

Coupled Transformations of Graph Structures applied to Model Migration

Florian Mantz

Dissertation for the degree of "Doktor der Naturwissenschaften" (Dr. rer. nat.)
Philipps-Universität Marburg, Germany
2014

Philipps-Universität Marburg, Germany
Fachbereich 12
2014



- Fachbereich Mathematik und Informatik -

Coupled Transformations of Graph Structures applied to Model Migration

FLORIAN MANTZ

Coupled Transformations of Graph Structures applied to Model Migration

DISSERTATION

vorgelegt von
Dipl.-Inf. FLORIAN MANTZ
geboren am 25.10.78 in Schwalmstadt-Ziegenhain

Vom Fachbereich 12
– Mathematik und Informatik –
der Philipps-Universität Marburg



zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
– Dr. rer. nat. –

genehmigte Dissertation

Gutachter:

Prof. Dr. Gabriele Taentzer, Philipps-Universität Marburg
Prof. Dr. Juan de Lara, Universidad Autónoma de Madrid

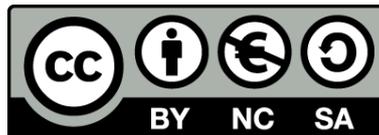
Prüfungskommission:

Prof. Dr. Manfred Sommer, Vorsitzender und Dekan
Prof. Dr. Gabriele Taentzer
Prof. Dr. Juan de Lara
Prof. Dr. Bernhard Seeger

als Dissertation eingereicht am: 20.08.2014
Tag der mündlichen Prüfung: 15.10.2014

erschienen: Philipps-Universität Marburg, 2014
Hochschulkennziffer 1080

Originaldokument gespeichert auf dem Publikationsserver der
Philipps-Universität Marburg
<http://archiv.ub.uni-marburg.de>



Dieses Werk bzw. Inhalt steht unter einer
Creative Commons
Namensnennung
Keine kommerzielle Nutzung
Weitergabe unter gleichen Bedingungen
3.0 Deutschland Lizenz.

Die vollständige Lizenz finden Sie unter:
<http://creativecommons.org/licenses/by-nc-sa/3.0/de/>

To my mom

Contents

Preface	xi
Scientific Environment	xv
Abstract	xvii
Abstract (Deutsch)	xix
Abstract (Norsk)	xxi
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Results	3
1.4 Overview	5
2 Model-Driven Engineering	9
2.1 Introduction	9
2.2 Modeling in Software Engineering	11
2.3 Meta-modeling	12
2.4 Constraints	14
2.5 Model Transformation	18
2.6 Model Migration	20
2.6.1 General Approaches to Model Migration	25
2.6.2 Correctness of Model Migrations	26
2.6.3 Reusability of Model Migrations	26
2.6.4 Customization of Model Migrations	27
3 Graph-based Modeling	29
3.1 Graphs supporting Attribution	29
3.2 Graphs supporting Inheritance	37
3.3 Graphs supporting Language Constraints	43
4 Adhesive Categories and Graph Transformations	47
4.1 Adhesive Categories	47

vii

4.2	Properties of (Adhesive) Categories	50
4.3	Graph Transformations based on Cospans	52
4.4	Cospan Double Pushout Approach	53
4.5	Cospan Sesqui Pushout Approach	56
4.6	Summary of Approach Differences	58
4.7	Application Condition	58
4.8	Transformation Variants	59
5	Detecting Evolution Steps by Graph Transformation Rules	61
5.1	Introduction	62
5.2	Detecting Evolution Steps with Cospan Rules	65
5.3	Detecting Evolution Steps with Span Rules	70
5.4	Advantages of Cospan Rule Detection	72
6	Coupled Transformations based on Graph Transformations	75
6.1	Coupled Transformations	75
6.2	Constructing Coupled Transformations (Left Part)	81
6.3	Constructing Coupled Transformations (Right Part)	85
6.4	Standard Construction for Coupled Transformations	90
6.5	Span versus Cospan Transformations	97
7	Model Migration Schemes based on Coupled Transformations	101
7.1	Migration by Amalgamated Graph Transformations	101
7.2	Migration Rules from Migration Schemes	103
7.3	Default Migration Schemes	114
8	Co-Evolution of Object-Oriented Models	117
8.1	Introduction	117
8.2	Supported Change Operations	118
8.3	Merging of Model Elements	119
8.4	Retrying Model Elements to Subtypes	123
8.5	Model Migration Schemes	126
8.6	Classification of Meta-model Changes Revisited	130
9	Towards Model Migration Ensuring Constraint Satisfaction	135
9.1	Introduction	135
9.2	Resolution Procedure	136
9.3	Finitely Satisfiable Meta-models wrt. Multiplicities	141
9.4	Deriving Constraint Resolution Rules	143
9.5	Resolving Multiplicity Constraint Violations	150
10	Migrating UML Activity Models from Version 1.4 to 2.2	155
10.1	About the Transformations Tool Contest 2010	155
10.2	The Migration Task	157
10.3	The DPF Text Modeling Framework	160

10.4	Model Migration by Coupled Transformations	163
10.5	On the Results of the Transformation Tool Contest	178
11	Related Work	181
11.1	Schema Evolution	181
11.2	Meta-model Evolution	182
11.3	Correctness Properties of Model Migrations	184
11.4	Reuse of Migration Knowledge	186
11.5	Deduction of Model Migration Specifications	187
11.6	Customization of Model Migration Specifications	188
11.7	Employed Model Transformation Approaches	188
12	Conclusion and Future Work	191
12.1	Summary	191
12.2	Outlook	193
	Appendices	195
A	Proofs of Auxiliary Propositions	197
A.1	Generalizing the Special Pullback-Pushout Property	197
A.2	On the Stability of Final Pullback Complements	202
B	Case Study: Adhesiveness	209
B.1	Categories of Simple Directed Graphs	209
B.1.1	Pushouts/Pullbacks of Simple Directed Graphs	210
B.1.2	Van Kampen Property for Simple Graphs	212
B.2	Category of (Directed Multi-)Graphs	214
B.2.1	Pushouts/Pullbacks of Directed Multi-Graphs	214
B.2.2	Van Kampen Property for Directed Multi-Graphs	221
B.3	Category of DPF Specifications	224
B.3.1	Pushouts and Pullbacks in Spec	225
B.3.2	Van Kampen Property in Spec	227
B.4	The Category of Generalized DPF Specifications	228
B.4.1	Pushouts and Pullbacks in GSpec	230
B.4.2	Van Kampen Property in GSpec	236
B.5	Conclusion	238
	Bibliography	241

Preface

Finally, after more than four years of hard work but also lots of fun and traveling, I am very happy and proud that I finished my doctoral thesis. It has not always been easy and many times I was not sure if this day will come but in the end everything turned out well. I have got some nice results which beautifully fit together as you will hopefully agree, if you find the time to read it. If not, I hope you will at least enjoy reading this preface.

Bergen, 31st July 2014

Acknowledgements

Doing a PhD, or in my case a Dr. rer. nat, is seldom a process that one can perform alone. Usually a lot of people are involved who provide guidance, friendship, constructive critics or moral support. Therefore I wanted to thank at this point all who in one way or another contributed to the completion of this thesis.

First, I want to thank my supervisors Gabriele Taentzer, Yngve Lamo and Uwe Wolter (in no particular order). Although each of them is affiliated with a different university, and I drove them sometimes crazy (or they me), they helped me getting the job done. I learned a lot from them, enjoyed fruitful discussions, dinners and conferences. Without their help, I probably would have never understood the idea of category theory or the fact that solutions to problems in software engineering cannot only be expressed in form of algorithms but also in form of mathematical structures that can be constructed in one way or another. Furthermore, I want to thank Gabriele Taentzer to support my decision to do a doctorate and yes, also Adrian Rutle, who convinced me to apply for a position at Høgskolen i Bergen. After meeting him in 2008 as visiting researcher in Marburg, he became a good friend. I remember that I was not so sure at that time if I should really move abroad, leaving my family and friends behind, but finally it was a good decision and I enjoyed living in Norway a lot.

During my doctoral period I have had fruitful research discussions with many other people beside my supervisors. Here, I want to thank in

particular Harald König and Michael Löwe who I visited at FHDW Hannover. Furthermore, I want to thank Juan De Lara from the Universidad Autónoma de Madrid (who also immediately agreed to be my second opponent), Jon Eivind Vatne from Høgskolen i Bergen and Wendy MacCaull at the St. Francis Xavier University in Canada.

I also want to thank Høgskolen i Bergen, the strategic research initiative DISTECH and the FormGrid project for providing a nice working environment, a good salary and great funding, although my research has not been in the focus of Grid computing. In particular, I want to thank our department leader Carsten Helgesen, who always had an ear for our problems and always found a solution. Moreover, I want to thank the Philipps-Universität Marburg, who accepted me as doctoral candidate and Universitetet i Bergen, where I took courses in category theory and the Norwegian language.

In addition, I want to thank my former colleagues at Høgskolen i Bergen and particularly those, I enjoyed teaching with, who are Lars Michael Kristensen, Remy Monsen, Pål Ellingsen and Yngve Lamo as well as our former master students Øyvind Bech, Dag Viggo Lokøen, Anders Sandven, Suneetha Sekhar, Petter Barvik and Ole Klokhammer for their collaborative work. Here, also my fellow (and former) doctoral students should not be forgotten. We shared offices, lunch breaks, discussions about the world or at least the goal of doing a doctorate: Adrian Rutle, Alessandro Rossini (also for being a good flatmate for nearly two years), Piotr Kaźmierczak, Erik Eikeland, Kent Fagerland Simonsen, Hege Erdal, Yi Wang, Xiaoliang Wang, Ajith Admar Kumar Somappa, Atle Loneland, Fazle Rabbi, Bin Wu, Camilla Hanquist Stokkevåg, Truls Pedersen, Thorsten Arendt, Stefan Jurack, Daniel Strüber and Mischa Dieterle. Special thanks to those of them giving me valuable comments on my thesis: Adrian Rutle, Piotr Kaźmierczak and Thorsten Arendt.

Living in Norway I also learned skiing in Høgskolen's alpine group. Therefore I want to thank the members of the alpine group for the nice skiing trips and "lessons" (we started at a red track), and in particular Håvard Skibenes, who took us for several rides to Voss/Myrkdalen.

