In fact, an extended version of OCL is used to enable multi-model transformations. In ATL, traceability links are handled automatically. ATL also supports rule inheritance and polymorphic rule references [66].

7.1.3 VERSION CONTROL OF MODELS

The literature on model evolution is abundant. Firstly, there is the issue of model differencing; DSMDiff [76] and EMF Compare [43] 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 model differences, detecting all the additions, deletions and changes. The great benefit of this approach is that it is general, but this is at the price of being resource greedy.

Compared to this approach, our calculation of model differences should require less resources since an explicit representation of commonalities between models is recorded and stored in common models. This avoids the need for structural similarity comparisons.

Secondly, there is the issue of how to represent differences among models that conform to an arbitrary metamodel. Typical approaches represent model differences as follows:

- As models which conform to a difference metamodel. The difference metamodel can be generic [98], or obtained by an automated transformation [28]. 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 difference representation can be computed).
- As a model which is the union of the two compared models, with the modified elements highlighted by colours, tags, or symbols [92]. 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.
- As a sequence of atomic actions specifying how the initial model has been procedurally modified [3]. While this technique has the great advantage of being efficient, difference representation is neither readable nor intuitive. In addition, edit scripts do not follow the “everything is a model vision” [17]. They are suitable for internal representations but quite ineffective for documenting modifications in MDE environments.

According to this classification, our representation of model differences falls into the second category. We represent differences by showing the union of the two compared models and tag the modified elements with predicates (which are also highlighted by colours to enhance readability).

Thirdly, there is research focusing on identifying the types of structural and semantic conflicts that can occur in distributed development. In [82] a predefined set of *a priori* conflicts is identified, on the basis that it is not possible to provide a generic technique for conflict detection with arbitrary accuracy. However, in [29] the authors propose a domain specific modelling language for the definition of weaving models which represent custom conflicting patterns. Moreover, it is possible to describe the resolution criteria through OCL expressions. Currently, our formalisation enables the detection of only syntactic conflicts.

A fourth strand of research focuses on the problem of *heterogeneous synchronisation*. In [6] 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.

7.2 Further Work

As any other research, the content of this thesis is under continuous development. This section is dedicated to present further work and open ends which we are aware of at the present state of DPF. Some of the issues presented here are theoretically oriented, while others are implementation related issues.

DIAGRAM OPERATIONS

When it comes to the definition of diagrammatic predicate signatures, we have encountered several examples in which diagram operations [36] were necessary to describe queries over specifications and to specify derived information. Especially, operations are necessary in order to fully cover OCL constraints. It is also worth a while that for the definition of the semantics of operations, i.e. the meaning of operations on the instance level, the so called Van Kampen square plays a central role. Extending the formal foundation of DPF with support for diagram operations will increase its flexibility and will widen its application areas.

LOGIC

Development of logic and reasoning systems for DPF makes the framework more ready for real-life practical usage. A fully fledged logic which includes diagram operations, and goes beyond specification entailment and universal constraints, will enable users to reason about properties of specifications and to detect faults in specifications, such as inconsistencies, contradictions and unsatisfiablility by instances. In this regard, further work may consider the following dimensions. Firstly, definition of further deduction rules for specification entailments – one deduction rule given in this thesis is described in Proposition 3. Secondly, definition of further logical connectives between constraints – one connective defined in this thesis is conjunction. Thirdly, encountering operations in specification entailments. Finally, studying the preservation and/or reflection of specification entailments along modelling formalism morphisms.

ABSTRACT AND CONCRETE SYNTAX

Another subject for formalisation, regarding (meta)modelling, would be the relation between the so-called concrete and abstract syntax of modelling languages. In most modelling languages, arrows (or associations, links, relations, etc.) at the model level are represented by nodes at the meta-level. In DPF, modelling formalisms are equipped with a visual syntax in which arrows are represented as arrows at all levels of a metamodelling hierarchy. While the syntax used in the examples of this thesis resembles a concrete syntax representation of models, a formal clarification of this syntax would give added value to the framework.

HLR CATEGORY

As obvious from Section 7.1.2, the DPF based approach to model transformation is closely related to the formal framework of graph transformations. The plethora of formal results regarding graph transformations may be extended to DPF.

This may require proving that the category of diagrammatic specification $\mathsf{Spec}(\Sigma)$ is an Adhesive High-Level Replacement (HLR) category.

UNIVERSAL CONSTRAINTS

The DPF based approach to model transformation facilitates the specification of constraint-aware transformation rules. This feature of DPF takes into account atomic constraints from the source side, and uses these constraints to decide how the target side is created. Bringing universal constraints into this picture, along with their roles both during the definition and during the application tasks of model transformations, is left for future work.

RULE SCHEDULING

In this thesis, the application of transformation rules were controlled by means of layering and negative application conditions. As presented in the related work section, there are other mechanisms which can be applied to control rule applications in a way that guarantees functional behaviour of model transformations. The development and implementation of such mechanisms in DPF will increase the usefulness and the usability of the framework.

OTHER DESIRED FEATURES

In order to facilitate various cases of model modifications, such as refactoring of models, it is desirable to add support for in-place model transformations in DPF. This requires support for rules with deletion capabilities. Moreover, higher order transformations – model transformations which have model transformations as input and output – as well as description and characterisation of bidirectional transformations are interesting subjects which both the theoretical foundation and the implementation of DPF should encounter in the future. Furthermore, it will be beneficial to support a collection operator for definition of transformation rules with variable input patterns, as done in [55] for graph transformation.

ANNOTATION OF CONSTRAINTS

In the formalisation of aspects of version control presented in this thesis, difference models and merge models were annotated by means of predicates. These annotations were used on the underlying graphs of specifications. One way to extend this approach is to add support for also annotating constraints, in addition to the underlying graphs.

SEMANTIC CONFLICTS

The rules used for conflict detection in this thesis check only for syntactic conflicts caused by concurrent modifications of the same model elements. Adding support for detection of semantic conflicts – such as conflicts caused by violating of metamodel constraints – will increase the robustness of the approach.