Embarking on a doctorate is a big decision and I would have had a harder time without my family and friends. Lately I read somewhere in the Internet that «during that time, the "great work" (i.e. the thesis) will hover above the candidate like the sword of Damocles, even in moments of supposed rest». I could not agree more. Therefore, I want to thank my family and in particular my brother Tobias and his wife Suzana as well as my father Harald and his wife Magret for their support. Not less, I want to thank my friends, without them my life would have been boring. First of all, I want to mention my closest friends in Norway, Mattia Natali with whom I went out quite frequently and with whom I played basketball, and Andreas Beck and Emma Bengtsson, who just become

proud parents and liked my pancakes a lot. We had many nice trips and dinners together, thank you especially for this. Also special thanks to Mikal Carlsen Østensen, who helped us, e.g., to furnish an empty apartment. Second, my friends in Germany, Francesco Henning, who was also the only friend managing to visit me in Bergen and Frederick Kämpfer, who also gave me valuable comments on my thesis. In addition, I also want to thank all the other people I did not mention here but made my life as good as it has been as well as the examination board and all those anonymous reviewers who gave me valuable feedback on my articles.

Scientific Environment

The research presented in this thesis has been conducted as part of the FormGrid project (NFR project 194521) in the Department of Computer Engineering at Bergen University College, Norway, as well as within the Department of Mathematics and Informatics at the Philipps-University Marburg, Germany, during several monthly research visits.



Abstract

Model-Driven Engineering (MDE) is a relatively new paradigm in software engineering that pursues the goal to master the increased complexity of modern software products. While software applications have been developed for a specific platform in the past, today they are targeting various platforms and devices from classical desktop PCs to smart phones. In addition, they interact with other applications. To easier cope with these new requirements, software applications are specified in MDE at a high abstraction level in so called models prior to their implementation. Afterward, model transformations are used to automate recurring development tasks as well as to generate software artifacts for different runtime environments. Thereby, software artifacts are not necessarily files containing program code, they can also cover configuration files as well as machine readable input for model checking tools. However, MDE does not only address software engineering problems, it also raises new challenges.

One of these new challenges is connected to the specification of modeling languages, which are used to create models. The creation of a modeling language is a creative process that requires several iterations similar to the creation of models. New requirements as well as a better understanding of the application domain result in an evolution of modeling languages over time. Models developed in an earlier version of a modeling language often needs to be co-adopted (migrated) to language changes. This migration should be automated, as migrating models manually is time consuming and error-prone. While application modelers use ad-hoc solutions to migrate their models, there is still a lack of theory to ensure well-defined migration results.

This work contributes to a formalization of modeling language evolution with corresponding model migration on the basis of algebraic graph transformations that have successfully been used earlier as theoretical foundations of model transformation. The goal of this research is to develop a theory that considers the problem of modeling language evolution with corresponding model migration on a conceptual level, independent of a specific modeling framework.

Abstract (Deutsch)

Die modellgetriebene Softwareentwicklung (MSE) ist ein relativ neues Paradigma in der Anwendungsentwicklung, welches zum Ziel hat die gestiegene Komplexität moderner Softwareprodukte zu meistern. Während Softwareanwendungen früher für eine bestimmte Laufzeitumgebung entwickelt wurden, sollen Anwendungen heute auf unterschiedlichsten Laufzeitumgebung und Plattformen vom klassischen Desktop-PC bis hin zum Smartphone laufen und mit anderen Anwendungen interagieren. Um diese neuen Anforderungen besser beherrschbar zu machen werden Softwareanwendungen in der MSE zunächst vor ihrer Entwicklung auf einem hohen Abstraktionsniveau in sogenannten Modellen spezifiziert. Anschließend kommen Modelltransformationen zum Einsatz, die sowohl dafür genutzt werden können um wiederkehrende Entwicklungsaufgaben zu automatisieren, als auch um Softwareartefakte für verschiedene Laufzeitumgebungen zu erzeugen. Hierbei sind Softwareartefakte nicht zwangsläufig auf Dateien die Programmcode enthalten beschränkt, sondern können auch Konfigurationsdateien sowie maschinenlesbare Problembeschreibungen für Modellverifikationswerkzeuge umfassen. Allerdings geht die MSE nicht nur Softwareentwicklungsprobleme an, sondern sie bringt auch neue Herausforderungen mit sich.

Eine dieser neuen Herausforderungen ist an die Entwicklung von Modellierungssprachen geknüpft, die dazu verwendet werden um Modelle zu erstellen. Die Entwicklung einer Modellierungssprache ist, wie als auch die Entwicklung von Modellen, ein kreativer Prozess, der gewöhnlich nicht nach einer Iteration abgeschlossen ist. Neue Anforderungen sowie ein besseres Verständniss der Anwendungsdomäne, führen dazu, dass Modellierungssprachen im Laufe der Zeit weiterentwickelt werden. Modelle, die in einer früheren Version der Modellierungssprache erstellt wurden, müssen oft anschließend entsprechend angepasst (migriert) werden. Da die manuelle Migration von Modellen zeitaufwendig und fehleranfällig ist, ist es sinnvoll diese Migration zu automatisieren. Während Modellierer ihre Modelle mit ad-hoc Lösungen migrieren, fehlt es noch immer an einer Theorie, die ein wohldefiniertes Migrationsergebnis garantiert.

In dieser Arbeit steht die Formalisierung der Modellierungssprachen-evolution mit korrespondierender Modellmigration auf Basis von alge-

ABSTRACT (DEUTSCH)

braischen Graphtransformationen, welche in der Vergangenheit bereits erfolgreich genutzt wurden um Modelltransformationen theoretisch zu fundieren, im Vordergrund. Ziel der Arbeit ist es eine Theorie zu entwickeln, die das Problem der Modellierungssprachenevolution mit korrespondierender Modellmigration auf einer konzeptuellen Ebene betrachtet unabhängig von einem konkreten Modellierungsframework.

Abstract (Norsk)

Modelldrevet systemutvikling (MDSE) er et relativt nytt paradigme i programvareutvikling som har som målsetting å håndtere den økte kompleksiteten i moderne programvare. Mens programvare tidligere har blitt utviklet for en spesifikk plattform, utvikles det i dag for flere plattformer og enheter, fra skrivebordmaskiner til smarttelefoner. I tillegg skal disse programvarene integreres med andre applikasjoner. For å imøtekomme disse nye kravene, spesifiseres applikasjonene i MDSE på et høyt abstraksjonsnivå i såkalte modeller før de implementeres. Etterpå brukes modelltransformasjoner for å automatisere gjentakende utviklingsoppgaver, så vel som for å generere programvarekomponenter for ulike plattformer. Programvarekomponenter i denne sammenhengen er ikke nødvendigvis bare kjørbare kode, men kan for eksempel også være konfigurasjonsfiler eller maskinlesbar input til et modellverifiseringsverktøy. MDSE løser en del problemer i programvareutvikling, men det skaper også noen nye utfordringer.

En av de nye utfordringene er forbundet med spesifiseringen av de modelleringspråkene som brukes til å definere modellene. I likhet med modelleringsprosessen er utvikling av et modelleringspråk en kreativ prosess som krever mange iterasjoner. Nye krav og bedre forståelse for applikasjonsdomenet, resulterer i en evolusjon av modelleringspråket over tid. Modeller utviklet i en tidligere versjon av et modelleringspråk må ofte tilpasses til endringer (migreres) i modelleringspråket. Denne migreringen bør automatiseres, ettersom manuell tilpasning er ofte tidkrevende og fører til feil i modellene. Modellutviklerne bruker fremdeles ad-hoc løsninger for å migrere modellene sine til de nye versjonene av modelleringspråkene, og dette mangler en teori som sikrer et veldefinert resultat av migreringen.

Denne avhandlingen bidrar til å formalisere evolusjonen i modelleringspråk gjennom tilsvarende modellmigrering basert på algebraisk grafransformasjon, en metode som tidligere har blitt brukt som teoretisk fundament for modelltransformasjoner. Formålet med denne forskningen er å utvikle en teori som hensyntar problemet med at modelleringspråkene utvikler seg over tid, med tilhørende modellmigrering på et konseptuelt nivå uavhengig av det spesifikke modelleringsrammeverket.

Introduction

1.1 Motivation

Model-Driven Engineering [129] (MDE) is a relatively new paradigm in software engineering, which has become an essential part in different application areas during the past years. First, MDE is heavily used in the area of *embedded systems* [82]. For example, models are used to describe the control and data-flow of electrical devices used e.g. in cars. Errors in such systems may be safety-critical or may result in high repair costs. Therefore, their models are (automatically) tested or verified before they are iteratively mapped into hardware or software implementations by means of model transformations. Recently, easy to use tools to specify *domain specific languages* [34, 35, 48, 76, 139] and to generate code from models have been available. Therefore, second, MDE has become popular also in *other* areas, where recurring development tasks need to be solved. This is for example the case if a complex but homogeneous system shall be developed or different applications of the same type e.g. data-driven web applications [2, 148].

Nevertheless, though MDE has become more popular lately, there are still many open problems putting the technology into action. One of these open problems is that software engineers have the desire to develop models and enhance modeling languages in parallel, in particular if they are domain-specific. Modeling languages and models are developed for specific domains and both are improved over time. The challenge occurs with each modeling language evolution step: (usually) models need to be migrated correspondingly (see Figure 1.1). As this challenge is time-consuming and error-prone, it has become a hot topic in MDE research [21, 57, 118, 127, 132]. The result has been that the problem has been studied in

the context of existing modeling frameworks for which initial tool support has been developed.

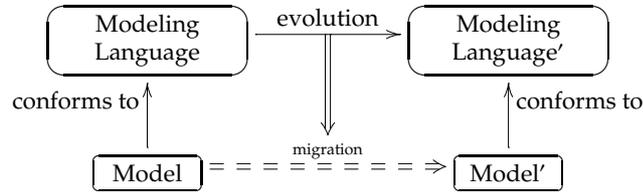


Figure 1.1: Modeling language evolution and model migration

However, MDE is not as mature as other areas in computer science are. Databases [38], compilers [1] or operating systems [126] e.g. are often highly reliable due to decades of research in theory and practice. In particular, comprehensive theories exist for such areas that are consulted and implemented in related tools to ensure specific application/system properties. For example, the development of a compiler is standardized and well-supported by auxiliary tools such as parser generators based on the theory of context-free grammars [19, 20]. For modeling language evolution with corresponding model migration, equivalent strong theories and tools do not exist. The aim of this work, therefore, is to establish a basic theory of modeling language evolution with corresponding model migration that helps to ensure desired migration results and that can be consulted if MDE tools should be built. The vision is that MDE becomes as mature as the mentioned areas one day. Therefore, this thesis focuses on formalization of and studying the problem of modeling language evolution on a formal level, which is valuable for many reasons:

1. First, formalization helps to *understand* a problem. Ambiguous or abstract terms in an area become concrete definitions that help researchers to analyze and discuss problems and solutions efficiently.
2. Second, formalization allows development of solutions to problems that are *exact* and that have *provable* properties. In particular, this aspect contributes to the development of *mature* and *robust* tools.
3. Furthermore, if the abstraction is properly chosen, problems and solutions can be described *independently* from a concrete tool and are therefore *more generally applicable* than specific solutions.