DPF AND DSMLS

We expect that the principles of DPF will ease the definition of domain specific modelling languages (DSML). An application domain for which we have already designed a modelling formalism is the specification of data validation constraints at model level [100]. Other application domains which we plan to apply the DPF based approach to specification of DSMLs are GRID computing [113] and metamodel evolution [49]. These applications will be part of the FormGrid and DISTECH projects [114].

7.2.1 TOOL SUPPORT

An important step in the popularisation of MDE is the development of techniques and tools that support development of models, metamodels and model transformations, which are both precise and intuitive. We have already initiated the design

MODELLING TOOL

and development of a tool environment based on DPF. This tool, which is still in its early stages of development, is implemented as a set of Eclipse plugins and will be available as an open source project at the DPF project web site [16]. The DPF based tool realises the main ideas of DPF, in which modelling formalisms consist of a diagrammatic predicate signature together with a metamodel. Each modelling formalism will be equipped with a palette for creation and modification of models conforming to the corresponding metamodel of the formalism. The palette will be derived from the metamodel and the signature, e.g. a button for each type in the metamodel, and a button for each predicate in the signature.

METAMODELLING SUPPORT

Moreover, the tool will facilitate metamodelling in the following way. When a model is specified in the DPF based tool, it can be used as the metamodel of another modelling formalism. Furthermore, the tool facilitates definition of signatures which can be combined with metamodels to define modelling formalisms. Hence, based on a signature and a metamodel, another (meta)modelling tool will be generated. This tool can serve the same purpose, namely, create and modify models and signatures, which may in turn be used as the building blocks for other modelling formalisms. We expect that this principle of DPF will ease the definition of domain specific modelling languages.

CONFORMANCE

The conformance relation between models and the corresponding metamodel of the formalism will be implemented as a graph homomorphism respecting the constraints added to the metamodel.

OTHER FEATURES

Support for specification of universal constraints, and logic for reasoning about the constraint types mentioned in this thesis will be added gradually. In addition to the (meta)modelling capabilities of the tool, an implementation of the main concepts of constraint-aware model transformations and version control of models is planned. Both extensions will be initiated when the (meta)modelling part nears completion.

7.3 Conclusion

FORMAL BASIS

DPF is a generalisation and adaptation of the categorical sketch formalism, where user-defined diagrammatic predicate signatures represent the constructs of modelling languages. In particular, DPF is an extension of the Generalised Sketches formalism and it aims to combine mathematical rigour with diagrammatic modelling.

(META)MODELLING CONCEPTS

DPF provides a formal, diagrammatic approach to metamodelling, in which models at any level are formalised as diagrammatic specifications. Each diagrammatic specification consists of an underlying graph together with a set of diagrammatic constraints. Moreover, modelling languages are formalised as modelling formalisms. Each modelling formalism consists of a corresponding meta-specification – which specifies the types allowed by the language – and a diagrammatic predicate signature – which collects the set of predicates used to add constraints to specifications specified by the modelling formalism. Furthermore, the conformance relation between a specification at any level, and a specification at the level directly above it, is formalised as a graph homomorphism between the underlying graphs of the specifications satisfying the constraints that are added to the upper level specification. In addition, the conformance relation is strengthened by the concept of universal constraints. These constraints are connected to a certain modelling formalism and are used to express overall requirements that each specification specified by the modelling formalism should satisfy.

DPF provides also a formal approach to the definition of constraint-aware model transformation which is applied to language translation in this thesis. This is possible due to the diagrammatic formalisation the metamodelling hierarchy in which attached constraints are integrated in modelling formalisms, facilitating a uniform transformation of these constraints.

TRANSFORMATION APPROACH

The DPF based approach to model transformation is divided into two tasks: definition task and application task. The definition task is further divided into two steps. Firstly, the source and target modelling languages are joined together; i.e. a joined metamodel is defined. Secondly, the transformation rules are declared as input and output patterns which are typed by the joined metamodel. The input and output patterns of the transformation rules are specifications; and the morphisms between input and output patterns as well as their matches are formalised as constraint- and type preserving specification morphisms. Hence, constraints can be added to the input patterns, and these constraints can be used to control which structure to create in the target model and which constraints to add to the created structure.

The application task is performed as follows. Firstly, the source model is converted to an intermediate model which is typed by the joined metamodel. Next, the transformation rules are iteratively applied to the intermediate model. Finally, the target model is obtained by projection. The approach exploits existing machinery from category theory for the application of transformation rules and for the projection of target models.

VERSION CONTROL

DPF provides also a formal approach to version control of models. In this approach, calculation of differences and the process of merging are formalised as pushout constructions. Moreover, differences between models are represented by means of a diagrammatic predicate signature specially designed for this purpose. In addition, conflict detection is performed based on rules which detect syntactic conflicts.

DPF is a general and open framework – still under development and with potential applications in many areas of software engineering and computer science. This thesis focused on DPF as a formal foundation for MDE. The intention of writing a monograph was to consolidate and present the current state of the development regarding DPF, especially by showing the characteristics of DPF through examples.

There are many aspects and dimensions in the foundation and application of MDE. From these dimensions, we have merely managed to cover and investigate the following:

- Diagrammatic specifications, both typed and untyped versions.
- Definition of domain specific modelling languages.

- Metamodelling hierarchy.
- Specification entailment and Universal constraints.
- Outplace model transformations.
- Version control.

Nevertheless, we hope that we have convinced the reader that DPF has a promising potential to support the foundation and the further development of MDE.

APPENDIX A

This Appendix contains definitions of some categorical constructions we have used in the thesis. In addition, it presents a list of transformation rules for transformation of structural object-oriented models to relational data models.

A.1 Pullback and Pushout

Bearing in mind that the construction of limits and colimits in the category **Graph** is based on the corresponding component-wise constructions in the category **Set** [41], it is possible to extend limit and colimit construction for graphs to the corresponding construction for (typed) specifications.