1.2 Problem Statement

For modeling language evolution with corresponding model migration, there is not much theory available. Existing theories classify modeling

language changes [146], define model migration processes on a high abstraction level [127] leaving out *how* models shall be migrated or are only limitedly suitable for model migration [71]. Furthermore, current research focuses on popular modeling frameworks such as the Eclipse Modeling Framework [34] and tool development supporting “best-practice” implementations of model migrations that have manually been developed earlier. It is still open to show under which conditions developed model migrations are well-defined. Moreover, model migrations are typically programmed in a rather low-level language and we are looking for an approach to raise the abstraction level of migration specifications for more convenient migration definition and high reuse facilities. Accordingly, the following requirements must be considered:

1. Migrated models must belong to the evolved modeling language. This property is usually called *soundness*. It subdivides into *well-typedness* and *well-definedness* wrt. language constraints. A migrated model is well-typed if all of its elements are supported elements of the evolved modeling language. Moreover, all well-definedness rules of the evolved language have to be satisfied.
2. All models of the original modeling language can be migrated to the evolved language ensuring that the migration is viable. This property is usually referred to as *completeness*.
3. Model migration should be specified on a *high abstraction level*. This means that a model migration is either automatically deduced from its language evolution or specified using a high-level language.
4. The specification of model migrations is *reusable* (see also [57]). In particular, equivalent migration steps are described only once.
5. General strategies for model migration are formulated *independent of a specific modeling framework*.

1.3 Results

In this thesis, first steps towards a theory for modeling language evolution with corresponding model migration for a specific class of models have been taken. Note, we refer to this type of language evolution with migration in the following by the term *model co-evolution*. Many models are graph-based structures [34, 107, 121] such as class models, state machine models or activity models. This class of models is considered in this thesis and for this class of models a theory that is proposed. In this theory, a formalization of modeling languages and models based on category theory [11] is employed, which allows for describing modeling languages

and models by different types of graphs. To formulate modeling language changes as well as model changes, algebraic graph transformations [36] have been chosen. Algebraic graph transformations are a well-established means to formally underpin model transformations [16, 36]. Moreover, graph transformations have been lifted to high-level structures, not even necessarily being graphs [36]. In this thesis, we present a formal approach to model co-evolution that is first of all based on graphs but can be also applied to other types of high-level structures. The contributions of the thesis are:

1. We present a *formal approach*, where modeling language evolution steps and their model migration steps are specified by coupled graph transformations. This formal setting allows us to reason about *completeness* of model migrations and *well-typedness* of migrated models. Moreover, *well-definedness* is (partly) considered. In particular, the often used *multiplicity constraints* are examined.

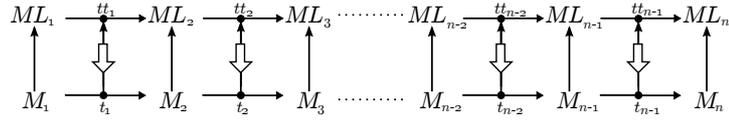


Figure 1.2: Co-Evolution Steps

Modeling language evolution steps, we formulate by graph transformations $tt_i : ML_i \rightarrow ML_{i+1}$. Model migration steps are defined by graph transformation steps $t_i : M_i \rightarrow M_{i+1}$ being typed over tt_i , i.e. M_i is typed over ML_i and M_{i+1} is typed over ML_{i+1} (see Figure 1.2). Note in Figure 1.2, black arrows denote modeling language evolution steps, model migration steps as well as typings. White arrows denote that evolution steps lead to corresponding migration steps (compare Figure 1.1). To migrate a model along several subsequent modeling language evolution steps, all of these steps have to be taken into consideration for model migrations.

Item 1 addresses Items 1 and 2 in the requirements list above. Partly, Item 1 also addresses also Item 5 of the requirements list above, as the approach is formulated on the level of category theory and algebraic graph transformations.

2. To allow model migrations being specified on a high abstraction level and to be reused, we introduce *model migration schemes*. They support the pattern-based definition of model migrations. Once a model migration scheme is specified, it can be applied completely

automatically to models yielding well-typed migrated results. Migration schemes lift the level of migration specification, as they focus on the definition of migration pattern. Pattern recognition and its synchronized replacement does not have to be specified but is defined by the approach. In this context, *reuse* has two dimensions: Migration schemes can be used for (1) different instance models and also (2) for migrating models of different modeling languages being evolved by the same evolution rules.

To increase the level of automation, we show how *default migration schemes* can be derived from given modeling language evolution rules¹ by identifying related model patterns and reflecting modeling language evolution steps on these patterns. Since default migration schemes cannot always reflect the intended semantics of evolution steps, we allow well-defined *customizations* of migration schemes.

Furthermore, migration schemes are used to classify modeling language changes.

Item 2 addresses Items 3 to 5 in the requirements list above.

3. In addition, a first step towards a theory for *detecting a sequence of evolution steps* given two versions of a modeling language is taken. In particular, a procedure based on graph transformations is presented that can be used to find such sequences.
4. Finally, the developed theory and the proposed approach is evaluated. Therefore, a “real” world scenario is considered. In a *case study*, it is examined if the proposed approach is sufficient to express desired changes on a concise and adequate level.

1.4 Overview

The thesis consists of 12 chapters and an additional appendix. The appendix includes one extra chapter on a related topic and proofs of auxiliary propositions. We assume a reader with a basic knowledge of algebraic graph transformation [36] and category theory [11]. The thesis is structured as follows:

- Chapter 2 gives a general introduction into MDE and the subject of modeling language evolution with corresponding model migration.
- Chapter 3 presents different types of graphs that can be used to formalize models supporting various modeling concepts.

¹which are defined manually or automatically

- Chapter 4 abstracts away from graphs and introduces adhesive categories [77] and (weak) adhesive HLR categories [36]. Different types of graphs are classified as adhesive, adhesive HLR or weak adhesive HLR category. Furthermore, propositions and facts used in proofs in subsequent chapters are recalled. In addition, special types of algebraic graph transformations based on cospans are recalled as respectively introduced. In particular, the cospan Double Pushout Approach is recalled [37], while the Sesqui Pushout Approach [24] is transferred to cospan rules.
- In Chapter 5, a new approach to detect sequences of modeling language evolution steps by graph transformation rules is presented with a running example.
- Chapter 6 introduces coupled transformations² as coupled graph transformations. Different constructions are discussed and proven using propositions and facts from Chapter 4. Finally, we decide on a standard construction, which is used in subsequent chapters. Coupled transformations in particular allow to specify type-safe migration steps. Furthermore, the standard construction ensures that migration steps are viable.
- Chapter 7 introduces model migration schemes based on coupled transformations. Model migration schemes allow to specify coupled transformation by migration pattern. Furthermore, a heuristic to deduce default migration schemes from modeling language evolution rules formalized by graph transformation rules on (directed multi-)graphs is explained. Model migration schemes allow well-defined customizations. In addition, the construction of coupled transformations by migration schemes is shown to be complete and sound wrt. to typing.
- Chapter 8 instantiates the framework of coupled transformations and migration schemes for a type of graph supporting inheritance. Inheritance is a well-known concept from object-oriented programming that is frequently used also when defining modeling languages. Further extensions of the framework are presented and proven for these specific types of graphs. Finally, migration schemes are used to classify modeling language changes in the context of a well-known change classification [51].
- Chapter 9 considers coupled transformations and migration schemes wrt. modeling language constraints such as multiplicity constraints known from class models.

²In our previous work also called co-transformations [94, 95, 136].

- In Chapter 10, we reconsider a case-study on activity models to evaluate the approach. The evolution challenge has been used in a tool contest [119] as a real world scenario. We examine if the proposed approach can be used to solve the given evolution problem in a concise and appropriate manner using a suitable type of graph presentation.
- In Chapter 11, related work on modeling language evolution and model migration is presented.
- Chapter 12 concludes and outlines future work.
- In addition, the chapter in the appendix examines the adhesiveness property of categories through an example.

Figure 1.3 shows an activity diagram modeling the proposed approach of the thesis with references to related chapters.

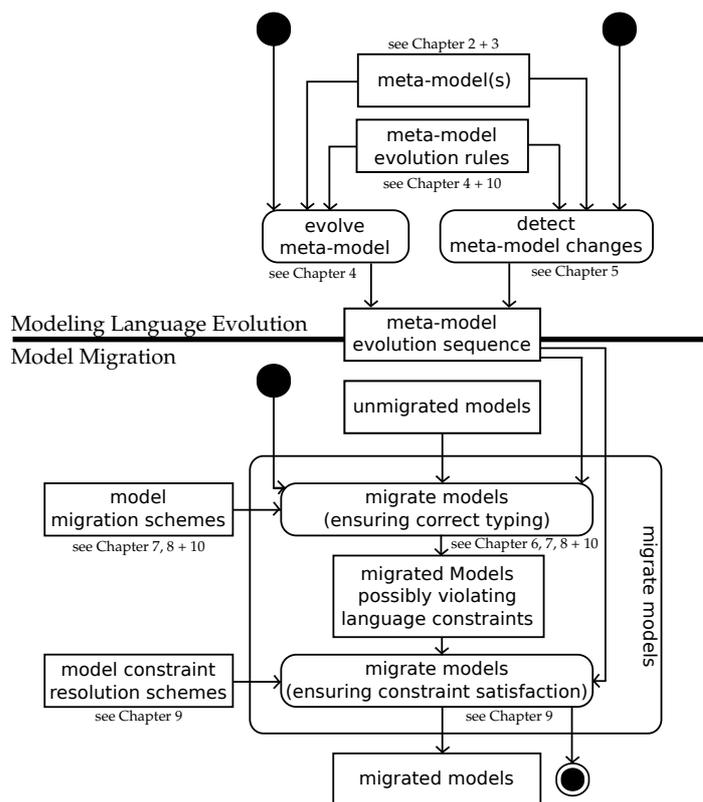


Figure 1.3: Activity diagram: chapter overview

In the upper partition of the activity diagram in Figure 1.3, modeling language evolution is considered. Evolution rules can be used for two purposes: either they are used to evolve a modeling language or they are used to detect changes between two versions of a modeling language.