A.1.1 PULLBACK

In this thesis, pullback constructions were used in the definitions of concepts such as instances of specifications and the projection step during the application task of model transformations. In fact, the pullback constructions we have used are for the underlying graphs of the specifications.

For the general pullback construction of graphs, we refer to [41]. Here, we describe a special case, namely the generalised inverse image construction, that we have used in our examples.

Proposition 9 (Inverse Image of Graphs) *Given the graphs B, C, D, an arbitrary graph homomorphism $m : C \rightarrow D$ and an injective graph homomorphism $n : B \rightarrow D$, we can construct a subgraph A of C together with a graph homomorphism $m^* : A \rightarrow B$ and an inclusion graph homomorphism $inc : A \hookrightarrow C$ such that the resulting diagram is commutative and a pullback in* **Graph**.

$$\begin{tikzcd}
 & A \arrow[dl, "m^*"'] \arrow[dr, hook, "inc"] & \\
B \arrow[dr, "n"'] & PB & C \arrow[dl, "m"] \\
 & D &
\end{tikzcd}$$

Proof. The graph A is defined as follows:

$$\begin{aligned}
A_i &:= \{x \in C_i \mid \exists y \in B_i \quad \text{with } m_i(x) = n_i(y)\}, i = 0, 1 \\
src^A(f) &:= src^C(f) \quad \text{for all } f \in A_1 \\
trg^A(f) &:= trg^C(f) \quad \text{for all } f \in A_1
\end{aligned}$$

The homomorphism property of m and n ensures that we indeed obtain a subgraph A of C. Moreover, there is for any $x_i \in A_i$, with $i = 0, 1$, exactly one $y \in B_i$ with $m_i(x) = n_i(y)$ since the map $n_i : B_i \to D_i$ is injective. This allows us to define $m_i^* : A_i \to B_i$ by assigning to any $x \in A_i$ this unique $y \in B_i$ with $m_i(x) = n_i(y)$. Moreover, the homomorphism property of m and n ensure that m^* becomes a graph homomorphism as well, and the commutativity $m^*; n = inc; m$ is immediately given by the definition of m^*. ♢

A.1.2 PUSHOUT

The general pushout construction for graphs [41] can be extended straightforwardly to the construction of general pushouts for untyped and typed specifications. However, in this thesis we have used only some special cases of the pushout construction and we will now present these special cases in more detail.

In Section 6.3, the calculation of differences and the process of merging were formalised as pushout constructions for injective specification morphisms. In Section 5.4, the application of non-deleting transformation rules was formalised as pushout construction for spans of typed specifications and typed specification morphisms, with one of the legs being inclusion. Both of these cases can be covered by pushout of spans of one injective and one arbitrary specification morphism.

In the following, we show this pushout construction first for untyped specifications. Then we extend the construction, in the standard way [41], to the typed version. Note that we adopt the dot-notation from object-oriented programming to describe the disjoint union of sets.

Proposition 10 (Pushout in $\mathsf{Spec}(\Sigma)$) *Given specifications $\mathfrak{A}, \mathfrak{B}, \mathfrak{C}$, an injective specification morphism $n : \mathfrak{A} \to \mathfrak{C}$, and an arbitrary specification morphism $m : \mathfrak{A} \to \mathfrak{B}$, we can construct a specification $\mathfrak{D}$, an inclusion specification morphism $n^* : \mathfrak{B} \to \mathfrak{D}$ and a specification morphism $m^* : \mathfrak{C} \to \mathfrak{D}$, such that the resulting diagram is commutative and a pushout in $\mathsf{Spec}(\Sigma)$.*

$$\begin{tikzcd}
 & \mathfrak{A} \arrow[dl, "m"'] \arrow[dr, "n"] & \\
\mathfrak{B} \arrow[dr, hook, "n^*"'] & PO & \mathfrak{C} \arrow[dl, "m^*"] \\
 & \mathfrak{D} &
\end{tikzcd}$$

Proof. First, we construct the pushout for the underlying graphs and graph homomorphisms: The graph D is defined as follows:

$$\begin{aligned}
D_i &:= B_i \cup \{C.x \mid x \in C_i, \quad x \notin n_i(A_i)\}, i = 0,1 \\
src^D(f) &:= \begin{cases} src^B(f), & if \quad f \in B_1 \\ m_0(src^C(f)), & if \quad f \in C_1, src^C(f) \in n_0(A_0) \\ src^C(f), & if \quad f \in C_1, src^C(f) \notin n_0(A_0) \end{cases} \\
trg^D(f) & \quad \text{is defined analogously}
\end{aligned}$$

The inclusion $n^* : B \hookrightarrow D$ is given by construction and the graph homomorphism $m^* : C \to D$ is defined by the following, for $i = 0, 1$:

$$m_i^*(x) := \begin{cases} m_i(x), & if \; x \in n_i(A_i) \\ C.x, & if \; x \notin n_i(A_i) \end{cases}$$

Second, we define the set of constraints $C^{\mathfrak{D}}$ by

$$C^{\mathfrak{D}} := C^{\mathfrak{B}} \cup \{(q, \sigma; m^*) \mid (q, \sigma) \in C^{\mathfrak{C}}\}$$

and obtain, finally, a pushout in $\mathsf{Spec}(\Sigma \rhd G)$. $\diamondsuit$

Remark 32 (Identification of Constraints) *Two constraints $(p, \delta) \in C^{\mathfrak{B}}$ and $(p, \sigma) \in C^{\mathfrak{C}}$ such that $\delta; m^* = \sigma; n^*$ are mapped to the same constraint $(p, \delta; m^*) = (p, \sigma; n^*) \in C^{\mathfrak{D}}$. Especially, we obtain for all constraints $(r, \gamma) \in C^{\mathfrak{A}}$ just one constraint $(r, \gamma; n; m^*) = (r, \gamma; m; n^*) \in C^{\mathfrak{D}}$.*

Proposition 11 (Pushout in $\mathsf{Spec}(\Sigma \rhd G)$) *Given a graph G and typed specifications $\mathfrak{A} \rhd G, \mathfrak{B} \rhd G, \mathfrak{C} \rhd G$, an arbitrary typed specification morphism $m : \mathfrak{A} \rhd G \to \mathfrak{B} \rhd G$ and an injective typed specification morphism $n : \mathfrak{A} \rhd G \to \mathfrak{C} \rhd G$, we can construct a typed specification $\mathfrak{D} \rhd G$, an inclusion typed specification morphism*

$n^* : \mathfrak{B}\rhd G \hookrightarrow \mathfrak{D}\rhd G$ and a typed specification morphism $m^* : \mathfrak{C}\rhd G \to \mathfrak{D}\rhd G$, such that the resulting diagram is commutative and a pushout in $\mathsf{Spec}(\Sigma\rhd G)$*.*

$$\begin{tikzcd}
& \mathfrak{A}\rhd G \arrow[dl, "m"'] \arrow[dr, "n"] & \\
\mathfrak{B}\rhd G \arrow[dr, hook, "n^*"'] & \text{P.O.} & \mathfrak{C}\rhd G \arrow[dl, "m^*"] \\
& \mathfrak{D}\rhd G &
\end{tikzcd}
\qquad
\begin{tikzcd}
& (A,\iota^A) \arrow[dl, "m"'] \arrow[dr, "n"] & \\
(B,\iota^B) \arrow[dr, hook, "n^*"'] & \text{P.O.} & (C,\iota^C) \arrow[dl, "m^*"] \\
& (D,\iota^D) &
\end{tikzcd}$$

$$\begin{tikzcd}
& A \arrow[dl, "m"'] \arrow[dr, "n"] & \\
B \arrow[dr, hook, "n^*"'] \arrow[ddr, bend right, "\iota^B"'] & \text{P.O.} & C \arrow[dl, "m^*"] \arrow[ddl, bend left, "\iota^C"] \\
& D \arrow[d, "\iota^D"'] & \\
& G &
\end{tikzcd}$$

Proof. The construction in Proposition 10 provides a pushout in $\mathsf{Spec}(\Sigma)$. We only need to show how the construction extends to typing:

By assumption we have $m;\iota^B = \iota^A = n;\iota^B$ thus the pushout property provides us a unique graph homomorphism $\iota^D : D \to G$ such that $n^*;\iota^D = \iota^B$ and $m^*;\iota^D = \iota^C$. ι^D is given by, for $i = 0, 1$:

$$\iota_i^D(x) := \begin{cases} \iota^B(x), & if\ x \in B_i \\ \iota^C(x), & if\ x \in C_i, x \notin n_i(A_i) \end{cases}$$

The two equations for ι^D mean that n^* and m^*, respectively, are compatible with typing and we obtain, in such a way, a pushout in the category $\mathsf{Spec}\Sigma\rhd G$. ♢

A.2 More Transformation Cases

In this section, we show a list of transformation rules for the transformation object-oriented structural models to relational data models. These rules are defined for pairs of classes which are connected by single references of by pairs of inverse references. For readability reasons, the rules are expressed in an abbreviated fashion which, in fact, show the results of applying the rules, e.g. the first rule in Table A.1 corresponds to the rule r_5 in Table 5.3.

In Table A.1, the names of the nodes indicate the mappings between the LHS and RHS of the rules. Moreover, we have omitted the names and types of arrows in LHS and RHS of the rules since it is obvious that they are of types Reference and Column, respectively.

Note also that on the object-oriented side, the constraints $(\texttt{[surjective]}, \delta)$ and $(\texttt{[non-overlapping]}, \delta)$ are entailed from the corresponding multiplicity constraints. These constraints are written in blue.

Table A.1: List of transformation rules

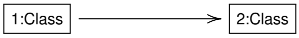
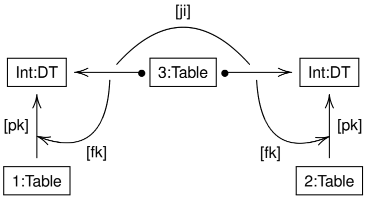
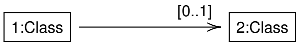
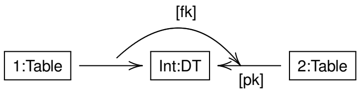
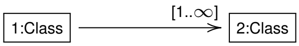
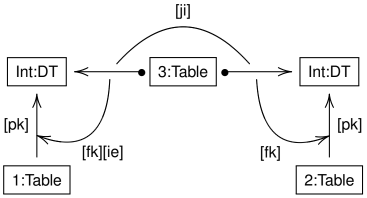
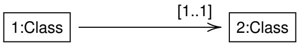
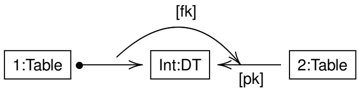
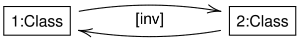
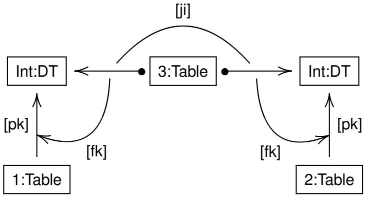
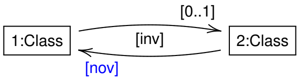
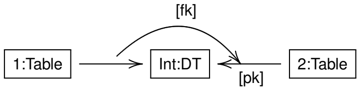
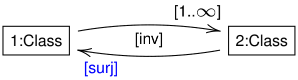
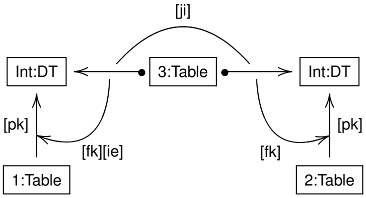
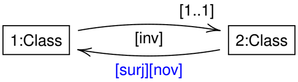
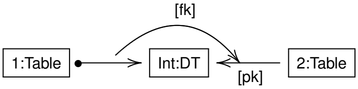
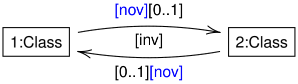
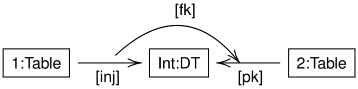
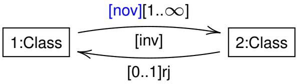
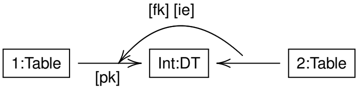
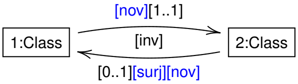
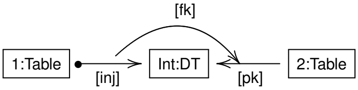
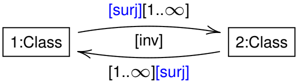
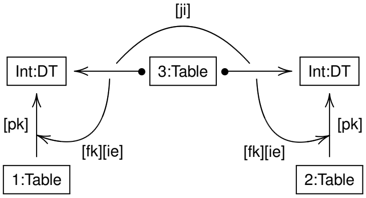
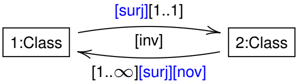
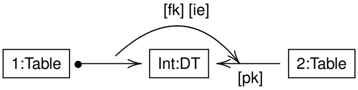
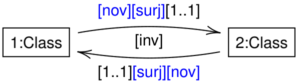
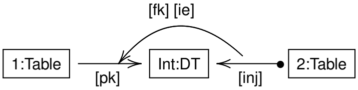
No	OO	RDM
1.		 [ji] Int:DT 3:Table Int:DT [pk] [fk] [fk] [pk] 1:Table 2:Table
2.		
3.		 [ji] Int:DT 3:Table Int:DT [pk] [fk][ie] [fk] [pk] 1:Table 2:Table
4.		
5.		 [ji] Int:DT 3:Table Int:DT [pk] [fk] [fk] [pk] 1:Table 2:Table
6.		