In the lower partition of the activity diagram in Figure 1.3, model migration is considered. Migrations are two-fold: first, models are adapted respecting typing relations only, second, constraints such as multiplicities are checked and violations are resolved.

The content of this thesis is mostly based on a sequence of publications resulting from joint work with researchers from Philipps-Universität Marburg, Bergen University College and the University of Bergen. We refer to such publications in each chapter.

Model-Driven Engineering

In the following, we recall and discuss some of the fundamental concepts, techniques and standards in model-driven engineering [129]. In particular, the challenge of model migration due to modeling language evolution is explained.

2.1 Introduction

From the beginning of informatics, the diversity and capability of computing devices has been continuously evolving. While only a small number of different computers with limited capability existed in the middle of last century, there exist many different devices, from smart phones to workstations, covering multi-core CPUs today. Furthermore, devices are more and more integrated into different types of networks such as the Internet. Therefore, today's software is more complex, often aims at different platforms and interacts with other applications.

Along with this development, the way how to create software has also been changed. There have been several shifts in programming paradigms; e.g. from machine code to assembler programming and from imperative to object-oriented programming, each step with the purpose to master more complexity. The method to keep pace with this development has always been to raise the abstraction level of the software development techniques.

During the past years, software engineers and computer scientists have integrated the use of models and modeling languages into the development processes. While models initially were introduced for documentation and communication purposes, they are also used as input for code generators and model checkers today. In the literature, this discipline in

EVOLUTION OF
SOFTWARE
DEVELOPMENT

MODEL-DRIVEN
ENGINEERING

informatics is referred to as model-driven engineering (MDE) [68], model-driven development (MDD) [103] and model-driven software development (MDSD) [129]. In this thesis, we adopt the term MDE to denote this discipline.

ADVANTAGES OF
MDE

MDE promotes models as the primary artifacts of the development process and model transformation as the key technique to adapt them for various purposes. Developers with domain knowledge can focus on the problem domain rather than on implementation details. Due to a high abstraction level, models may be easier built, understood, maintained and analyzed [59] than implementations. Technical experts can specify model transformations for repetitive tasks generating different artifacts such as code for different runtime environments. This also affects the quality of the code. Generated code is homogeneous due to the model-to-text transformation and often of good quality, as transformations are usually specified by experienced developers. Furthermore, models can help to detect problems in a software system early by validation and testing. Partly, models are also used to ensure invariants of software systems by translating them into formal problem descriptions that can be verified by model checking tools. In some application domains, such as in the field of embedded systems, these techniques have enhanced productivity, reusability and quality [67].

DISADVANTAGES OF
MDE

However, MDE is not a suitable development approach for all software engineering areas. To obtain benefits from MDE, models need to be at a high abstraction level. This is usually only possible if models specify similar and recurring concepts. Furthermore, to specify models on a high abstraction level, domain specific modeling languages are often built, which implies notable effort to build and maintain the required tool chain around the models. Hence, domain specific modeling language needs to be used for a larger development task or for several projects for this effort to pay off [67]. In addition, MDE is a young software engineering technology, and many tools which exist for programming languages as e.g. for versioning or refactoring are for models only partly available and subject to current research [90, 120].

MODEL-DRIVEN
ARCHITECTURE

In the industry, the idea of MDE has been adopted and standardized as Model-Driven Architecture (MDA) by the Object Management Group (OMG) [103] in late 2000 [43, 69, 104, 111]. The basic ideas of MDA are closely related to generative programming [28], software factories [49] and domain-specific languages [76]. MDA is based on multiple standards, including the Meta-Object Facility (MOF) [110], the Unified Modeling Language (UML) [107], the Object Constraint Language (OCL) [109] and the XML Metadata Interchange (XMI) [108]. A reference implementation of essential MOF is e.g. the Eclipse Modeling Framework (EMF) [34, 131].

2.2 Modeling in Software Engineering

The term *model* is used in many contexts with different meanings. Even in informatics the meaning varies. According to the general model theory by Stachowiak [128], a model is a (1) *representation of an original*, either a physical object or something artificial, (2) *reduced* to its relevant attributes for a specific purpose that can (3) *replace* the original for a particular function or time. In the context of software engineering, a model denotes “an abstraction of a (real or language-based) system allowing predictions or inferences to be made” [74]. In this thesis, we interpret the term model from the software engineering perspective. Furthermore, the models we consider in this thesis are primary *prescriptive* and *graph-based*.

MODEL

A *prescriptive* model specifies aspects of an original that is to be built, e.g., a blueprint of a building. In contrast, a *descriptive* model describes an existing original, e.g., a map of a real city with streets, buildings, etc. In software engineering, models may be both: a model can be used to sketch relevant aspects of a software system to be built (prescriptive) and later serve as documentation (descriptive).

DESCRIPTIVE VS.
PRESCRIPTIVE

Graph-based may also have different meanings depending on the context. In software engineering, a graph denotes a structure that is based on vertices and edges without a concrete visualization. Since graph-based structures are often visualized in a natural way, *graph-based* and *visual* models are often treated as synonyms. In this thesis, we distinguish between these terms. We consider graph-based models independent of an intuitive representation for humans. This in particular means we focus in this thesis on the migration of graph-based structures independent of their visualization. In MDE, this difference is usually taken into account by distinguishing between models in *abstract* and *concrete* syntax.

GRAPH-BASED VS.
VISUAL

Models in abstract syntax consider only domain objects and their relationships, i.e. graphs of information. Models in concrete syntax visualize the information of the model in abstract syntax (see Figure 2.1). Thereby, the concrete syntax defines the physical appearance of language. A visualization of a graph-based model is called a diagram of the model, e.g. a class diagram visualizes a class model [107].¹ Depending on the presentation technique, auxiliary diagram models may be used to store model-specific diagram and layout information. Other presentation techniques, e.g. textual or with fixed layouts such as Nassi-Shneiderman diagrams [101] do not require auxiliary information.

ABSTRACT VS.
CONCRETE SYNTAX

Furthermore, models can be shown in *domain-specific* or *generic* presentations. If models are shown by a generic presentation, e.g. as object diagrams [107] in MOF, the diagram of the model is still called in abstract syntax.

DOMAIN-SPECIFIC
VS. GENERIC
PRESENTATION

¹Note that *diagrammatic* however, may also be used as a synonym for graph-based, e.g. in [121].

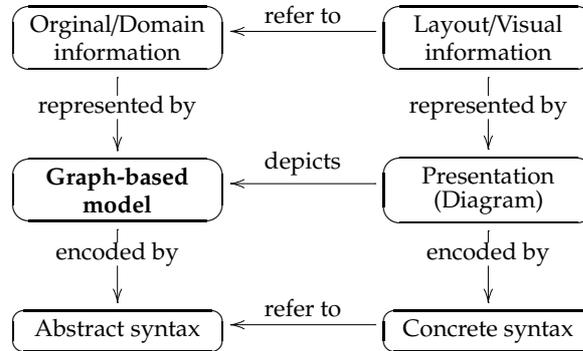


Figure 2.1: Aspects of a model

Remark 2.2.1 (Abstract vs concrete syntax). The terms *abstract syntax* and *concrete syntax* are loosely based on the terms *abstract syntax tree* and *concrete syntax tree* in compiler construction, where the abstract syntax tree can be obtained from the concrete syntax tree by forgetting syntactic sugar elements that are only used to make a program easier to read for humans (i.e. for presentation). In MDE however, these terms may be misleading, as visualizations of models often hide elements of graph-based models in their presentation. Hence, in MDE [129], it is debatable if the concrete syntax is not “more abstract” than the abstract syntax.

GRAPH-BASED
MODELS

Graph-based models [121] have already been adopted in software engineering for some decades; e.g., flowcharts [130] (Seventies) for the description of behavioral properties of software systems; Petri nets [62] (Eighties) for the specification of discrete distributed systems; entity-relationship diagrams [10] (Eighties) for the conceptual modeling of data structures; the unified modeling languages [107] (Nineties) as a collection of several modeling languages for structural and behavioral modeling purposes. While the unified modeling languages (UML) was promoted as a general purpose modeling languages, domain specific modeling languages [42] have become popular these days. Domain specific modeling languages are created to serve a specific scope and help to truly gain advantages in MDE. The state-of-the-art technique to create modeling languages is meta-modeling.

2.3 Meta-modeling

META-MODEL

In MDE, *meta-modeling* [120, 121] is a technique to define a modeling language by a model. Hence, a *meta-model* is a model of a modeling language. The precise definition of the term *meta-model* is frequently debated in the

literature (see [4, 14, 15, 47, 60, 74, 75, 124] for a comprehensive discussion). In particular, there is not a common agreement as to which artifacts belong to a meta-model. In this thesis, a meta-model is a graph-based model defining a modeling language syntax, which restricts the set of valid models (see Figure 2.2).

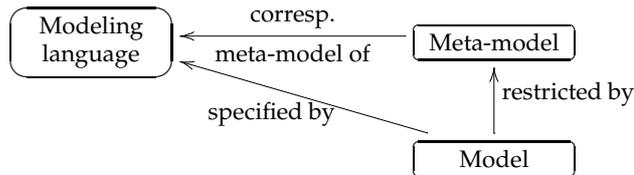


Figure 2.2: Meta-model and model (adopted from [121])

Conceptually, each model can be considered as a modeling language and the pattern can be applied many times (see Figure 2.3). For a more detailed discussion, see [121].

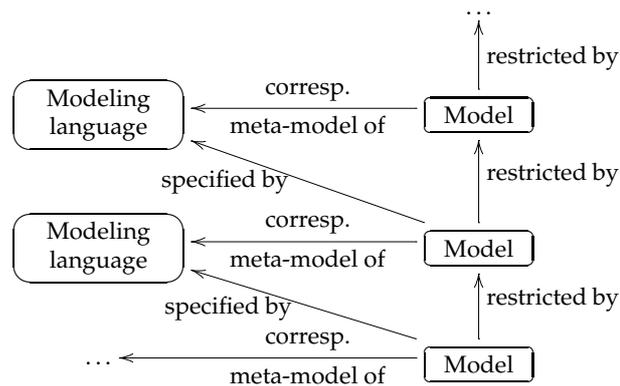


Figure 2.3: Generic pattern: modeling languages and (meta-)models (adopted from [121])

In practice, three level hierarchies are common with a reflexive model at the top (see Figure 2.4). A reflexive model is a model that is at the same time a meta-model for itself. Since model hierarchies are in practice finite, this technique prevents having a model that is not specified by any modeling language. Models of different levels are denoted as *meta-meta-model*, *meta-model* and *model*. In the UML infrastructure specification [107], which bases on Meta-Object-Facility (MOF), the meta-modeling architecture of the OMG, the *original* is added as an additional bottom-level. In the

MODEL
HIERARCHIES

eclipse's MOF implementation EMF, the levels are called *meta-model*, *model* and *instance* instead, inspired by its close relationship to the programming language Java. In this thesis, however, we are considering three level hierarchies and use the terms *meta-meta-model*, *meta-model* and *model* to denote models on different levels. To reflect that a model is restricted by a meta-model, it is also usual to say that a model is an *instance of* its meta-model or an *instance model*.