Table A.1: List of transformation rules

<table>
<tr><th>No</th><th>OO</th><th>RDM</th></tr>
<tr><td>7.</td><td> </td><td> </td></tr>
<tr><td>8.</td><td> </td><td> </td></tr>
<tr><td>9.</td><td> </td><td> </td></tr>
<tr><td>10.</td><td> </td><td> </td></tr>
<tr><td>11.</td><td> </td><td> </td></tr>
<tr><td>12.</td><td> </td><td> </td></tr>
<tr><td>13.</td><td> </td><td> </td></tr>
<tr><td>14.</td><td> </td><td> </td></tr>
</table>

References

[1] A Tool for Multi-formalism and Meta-Modelling. *Project Web Site*. `http://atom3.cs.mcgill.ca/`.

[2] D. H. Akehurst and S. Kent. A Relational Approach to Defining Transformations in a Metamodel. In *UML*, pages 243–258, 2002.

[3] M. Alanen and I. Porres. Difference and Union of Models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003: 6th International Conference on The Unified Modeling Language, Modeling Languages and Applications*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003. ISBN 3-540-20243-9.

[4] Alcatel, Softeam, THALES, TNI-Valiosys, and Codagen. *OpenQVT Revised Submission to the MOF 2.0 QVT RFP*, August 2003. `http://www.omg.org/cgi-bin/doc?ad/2003-08-05`.

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

[6] M. Antkiewicz and K. Czarnecki. Design Space of Heterogeneous Synchronization. In R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE 2007: Generative and Transformational Techniques in Software Engineering II, International Summer School*, volume 5235 of *LNCS*, pages 3–46. Springer, 2008. ISBN 978-3-540-88642-6. doi: 10.1007/978-3-540-88643-3_1.

[7] Antoni Diller. *Z: An Introduction to Formal Methods*. Wiley, 1994. ISBN 0471939730.

[8] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and tools for In-Place EMF Model Transformation. In *MoDELS 2010: 13^{t}h International Conference on Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.

[9] Ask.com. *Dictionary.com*. `http://dictionary.reference.com`.

[10] C. Atkinson and T. Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003. doi: 10.1109/MS.2003.1231149.

[11] Atlas Transformation Language. *User Guide*. `http://wiki.eclipse.org/ATL/User_Guide`.

[12] T. Baar and J. Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In I. Virbitskaite and A. Voronkov, editors, *PSI 2006: 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, volume 4378 of *LNCS*, pages 84–97. Springer, 2007. ISBN 978-3-540-70880-3. doi: 10.1007/978-3-540-70881-0_10.

[13] D. Balasubramanian, A. Narayanan, C. P. van Buskirk, and G. Karsai. The Graph Rewriting and Transformation Language: GReAT. *ECEASST*, 1, 2006.

[14] S. V. Balen, R. V. D. Straeten, and T. Mens, editors. *ChaMDE 2008: 1st International Workshop on Challenges in Model-Driven Software Engineering*, September 2008.

[15] M. Barr and C. Wells. *Category Theory for Computing Science (2nd Edition)*. Prentice Hall International Ltd., Hertfordshire, UK, 1995. ISBN 0-13-323809-1.

[16] Bergen University College and University of Bergen. *Diagram Predicate Framework (DPF)*. `http://dpf.hib.no/`.

[17] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005. doi: 10.1007/s10270-005-0079-0.

[18] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *ASE 2001: 16th IEEE International Conference on Automated Software Engineering*, pages 273–280, 2001. ISBN 0-7695-1426-X. doi: 10.1109/ASE.2001.989813.

[19] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. EMF Model Refactoring based on Graph Transformation Concepts. *ECEASST*, 3, 2006.

[20] M. Bohlen. QVT and multi metamodel transformations in MDA. Technical report, February 2006.

[21] A. Boronat and J. Meseguer. Algebraic Semantics of OCL-Constrained Metamodel Specifications. In M. Oriol and B. Meyer, editors, *TOOLS Europe 2009: 47th International Conference on Objects, Components, Models and Patterns*, volume 33 of *LNBIP*, pages 96–115. Springer, 2009. ISBN 978-3-642-02571-6. doi: 10.1007/978-3-642-02571-6_7.

[22] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A Visualization of OCL Using Collaborations. In *UML 2001: 4th International Conference on The Unified Modeling Language, Modeling Languages and Applications*, volume 2185 of *LNCS*, pages 257–271. Springer, 2001. ISBN 3-540-42667-1.

[23] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A Manifesto for Model Merging. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, New York, NY, USA, 2006. ACM. ISBN 1-59593-410-3. doi: 10.1145/1138304.1138307.

[24] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Analysing Graph Transformation Rules through OCL. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *ICMT 2008: 1st International Conference on Model Transformation*, volume 5063 of *LNCS*, pages 229–244. Springer, 2008. ISBN 978-3-540-69926-2. doi: 10.1007/978-3-540-69927-9_16.

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

[26] CBOP, DSTC, and IBM. *Joint Revised Submission to the MOF 2.0 QVT RFP*, August 2003. `http://www.omg.org/cgi-bin/doc?ad/2003-08-03`.

[27] P. P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976. ISSN 0362-5915. doi: 10.1145/320434.320440.

[28] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9, Special Issue on TOOLS Europe 2007):165–185, October 2007.

[29] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *MoDELS 2008: 11th International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 311–325, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-87874-2. doi: 10.1007/978-3-540-87875-9_23.

[30] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion (2nd Edition)*. O'Reilly Media, October 2008. ISBN 0596510330.

[31] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *OOPSLA 2003: 2nd Workshop on Generative Techniques in the Context of MDA*, 2003.

[32] J. de Lara and E. Guerra. Pattern-Based Model-to-Model Transformation. In *ICGT 2008: 4th International Conference on Graph Transformations*, volume 5214 of *LNCS*, pages 426–441. Springer, 2008. doi: 10.1007/978-3-540-87405-8_29.

[33] J. de Lara and G. Taentzer. Automated Model Transformation and Its Validation Using AToM 3 and AGG. In *Diagrams*, volume 2980 of *LNCS*, pages 182–198. Springer, 2004. ISBN 978-3-540-21268-3. doi: 10.1007/b95854.

[34] Z. 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.

[35] Z. Diskin and J. Dingel. Mappings, Maps and Tables: Towards Formal Semantics for Associations in UML2. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MoDELS 2006: 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 230–244. Springer, 2006. ISBN 3-540-45772-0. doi: 10.1007/11880240_17.

[36] Z. Diskin and B. Kadish. Variable set semantics for keyed generalized sketches: formal semantics for object identity and abstract syntax for conceptual modeling. *Data Knowl. Eng.*, 47(1):1–59, 2003. doi: 10.1016/S0169-023X(03)00047-8.

[37] Z. Diskin and B. Kadish. Generic Model Management. In *Encyclopedia of Database Technologies and Applications*, pages 258–265. Idea Group, 2005. ISBN 1-59140-560-2.

[38] Z. Diskin and U. Wolter. A Diagrammatic Logic for Object-Oriented Visual Modeling. In *ACCAT 2007: 2nd Workshop on Applied and Computational Category Theory*, volume 203/6 of *ENTCS*, pages 19–41, Amsterdam, The Netherlands, 2008. Elsevier Science Publishers B. V. doi: 10.1016/j.entcs.2008.10.041.

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

[40] H. Ehrig, U. Prange, and G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *ICGT 2004: 2nd International Conference on Graph Transformations*, volume 3256 of *LNCS*, pages 161–177. Springer, 2004. doi: 10.1007/b100934.

[41] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006. ISBN 3540311874.

[42] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information Preserving Bidirectional Model Transformations. In M. B. Dwyer and A. Lopes, editors, *FASE 2007: 10th International Conference on Fundamental Approaches to Software Engineering*, volume 4422 of *LNCS*, pages 72–86. Springer, 2007. ISBN 978-3-540-71288-6. doi: 10.1007/978-3-540-71289-3_7.

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

[44] Epsilon. *Book*. `http://epsilonlabs.wiki.sourceforge.net/Book`.

[45] J. L. Fiadeiro. *Categories for Software Engineering*. Springer, May 2004. ISBN 3540209093.

[46] F. Fondement and T. Baar. Making Metamodels Aware of Concrete Syntax. In A. Hartman and D. Kreische, editors, *ECMDA-FA 2005: 1st European Conference on Model-Driven Architecture Foundations and Applications*, volume 3748 of *LNCS*, pages 190–204. Springer, 2005. ISBN 3-540-30026-0. doi: 10.1007/11581741_15.

[47] D. S. Frankel and J. Parodi. *The MDA Journal: Model Driven Architecture Straight From The Masters*. Meghan Kiffer Pr, 2004. ISBN 0929652258.

[48] Fujaba Developer Team. *The Fujaba Tool Suite*. `http://www.fujaba.de/`.

[49] T. Gîrba, J.-M. Favre, and S. Ducasse. Using Meta-Model Transformation to Model Software Evolution. *ENTCS*, 137(3):57–64, 2005. doi: 10.1016/j.entcs.2005.07.005.

[50] C. Gonzalez-Perez and B. Henderson-Sellers. *Metamodelling for Software Engineering*. Wiley, 2008. ISBN 0470030364.

[51] Graphical Modeling Framework. *Project Web Site*. `http://www.eclipse.org/gmf/`.

[52] GReAT: Graph Rewriting and Transformation. *Project Web Site*. `http://www.isis.vanderbilt.edu/tools/GReAT`.

[53] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004. ISBN 0471202843.

[54] J. Greenyer and E. Kindler. Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars. *Software and System Modeling*, 9(1):21–46, 2010. doi: 10.1007/s10270-009-0121-8.

[55] R. Grønmo. *Using Concrete Syntax in Graph-based Model Transformations*. PhD thesis, Department of Informatics, University of Oslo, Norway, February 2010.