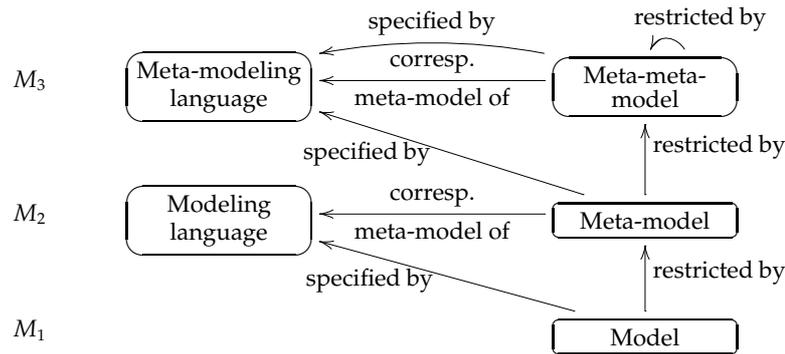


Figure 2.4: 3-layered modeling hierarchy illustrating (meta-)modeling languages and their corresponding models (adopted from [121])

THE MOF MODEL
HIERARCHY

In practice, meta-models are often based on MOF. The MOF meta-meta-model defines a meta-modeling language that allows to specify *class models*. Hence, it is natural to visualize them by *class diagrams*. Models specified by such modeling languages can instead be naturally visualized by *object diagrams*. This means in particular that class and object diagrams used in this thesis are visualizations of models on different levels.

WELL-TYPED
MODEL/TYPED BY

A meta-model restricts the set of valid instance models by their typings (see Example 2.4.1). A model element can only be typed by an element existing in the meta-model. Furthermore, source and target types of elements visualized as edges in an object diagram, such as links, need to be compatible with the source and target elements of their type edge in the corresponding class diagram. In this thesis, we call models that respect this meta-model restriction *well-typed* or *typed by* a meta-model. Moreover, typing the set of valid models can be further restricted by constraints.

2.4 Constraints

A meta-model restricts the set of possible instance models not only by typing. Additionally, meta-models can be annotated by constraints (see

Example 2.4.2). In MOF, some constraint annotations such as multiplicity constraints are integrated in the meta-modeling language, i.e., hard coded in the meta-meta-model. Their semantics are textually described in the MOF specification [110] and verified by model editors. Such annotations we call *integrated constraints*. In addition, MOF also supports *attached constraints* in a textual constraint language called Object Constraint Language (OCL). With OCL, the entire meta-model or model is annotated by a script and models are verified by a OCL checker. For more details, see [121], which also presents an alternate and flexible solution to annotate models by constraints. In this thesis, we call well-typed models that satisfy all constraints *valid instances of* or *conforming to* a meta-model or *well-defined*.

INTEGRATED AND
ATTACHED
CONSTRAINTS

VALID MODEL/
CONFORMS TO

Example 2.4.1 (Meta-model and activity model). Figure 2.5 (a) shows a simplified meta-model for activity models in the usual class diagram visualization. Classes are presented as vertices, while associations are presented as edges. A well-typed instance model visualized as object diagram modeling a simple “research” workflow is shown in Figure 2.5 (b). Figure 2.5 (c) shows the same model visualized as UML activity diagram.

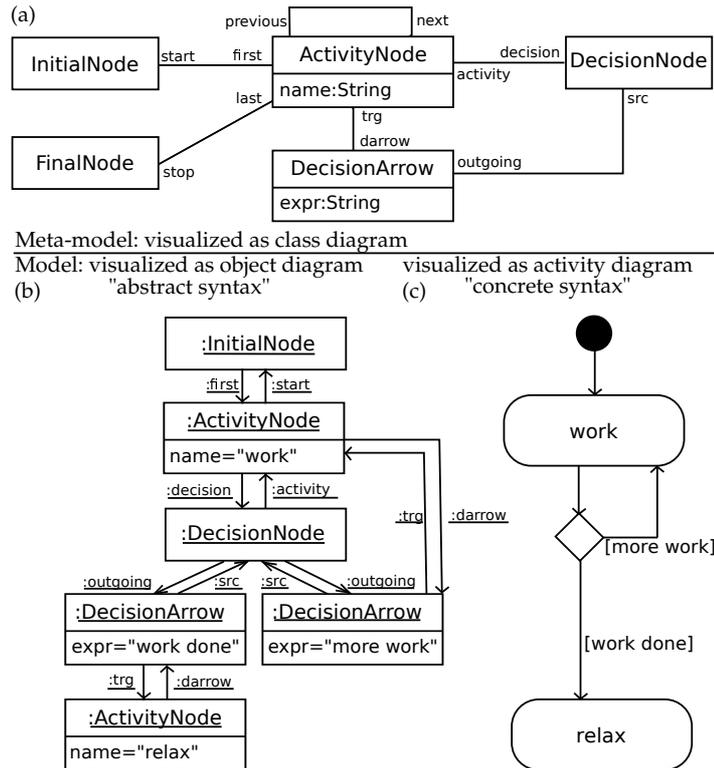


Figure 2.5: A simplified meta-model for activity models with two visualizations of an instance model

The meta-model defines different types of activity nodes that can be connected by different types of arrows. Arrows are modeled as associations beside the arrow that connects the `Decision` with the `Activity` node, which is modeled as class. This arrow is modeled as class, as the arrow needs an attribute for storing an expression. Each *association* represents two opposite *references* and is by default navigable in both directions. Instances of references are called *links* and are only navigable in one direction. The model defines a simple workflow that does not allow a PhD student to relax before the work is done.

Example 2.4.2 (Meta-meta-model and meta-model for activity models). Building upon Example 2.4.1, Figure 2.6 (a) shows a simplified meta-meta-model based on MOF that allows for describing class models. Class models are in MOF the technique to specify meta-models. Figure 2.6 (b) shows the meta-model for activity models used in Example 2.4.1, annotated with multiplicity constraints to allow only well-defined activity models. As multiplicity constraints are not enough to express all desired restrictions, an OCL constraint has been attached in Figure 2.6 (c). In addition, Figure 2.6 shows partially the typing of meta-model elements by typing of meta-model elements by dashed arrows.

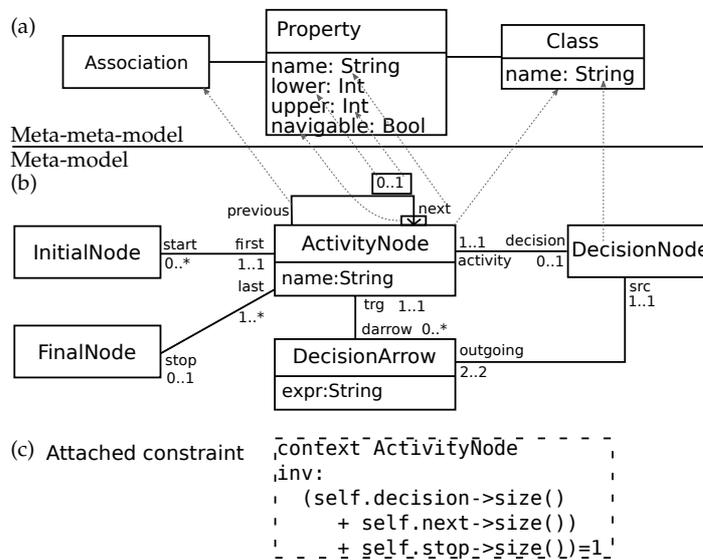


Figure 2.6: A simplified MOF meta-meta-model and an activity meta-model

The meta-meta-model consists of three meta-classes, `Class`, `Property` and `Association`, and two bidirectional associations between the meta-classes. The meta-class `Property` has four *attributes*: `lower`, `upper` to annotate multiplicity constraints and `navigable` to restrict the navigability of associations. In addition, meta-class `Property` and meta-class `Class` each has a `name` attribute. The activity meta-model below is annotated with multiplicity constraints. In addition, edges that are arrows are used to express the navigability of associations that are by default navigable in both directions. In contrast to the activity meta-model in Example 2.4.1, e.g. initial nodes need to be linked to exactly one activity node and decision nodes are required to have exactly two outgoing arrows to activity nodes. Because it is hard to specify with multiplicities that each `Activity` node

instance should have exactly one outgoing arrow of any type, an OCL constraint is attached.

2.5 Model Transformation

The key technique to automate recurring tasks in MDE are model transformations. Model transformations are used for many purposes [125] and can generally be divided into *model-to-text* and *model-to-model* transformations. Model-to-text transformations are, e.g., used to generate programming code, configuration and deployment files. Model-to-model transformations are used e.g. for refactoring [90], language translations, model evolution and model migration. Since software artifacts such as source code can also be considered as a model for a running software, model-to-text transformations can be also considered as model-to-model transformations. A general definition of model transformation can be found in [69, 96]:

MODEL-
TRANSFORMATION
DEFINITION

«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.»

To put it in a nutshell, a model transformation transforms a source model into a target model. While it is highly desired that target models conform to their meta-models, conformance or well-typing of target models are properties that are not always ensured by transformations.

CLASSIFICATION
CRITERIA

There are many existing model transformation approaches and tools available; the OMG for example provides a standard set of model transformation languages called Query View Transformation (QVT) [106]. QVT consists of different *imperative* and *declarative* model transformation languages. While imperative model transformation languages are often more efficient, declarative languages may support bidirectionality and are often theoretically well-founded. Model transformation languages can be further classified along many other criteria [29, 96]. In addition to the already mentioned criteria, model transformations are often classified into *homogeneous* and *heterogeneous* transformations as well as into *in-place* and *out-place* transformations. A more detailed discussion of model transformations can be found in [121].