[56] R. Grønmo, B. Møller-Pedersen, and G. K. Olsen. Comparison of Three Model Transformation Languages. In R. F. Paige, A. Hartman, and A. Rensink, editors, *ECMDA-FA: 5th European Conference on Model Driven Architecture - Foundations and Applications*, volume 5562 of *LNCS*, pages 2–17. Springer, 2009. ISBN 978-3-642-02673-7. doi: 10.1007/978-3-642-02674-4_2.

[57] E. Guerra, J. de Lara, and F. Orejas. Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions. In R. F. Paige, editor, *ICMT 2009: 2nd International Conference on Model Transformation*, volume 5563 of *LNCS*, pages 83–99. Springer, 2009. ISBN 978-3-642-02407-8. doi: 10.1007/978-3-642-02408-5_7.

[58] A. Habel, J. Müller, and D. Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001. doi: 10.1017/S0960129501003425.

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

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

[61] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0521496195.

[62] John Dawes. *The VDM-SL Reference Guide*. CRC Press, 1991. ISBN 0273031511.

[63] M. Johnson and R. D. Rosebrugh. Implementing a Categorical Information System. In J. Meseguer and G. Rosu, editors, *AMAST 2008: 12th International Conference on Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 232–237. Springer, 2008. ISBN 978-3-540-79979-5. doi: 10.1007/978-3-540-79980-1_18.

[64] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1996. ISBN 0521559766.

[65] C. B. Jones. *Systematic Software Development Using Vdm (Prentice-Hall International Series in Computer Science)*. Prentice Hall, 1990. ISBN 0138807337.

[66] F. Jouault and I. Kurtev. On the architectural alignment of ATL and QVT. In H. Haddad, editor, *SAC 2006: 21nd ACM Symposium on Applied Computing*, pages 1188–1195. ACM, 2006. ISBN 1-59593-108-2. doi: 10.1145/1141277.1141561.

[67] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In P. L. Tarr and W. R. Cook, editors, *OOPSLA 2006: 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 719–720. ACM, 2006. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176691.

[68] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 032119442X.

[69] A. Königs. *Model Integration and Transformation - A Triple Graph Grammar-based QVT Implementation*. PhD thesis, Electrical Engineering and Information Technology, Technischen Universität Darmstadt, Germany, October 2008.

[70] A. Königs and A. Schürr. Tool Integration with Triple Graph Grammars – A Survey. *ENTCS*, 148(1):113–150, 2006. doi: 10.1016/j.entcs.2005.12.015.

[71] T. Kühne. Matters of (Meta-)Modeling. *Software and System Modeling*, 5 (4):369–385, 2006. doi: 10.1007/s10270-006-0017-9.

[72] T. Kühne. Clarifying matters of (meta-) modeling: an author's reply. *Software and System Modeling*, 5(4):395–401, 2006. doi: 10.1007/s10270-006-0034-8.

[73] I. Kurtev, J. Bézivin, and F. J. andPatrick Valduriez. Model-Based DSL Frameworks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 602–616, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176632.

[74] L. Lengyel, T. Levendovszky, and H. Charaf. Constraint Validation Support in Visual Model Transformation Systems. *Acta Cybernetica*, 17(2): 339–357, 2005.

[75] L. Lengyel, T. Levendovszky, and H. Charaf. Validated model transformation-driven software development. *IJCAT*, 31(1/2):106–119, 2008. doi: 10.1504/IJCAT.2008.017723.

[76] Y. Lin, J. Gray, and F. 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.

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

[78] F. Mantz. Syntactic Quality Assurance Techniques for Software Models. Diploma thesis, Department of Mathematics and Informatics, Philipps University in Marburg, Germany, August 2009.

[79] S. Marković and T. Baar. Refactoring OCL annotated UML class diagrams. *Software and System Modeling*, 7(1):25–47, 2008. doi: 10.1007/s10270-007-0056-x.

[80] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1000449.

[81] T. Mens and P. V. Gorp. A Taxonomy of Model Transformation. *ENTCS*, 152:125–142, 2006. doi: 10.1016/j.entcs.2005.10.021.

[82] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *ENTCS*, 127(3):113–128, 2005. doi: 10.1016/j.entcs.2004.08.038.

[83] A. Narayanan and G. Karsai. Towards Verifying Model Transformations. *ENTCS*, 211:191–200, 2008. ISSN 1571-0661. doi: 10.1016/j.entcs.2008.04.041.

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

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

[86] Object Management Group. *Unified Modeling Language Specification*, January 2005. `http://www.omg.org/cgi-bin/doc?formal/2005-04-01`.

[87] Object Management Group. *Meta-Object Facility Specification*, January 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-01-01`.

[88] Object Management Group. *Object Constraint Language Specification*, May 2006. `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.

[89] Object Management Group. *XML Metadata Interchange Specification*, December 2007. `http://www.omg.org/cgi-bin/doc?formal/2007-12-01`.

[90] Object Management Group. *Query/View/Transformation Specification*, April 2008. `http://www.omg.org/cgi-bin/doc?formal/2008-04-03`.

[91] Object Management Group. *Unified Modeling Language Specification*, February 2009. `http://www.omg.org/cgi-bin/doc?formal/2009-02-04`.

[92] D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. In *ESEC/FSE 2003: 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003*, pages 227–236. ACM, 2003. doi: 10.1145/940071.940102.

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

[94] F. Orejas, E. Guerra, J. de Lara, and H. Ehrig. Correctness, Completeness and Termination of Pattern-Based Model-to-Model Transformation. In *CALCO 2009: 3rd International Conference on Algebra and Coalgebra in Computer Science*, volume 5728 of *LNCS*, pages 383–397. Springer, 2009. doi: 10.1007/978-3-642-03741-2_26.

[95] A. Petter, A. Behring, and M. Mühlhäuser. Solving Constraints in Model Transformations. In R. Paige, editor, *ICMT 2009: 2nd International Conference on Model Transformation*, volume 5563 of *LNCS*, pages 132–147. Springer, 2009. ISBN 978-3-642-02407-8. doi: 10.1007/978-3-642-02408-5_10.

[96] I. Poernomo. A Type Theoretic Framework for Formal Metamodelling. In *International Seminar on Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*, pages 262–298. Springer, 2006. ISBN 3-540-35800-5. doi: 10.1007/11786160_15.

[97] U. Ranger and E. Weinell. The Graph Rewriting Language and Environment PROGRES. In A. Schürr, M. Nagl, and A. Zündorf, editors, *AGTIVE 2007: 3rd International Symposium on Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *LNCS*, pages 575–576. Springer, 2007. ISBN 978-3-540-89019-5. doi: /10.1007/978-3-540-89020-1_41.

[98] J. E. Rivera and A. Vallecillo. Representing and Operating with Model Differences. In *TOOLS Europe 2008: 46th International Conference on Objects, Components, Models and Patterns*, volume 11 of *LNBIP*, pages 141–160. Springer, 2008. ISBN 978-3-540-69823-4. doi: 10.1007/978-3-540-69824-1_9.

[99] J. R. Romero, J. E. Rivera, F. Durán, and A. Vallecillo. Formal and Tool Support for Model Driven Engineering with Maude. *Journal of Object Technology*, 6(9):187–207, 2007.

[100] A. Rossini, A. Rutle, F. Mancini, D. Hovland, K. A. Mughal, Y. Lamo, and U. Wolter. A Formal Approach to the Specification of Data Validation Constraints in MDE. In *NWPT 2009: 21st Nordic Workshop on Programming Theory*, pages 86–88, October 2009. ISBN 978-87-643-0565-4.

[101] A. Rossini, A. Rutle, Y. Lamo, and U. 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. ISSN 1567-8326. doi: 10.1016/j.jlap.2009.10.003.

[102] J. Rothenberg. *The nature of modeling*. Santa Monica: The Rand Corporation, 1989.

[103] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific Publishing Company, River Edge, NJ, USA, 1997. ISBN 98-102288-48.

[104] A. Rutle, U. Wolter, and Y. Lamo. A Diagrammatic Approach to Model Transformations. In *EATIS 2008: Euro American Conference on Telematics and Information Systems*, pages 1–8. ACM, 2008. ISBN 978-1-59593-988-3. doi: 10.1145/1621087.1621105.

[105] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A Category-Theoretical Approach to the Formalisation of Version Control in MDE. In M. Chechik and M. Wirsing, editors, *FASE 2009: 12th International Conference on Fundamental Approaches to Software Engineering*, volume 5503 of *LNCS*, pages 64–78. Springer, 2009. ISBN 978-3-642-00592-3. doi: 10.1007/978-3-642-00593-0_5.

[106] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A Diagrammatic Formalisation of MOF-Based Modelling Languages. In M. Oriol and B. Meyer, editors, *TOOLS Europe 2009: 47th International Conference on Objects, Components, Models and Patterns*, volume 33 of *LNBIP*, pages 37–56. Springer, 2009. ISBN 978-3-642-02571-6. doi: 10.1007/978-3-642-02571-6_4.

[107] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A Formalisation of Constraint-Aware Model Transformations. In D. Rosenblum and G. Taentzer, editors, *FASE 2010: 13th International Conference on Fundamental Approaches to Software Engineering*, volume 6013 of *LNCS*, pages 13–28. Springer, 2010. ISBN 978-3-642-12028-2. doi: 10.1007/978-3-642-12029-9_2.

[108] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A Formal Approach to the Specification and Transformation of Constraints in MDE. *Journal of Logic and Algebraic Programming*, Submitted.

[109] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *WG :20$^t h$ International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994. ISBN 3-540-59071-4. doi: /10.1007/3-540-59071-4_45.

[110] R. W. Sebesta. *Concepts of Programming Languages (8th Edition)*. Addison Wesley, 2007. ISBN 0321493621.

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

[112] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5): 42–45, 2003.

[113] H. D. Simon and B. Wilkinson. *Grid Computing - Techniques and Applications*. Chapman & Hall, 2009. ISBN 78-1-4200695-3-2.

[114] Software Technologies for Distributed Systems. *Project Web Site*. `http://prosjekt.hib.no/distech/index.html`.

[115] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0 (2nd Edition)*. Addison-Wesley Professional, 2008. ISBN 0321331885.

[116] Steve Schneider. *The B-method (Cornerstones of Computing)*. Palgrave Macmillan, 2001. ISBN 033379284X.

[117] M. Stölzel, S. Zschaler, and L. Geiger. Integrating OCL and Model Transformations in Fujaba. *ECEASST*, 5, 2006.

[118] G. Taentzer and A. Rensink. Ensuring Structural Constraints in Graph-Based Models with TypeInheritance. In M. Cerioli, editor, *FASE 2005: 8th International Conference on Fundamental Approachesto Software Engineering*, volume 3442 of *LNCS*, pages 64–79. Springer, 2005. ISBN 3-540-25420-X.

[119] The Attributed Graph Grammar System. *Project Web Site*. `http://user.cs.tu-berlin.de/~gragra/agg/`.

[120] The EMF Henshin Transformation Tool. *Project Web Site*. `http://www.eclipse.org/modeling/emft/henshin/`.

[121] The Entity Attribute Sketch Implementation Kit. *Project Web Site*. `http://mathcs.mta.ca/research/rosebrugh/Easik/`.

[122] The Maude System. *Project Web Site*. `http://maude.cs.uiuc.edu/`.

[123] The VIsual Automated model TRAnsformations. *Project Web Site*. `http://www.eclipse.org/gmt/VIATRA2/`.

[124] D. Varró and A. Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.*, 68(3):214–234, 2007. doi: 10.1016/j.scico.2007.05.004.

[125] Visual OCL. *Project Web Site*. `http://tfs.cs.tu-berlin.de/vocl/`.

[126] J. Warmer and A. Kleppe. *The Object Constraint Language (2^{nd} Edition): Getting your models ready for MDA*. Addison-Wesley, August 2003. ISBN 0321179366.

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

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

[129] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof (Prentice-Hall International Series in Computer Science).* Prentice Hall, 1996. ISBN 0139484728.