Homogeneous model transformations transform models specified in one language into models of the same language (see Figure 2.7). Examples

of homogeneous model transformations are model refactorings or meta-model evolution steps that are implemented by transformation rules. In contrast to homogeneous model transformations, heterogeneous model transformations are model transformations that transform models of a source modeling language into models of a target modeling language that is not the same (see Figure 2.8). An example of a heterogeneous model transformation is the transformation from a class model into an entity-relationship model [10]. In addition, the migration of a model after a meta-model change can be considered as heterogeneous model transformation even though most of the modeling language may stay unchanged.

HOMOGENEOUS VS.
HETEROGENEOUS
TRANSFORMATIONS

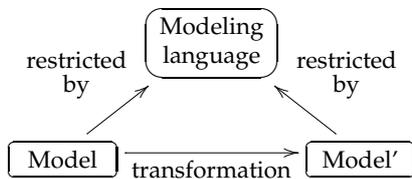


Figure 2.7: Homogeneous model transformation

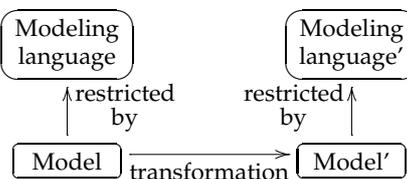


Figure 2.8: Heterogeneous model transformation

In-place transformations [29] are model transformations that transform models by changing an existing model in order to obtain the target model, while out-place transformations construct a new model. Though it is more likely that homogeneous transformations transform in-place and heterogeneous transformation transform out-place, both classifications are orthogonal to each other. There are for example in-place and out-place model migrations approaches. Both types of approaches have advantages and disadvantages. Implementing model migration as in-place transformation may require less adaptations, but it must be ensured that all language elements not existing in the evolved modeling language are migrated or deleted to obtain well-typed target models. Implementing model migration as out-place transformation in contrast may require many rules that copy elements and it must be ensured that no model element may get forgotten.

IN-PLACE VS.
OUT-PLACE
TRANSFORMATIONS

In this thesis we employ *algebraic graph transformations* [36] for model transformations and consider meta-models as *type graphs* and models as *instance graphs*. Algebraic graph transformations have a long history and are theoretically well understood. One algebraic graph transformation is defined by a declarative transformation rule, while *algebraic graph transformation systems* [36] apply transformation rules along different types of control structures [39]. Algebraic graph transformation rules consist of two related graphs, where one can be considered as *precondition* and the

ALGEBRAIC GRAPH
TRANSFORMATIONS

other as *postcondition*. A source graph is transformed into a target graph by matching the precondition and replacing it with the postcondition. Algebraic graph transformations support homogeneous and heterogeneous transformations as well as in-place and out-place transformations. Various tools for graph and model transformation are available, e.g. AGG, AToM³, VIATRA2, GReAT, Henshin [7, 30, 138, 142], which are based on algebraic graph transformations.

2.6 Model Migration

In this thesis, we consider model migrations that are induced by the evolution of their modeling languages, i.e. by the evolution of their meta-models (see Figure 2.9).

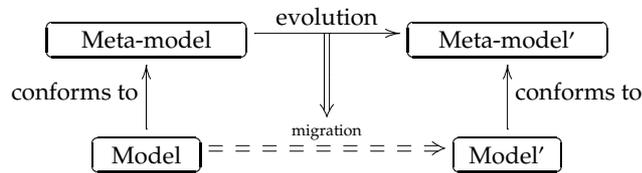


Figure 2.9: Meta-model evolution and model migration

Meta-models are the central artifact of a modeling language and often not only do a large number of models depend on them but also an entire chain of tools (see Figure 2.10). Such tools are e.g. model editors and model transformations that are provided by the language vendor to support the usage of the modeling language.

The engineers that provide a modeling language and build tool support for it are often not the same as those creating models with the language. In this thesis, we call therefore the first group *language engineers* and the second group *application modelers*. For the initial effort that is required to build a modeling language with tool support to pay off, modeling languages have to be used for larger development tasks. This means there are usually more application modelers than language engineers.

Obviously, models depend on meta-models and may require migration after a meta-model has been evolved. If models indeed require migration after a meta-model evolution, such depends on the types of changes that have been made, e.g. if a meta-model is only extended by optional elements, model migration is not required. In literature, such changes are called *non-breaking* in contrast to *breaking* ones, which either need model migration that can be automatically resolved and are hence called *resolvable*, or changes which require further information to be resolved and are hence called *unresolvable* [51].

META-MODEL
DEPENDENCIES

LANGUAGE
ENGINEER VS.
APPLICATION
MODELER

NON-BREAKING VS.
BREAKING
META-MODEL
CHANGES

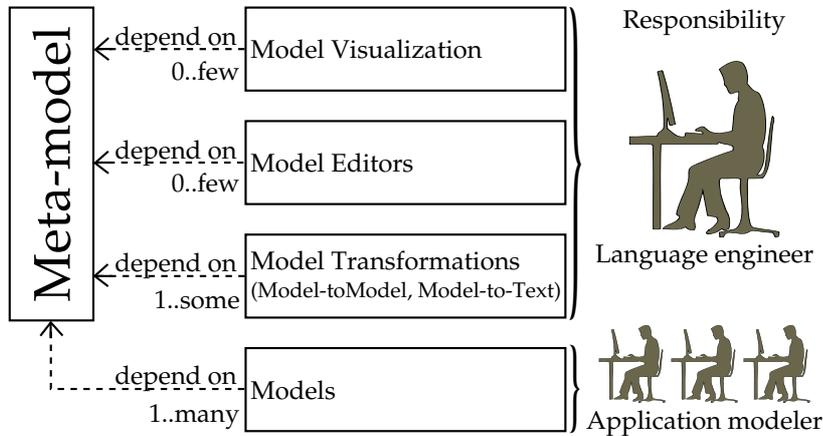


Figure 2.10: A meta-model with depending artifacts

However, not only models may require migration but also the tools supporting the language. Models are presented in editors depending on the types of their elements. Such editors are often developed for the specific modeling language. Hence, such model editors may also require adaptation after a meta-model has been changed, e.g. the visualization of elements may need to be adapted or extended.

Model editors may additionally support special edit operations depending on meta-model patterns or constraints, e.g. a model editor for the activity modeling language as specified by the meta-model in Example 2.4.2 may enforce to create objects of type `DecisionArrow` together with two edges of type `src`, respectively, `trg` so that the whole pattern can be visualized as one arrow linking two nodes. Another edit operation may require adding decision nodes only together with two outgoing arrows so that the defined multiplicity constraints are satisfied.

However, in contrast to the models that need to be migrated there is a difference: the modeling language engineers have full access to the tools they provide and there are only a few central artifacts to be changed. In addition, a modeling language editor may not always require adaptation after a meta-model change. Modeling frameworks such as EMF offer generic model editors with generic model visualizations. Also in this thesis, models are most often visualized using a generic visualization such as object diagrams or graphs. In addition, there are domain specific modeling languages for editor generation such as GMF [48], so that editors may be regenerated from evolved meta-models.

Furthermore, model-to-text as well as model-to-model transformations rely on meta-models, as transformation definitions transform models according to their typing. If models are not used for documentation only

MODEL EDITOR
MIGRATIONMODEL
TRANSFORMATIONS
MIGRATION

there are some model transformations e.g. for code generation that may require migration. However, in practice there are most likely substantially fewer model transformations than models [59]. In addition, model transformations are often developed by the modeling language provider together with the modeling language so that the modeling language provider has full control of their migration. This in particular is true for domain specific modeling languages.

MODEL MIGRATION

Considering the different artifacts that may require adaptation, the most time consuming and error-prone task is to migrate and maintain existing models (see Figure 2.9) after the corresponding modeling language has been evolved [59]. Therefore, it is highly desirable to automate model migration by model transformation. In addition, this transfers the control of how models are migrated back to the language engineers. This is desirable as model migrations are supposed to respect the intended semantics of the corresponding meta-model evolution change. Since the semantics of models are often equivalent to generated source code, it is advantageous to migrate models in close correspondence with the adaptations made to the code generator. Example 2.6.1 shows a simple Petri net example of a meta-model evolution step with corresponding model migration. In the meta-model, a reference is replaced by a class with two corresponding references. While in the sample model, this meta-model change requires only one adaptation (i.e. one edge needs to be replaced by a vertex and two edges), the required adaptations multiply with occurrences of links typed by the replaced meta-model reference. If larger Petri net models or many Petri net models need to be migrated many adaptations are required.

Example 2.6.1 (Petri net meta-model evolution and model migration). Figure 2.11 shows a meta-model evolution step in a simplified Petri net meta-model together with a corresponding model migration.

Figure 2.11 (a) shows a simplified meta-model for Petri net models in the usual class diagram visualization, while Figure 2.11 (b) shows an evolved version of the same meta-model. The meta-model in Figure 2.11 (a) defines `Place` and `Transition` as classes and one arrow going from `Place` to `Transition`, `pTArrow`, and one vice versa, `tPArrow`, as navigable references. In the evolved meta-model in Figure 2.11 (b), reference `tPArrow` has been replaced by a class `TPArrow` with two references going to `Place`, respectively `Transition`. In contrast to the previous meta-model, the evolved meta-model supports weights for arrows from transitions to places. This meta-model evolution step requires the migration of the simple instance model shown in Figure 2.11 (c). The evolved model is presented in Figure 2.11 (d). The model in Figure 2.11 (c) shows a simple Petri net consisting of only two places, one token and one transition, in Figure 2.11 (d) the outgoing link of the only transition is replaced by an object and two links. The weight of this new object presenting the arrow

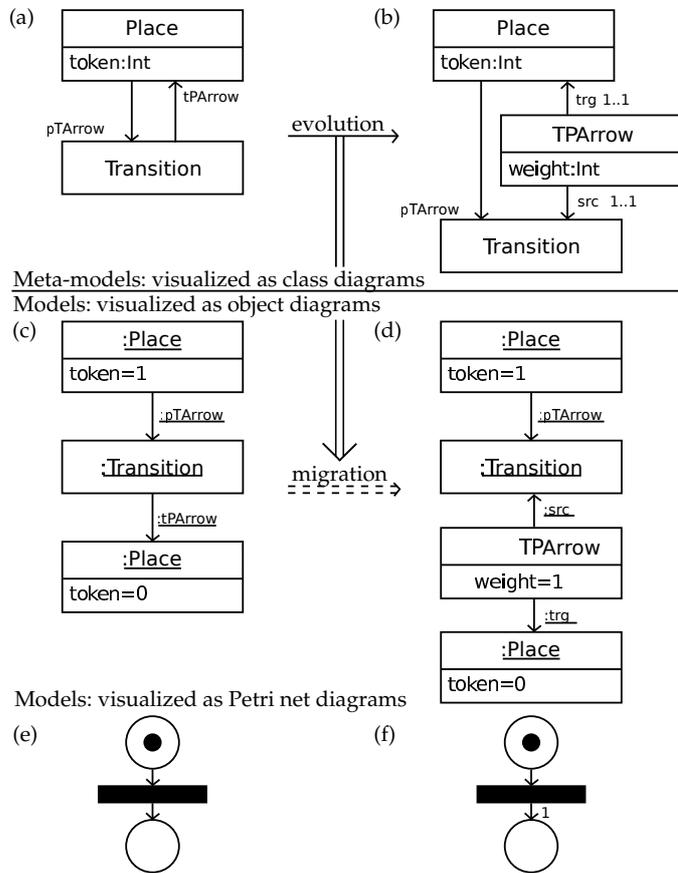


Figure 2.11: Petri Net meta-model evolution and model migration with different visualizations

from transition to the place is one. One has been chosen, since one has implicitly been the weight of all edges before the meta-model evolution step. Figure 2.11 (e) and Figure 2.11 (f) visualize the models of Figure 2.11 (e) and Figure 2.11 (f) in the classical Petri net presentation.

Fortunately, co-evolution of structures has been considered in several areas of computer science such as for database schemes and grammars [79, 85]. Database schema evolution has especially been a subject of research in the past decades [8].

CO-EVOLUTION

However, while for grammar and program co-evolution (syntax) trees need to be co-evolved, model co-evolution needs to consider graph structures. In general, graph structures do not have a fixed root node and

cannot be traversed as easily as trees. Hence, the challenge of grammar co-evolution obviously differs from the challenge of model co-evolution. Database schema evolution can be considered more similar.

Database schema evolution has been studied for relational as well as for object-oriented databases. An early survey on existing approaches for databases can be found in [116]. In particular, co-evolution in object-oriented databases has some similarities to model co-evolution in MDE. However, even the challenge of database schema evolution with data migration is quite similar to the challenge of meta-model evolution, with instance model migration there are differences:

- For example, there are *different important technological challenges*. While models are typically held in main memory, database tables often cannot be. Furthermore, the acceptance of downtimes of database servers is often very low. Therefore, performance issues may be more important for database schema evolution.
- There are *different types of depending artifacts*. While, for example, model editors, model presentations, code templates or model transformations depend on meta-models, SQL queries and stored database procedures depend on database schema. Data are retrieved from databases via *queries*. In MDE instead, models are typically used for code generation purposes where the whole model is processed. Hence, query rewriting seems to be more important for databases. In MDE, model transformations play a central role and have to be considered accordingly. Though these tasks may also require query rewriting, as transformation languages may be query-based e.g. based on OCL, this is not generally the case. There are other transformation approaches such as algebraic graph transformations not relying on queries. However, this thesis focuses on meta-model evolution and model migration only.
- Database schema evolution often tackles *one database instance* only while many models of the same language typically need to be migrated in MDE. Such models may not even be accessible at the same time due to the fact that models are created asynchronously from different application modelers possibly working at different companies. This means database schema evolution can often adequately be applied stepwise together with the corresponding data migrations on the basis of transactions, i.e., under control of the database management system. Instead, models are preferably migrated asynchronously in batch jobs. Therefore, we consider it as more important that model migrations are viable after meta-model evolution.

(META-)MODEL
CO-EVOLUTION VS.
DATABASE SCHEMA
CO-EVOLUTION

- Model structures may be *more complex*. Often, one element in a model presentation visualizes an entire pattern of model elements. Therefore, it may be more important to replace whole patterns in model migration than in data migration. In addition, models are not restricted to *statical properties* of software, they may also describe behavior. The BPEL meta-model [102], for example, also contains classes expressing behavior such as if-statements and loops. Therefore, it may also be more important to consider the context of elements and to make more elaborated case distinctions in model co-evolution.
- The *coupling* between database schema and data is usually tighter. Therefore, data migration steps are often straightforward and can be deduced, e.g. if two associated tables are merged, their data values need to be copied accordingly to a table join operation. In model co-evolution, meta-models and models are not so tightly coupled. Model migrations often need to be individually defined in correspondence with the language engineer's intentions of the meta-model changes.
- Meta-models use *different types of constraints* than database schema. While primary, foreign key and simple data value constraints are the most used constraints in relational databases [26], meta-models often make use of multiplicity constraints. Database schema instead rarely restrict the number of associated database records in a database relationship.

Recently, there has been more focus on meta-model evolution and model migration and how to adapt schema evolution concepts for models (see e.g. [21, 57, 127]).

2.6.1 General Approaches to Model Migration

Current model migration approaches can be classified into *manual specification*, *operator-based* and *meta-model matching* approaches [118]:

- *Manual specification approaches* [84, 118, 127] consider two meta-model versions as given and migrate models (out-place) by copying elements from the previous model version to the new one. Elements automatically copied are those that have unchanged or compatibly changed types. New elements of a meta-model have to be taken into account in manually defined migration specifications.
- In *operator-based* approaches [57, 143], a meta-model is evolved using pre-defined operators. The evolution history is tracked as a sequence of changes. Usually, a library of coupled evolution-migration operators is supported to stepwise evolve meta-models and migrate models accordingly.

CLASSIFICATION OF
MODEL MIGRATION
APPROACHES

- *Meta-model matching* approaches [22, 45] consider two versions of a meta-model as given. An evolution history i.e. a sequence of evolution steps, is (semi-)automatically derived from the difference of two meta-model versions. Afterward, all detected meta-model evolution steps are (semi-)automatically mapped to predefined migration operations.

While existing approaches are mainly driven by automating the adaptation, which application modelers would have done ad-hoc, in this thesis we focus on a formalization of model migration due to meta-model evolution. This contributes to a clear understanding of the model co-evolution challenge. In addition, a generic and sound approach for model migration is presented.

2.6.2 Correctness of Model Migrations

While model co-evolution approaches are usually not formally founded, there are two approaches that do consider formal frameworks: In [71], König et al. present a formal framework to data migration based on category theory. It can also be used as a formal basis for model migration with fixed migration strategies for meta-model refactorings only. For this setting, information-preservation for migrations is shown. In [84], Levendovszky et al. reason about termination and confluence of model migrations but do not consider completeness and soundness. We are heading towards a general approach for model co-evolution that ensures completeness and soundness of model migrations as the two main properties for their well-definedness.

2.6.3 Reusability of Model Migrations

Reusable co-evolution operators for meta-model evolution was first proposed by Wachsmuth [146], who tried to combine ideas from object-oriented refactorings with grammar adaptation. In [22, 45], a predefined set of meta-model evolution operations can be used to detect meta-model changes. An extensive catalog of reusable evolution operators being structured by different categories such as structural primitive operators, operators dealing with inheritance, and delegation, can be found in [58].

When pre-defined evolution operators cannot be used, it is typical for most approaches that model migrations are specified manually. Automatic deduction of migration specification typically supports the preservation and deletion of model elements as supported by Epsilon Flock [118].

2.6.4 Customization of Model Migrations

Customization of model migration specifications is usually supported in an ad-hoc manner only: considering Cope/Edapt [57] as one representative for operator-based approaches, it supports the implementation of migration operations as Groovy/Java programs. For customization, the modeler has to adapt these programs. In [22, 45], migration scripts are generated from recognized evolution operations using the textual model transformation language ATL. Customization of model migration scripts can be done on the level of such ATL scripts.

To summarize, for all informal approaches, there are no criteria to ensure that migration scripts remain to be consistently defined to their meta-model evolutions after customization. It is also not ensured that all instance models can still be migrated after customization. The formal framework to data migration presented by König et al. [71] has a fixed construction of migration specifications that cannot be customized yet. Hence, we are heading towards automatic deduction of customizable model migrations while still ensuring the completeness and well-typedness properties of model migrations.

In this chapter, the basic concepts of MDE have been summarized and in particular the challenge of model co-evolution has been discussed.

Graph-based Modeling

The standard meta-modeling architecture defined by the Object Management Group (OMG) [103] is Meta-Object Facility (MOF) [110]. MOF provides a meta-modeling language to specify class structures. Many modeling languages are MOF-based today, but other modeling frameworks as e.g. the Kernel Meta-meta-model (KM3) [64] or Microsoft DSL Tools also provide meta-modeling languages for class structures. Therefore, we focus on class modeling here. Models defined by such modeling languages can be considered as object models. Hence, meta-models can be visualized as class diagrams and models as object diagrams, respectively. Since class and object models are graph-based, it is natural to formalize them by graphs. In this chapter, we recall definitions from the literature and provide examples for different types of graphs and how they can be used as formalization of meta-models and models.

3.1 Graphs supporting Attribution

In the literature, different formalizations of models by different types of graphs have been proposed (e.g. [36, 54, 89, 121]), varying in the supported modeling concepts and their formalization. A simple and widely used formalization for models are directed multi-graphs, short graphs here:

Definition 3.1.1 ((Directed multi-)graph). A (directed multi-)graph $G = (G_V, G_E, src^G, trg^G)$ consists of a set G_V of vertices (or nodes), a set G_E of edges (or arrows), and two maps $src^G, trg^G : G_E \rightarrow G_V$ assigning the source and target to each edge, respectively. $e : x \rightarrow y$ denotes an edge with $src^G(e) = x$ and $trg^G(e) = y$.

(DIRECTED
MULTI-)GRAPH

3. GRAPH-BASED MODELING

MODELING WITH
(DIRECT
MULTI-)GRAPHS

The main concepts of class models used in MDE are *classes*, *references* and *attributes*. Bidirectional associations can be encoded as association classes or two opposite references similar to EMF. It is straightforward to encode classes as vertices and references as edges. Analogously object models can be formalized by graphs encoding objects as vertices and links as edges. Attributes can be supported by encoding data types respectively attribute values as vertices and attributes respectively value assignments as edges. Graph morphism can be employed to relate graphs. Note that we are using “;” to denote the composition of morphism, i.e. $f;g = g \circ f$. We prefer this notation as it allows us to trace morphisms along the direction of their arrows in corresponding diagrams.

GRAPH MORPHISM

Definition 3.1.2 (Graph morphism between (directed multi-)graphs). A graph morphism $\phi : G \rightarrow H$ consists of a pair of maps $\phi_V : G_V \rightarrow H_V$, $\phi_E : G_E \rightarrow H_E$ preserving the graph structure: for each edge $e : x \rightarrow y$ in G we get an edge $\phi_E(e) : \phi_V(x) \rightarrow \phi_V(y)$ in H , i.e., $src^G; \phi_V = \phi_E; src^H$ and $trg^G; \phi_V = \phi_E; trg^H$.

$$\begin{array}{ccc}
 G_V & \xrightarrow{\phi_V} & H_V \\
 \uparrow & & \uparrow \\
 src^G & = & src^H \\
 \downarrow & & \downarrow \\
 G_E & \xrightarrow{\phi_E} & H_E
 \end{array}
 \qquad
 \begin{array}{ccc}
 G_V & \xrightarrow{\phi_V} & H_V \\
 \uparrow & & \uparrow \\
 trg^G & = & trg^H \\
 \downarrow & & \downarrow \\
 G_E & \xrightarrow{\phi_E} & H_E
 \end{array}$$

Graphs and graph morphisms constitute the category of graphs:

CATEGORY
Graph

Definition 3.1.3 (Category of graphs). The category **Graph** has all graphs G as objects and all graph morphisms $\phi : G \rightarrow H$ as morphisms. The composition $\phi; \psi : G \rightarrow K$ of two graph morphisms $\phi : G \rightarrow H$ and $\psi : H \rightarrow K$ is defined component-wise $\phi; \psi = (\phi_V, \phi_E); (\psi_V, \psi_E) := (\phi_V; \psi_V, \phi_E; \psi_E)$. The identity graph morphisms $id^G : G \rightarrow G$ are also defined component-wise $id^G = (id^{G_V}, id^{G_E})$. This ensures that the composition of graph morphisms is associative and that identity graph morphisms are left and right neutral with respect to composition.

TYPING

While graphs are already sufficient to specify basic models, more concepts are desired. One of these concepts is the concept of typing, which is essential to describe meta-models and instance models. Typing restricts the set of valid instance model of a modeling language. Elements of different types are distinguished. In addition, graph structures are restricted i.e. edges of a specific type can only connect elements of specific types. While a meta-model can be formalized by a distinguished graph, the typing relationships between meta-models and models can be formalized by a graph morphism:

3.1. Graphs supporting Attribution

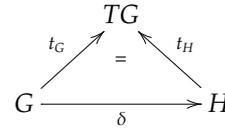
Definition 3.1.4 (Type graph and instance graph). A *type graph* is a distinguished graph $TG = (TG_V, TG_E, src^{TG}, trg^{TG})$. A typed graph $(G, t_G : G \rightarrow TG)$ that is typed by TG is a graph G together with a graph morphism $t_G : G \rightarrow TG$. A typed graph $(G, t_G : G \rightarrow TG)$ is also called an *instance graph* of graph TG and the morphism t_G is called a *typing morphism*.

TYPE GRAPH AND
INSTANCE GRAPH

If instance graphs are related by graph morphism, the relationship needs to respect the typing:

Definition 3.1.5 (Typed graph morphism).

Given a type graph TG , a typed graph morphism $\delta : (G, t_G : G \rightarrow TG) \rightarrow (H, t_H : H \rightarrow TG)$ is a graph morphism $\delta : G \rightarrow H$ such that $t_G = \delta; t_H$.



TYPED GRAPH
MORPHISM

Again, Graphs and graph morphisms constitute a category:

Definition 3.1.6 (Category of typed (directed multi-)graphs). Graph_{TG} is defined as the slice category (Graph/TG) . The category Graph_{TG} has all typed graphs $(G, t_G : G \rightarrow TG)$ as objects and all typed graph morphisms $(G, t_G : G \rightarrow TG) \rightarrow (H, t_H : H \rightarrow TG)$ as morphisms. The composition of two typed graph morphisms as well as the identity typed graph morphisms $id^{(G, t_G : G \rightarrow TG)} : (G, t_G : G \rightarrow TG) \rightarrow (G, t_G : G \rightarrow TG)$ is defined component-wise analogously to (untyped) graph morphisms (see Definition 3.1.3). This ensures that the composition of graph morphisms is associative and that identity graph morphisms are left and right neutral with respect to composition.

CATEGORY
 Graph_{TG}

Remark 3.1.1 (Typed graphs). Note that there is a faithful functor from category Graph_{TG} to category Graph . This means type and typed graphs can always also be considered as (untyped) graphs and typed graph morphisms and typing morphisms as (untyped) graph morphisms. The concept of typing naturally extends also to other types of graphs.

Example 3.1.1 (A modeling hierarchy formalized in category Graph). Figure 2.5 (I-III) shows a simplified meta-meta-model, a meta-model for Petri nets and a Petri net model, Figure 2.5 (a-d) shows corresponding formalizations by graphs. Since the graphs are related by morphisms, each graph can also be considered as type graph for the instance graph on the level below in category Graph_{TG} .

The left model hierarchy of Figure 2.5 shows a meta-meta-model for class structures supporting the concepts: classes, attributes and references. It can be reflexively typed by itself: `Class` and `DataType` can be typed by `Class` and `reference` and `attribute` by `reference`. The meta-model and model below shows the simple Petri net meta-model and model of Example 2.4.1 (a) and (c). The graph hierarchy to the right of Figure 2.5

3. GRAPH-BASED MODELING

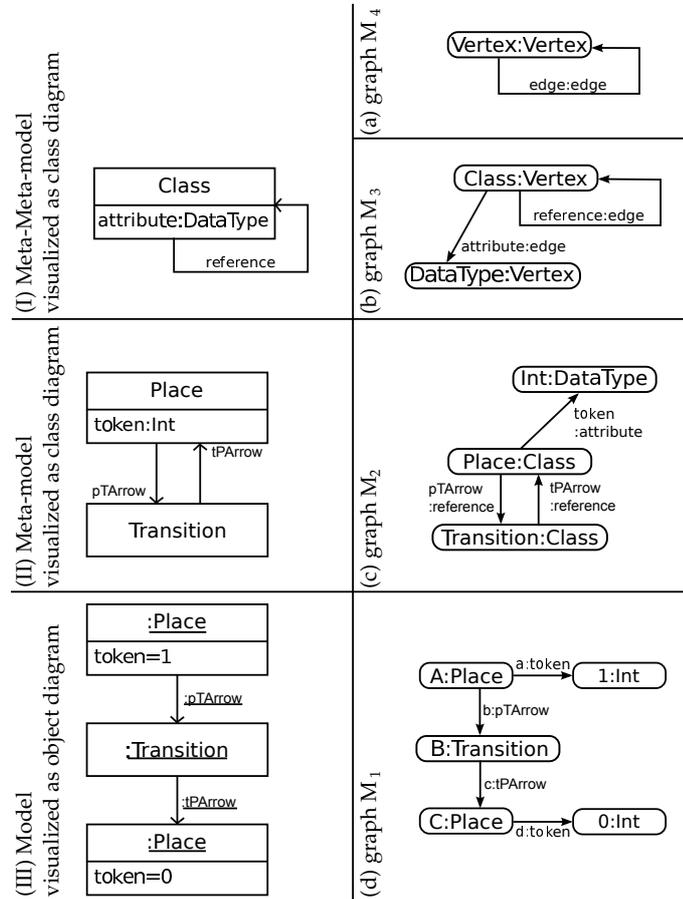


Figure 3.1: A model hierarchy and its formalization by graphs

shows the models formalized by graphs. While the model hierarchy in the left of Figure 2.5 uses three levels, an additional level (a) has been added in the graph hierarchy to the right of Figure 2.5. This has the advantage that the hierarchy starts with the unit graph, i.e. the terminal object in Graph as defined in Definition 3.1.1 ($\text{Graph} \cong \text{Graph}_1 = (\text{Graph}/TG)$ with 1 being the terminal object in Graph). While the typing of elements is implicit given by the visualization of elements in the left hierarchy, it is given in the traditional “identifier:type identifier” notation in the right hierarchy. Note that attribute values are identifier, which are data values of their corresponding type. A smaller font size has been used for edge labels only for space reasons.

Remark 3.1.2 (Element names and element identifier). Note that meta-models usually also have meta-attributes i.e. for denoting a type name. This is neglected in many examples presented here (see e.g. Example 3.1.1) for simplicity reasons. Instead, element identifiers are used as type names. This means that vertex identifiers are considered as names, edge identifiers consist of the edge name together with the names of the source and the target vertex. In practice, however, elements should have both, a unique identifier and a name given by a meta-attribute. Changing the name of an element, in particular if it is a type, should not effect its identity. On the formal level, this distinction is less important, as transformation results in algebraic graph transformations are only defined up to isomorphism. Changing a type name does not change the graph structure.

One important concept of class modeling is attributes. In Example 3.1.1, attributes have already been used and formalized on the level of (directed multi-)graphs. Usually, each possible data value is considered as special vertex i.e. no data value exists twice. This restriction is common in graph transformations as transformation results are conceptually distinguished up to isomorphism only, which is undesired when working with data values. If attribute values should also be manipulated, other types of graphs are needed, such as symbolic graphs [112], as summarized in the following. Symbolic graphs are defined based on so called \mathcal{E} -graphs [36]. An \mathcal{E} -graph is a type of labeled graph, where both vertices and edges may be decorated with labels from a given set E . Labels are hereby considered by a special class of vertices and the labeling relationship (vertex \rightarrow label, edge \rightarrow label) by special types of edges.

MODELING WITH
SYMBOLIC GRAPHS

Definition 3.1.7 (\mathcal{E} -graph). An \mathcal{E} -graph over the set of labels L is a tuple $G = (G_V, G_L, G_E, G_{NL}, G_{EL}, \{src_j^G, trg_j^G\}_{j \in \{NL, NE\}})$ consisting of:

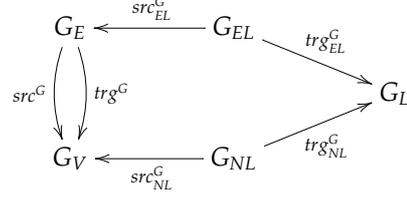
- G_V and G_L , which are the sets of graph vertices and of label vertices, respectively.
- G_E , G_{NL} and G_{EL} , which are the sets of graph edges, vertex label edges, and edge label edges, respectively.

and the source and target functions:

\mathcal{E} -GRAPH

- $src^G : G_E \rightarrow G_V$ and $trg^G : G_E \rightarrow G_V$
- $src_{NL}^G : G_{NL} \rightarrow G_V$ and $trg_{NL}^G : G_{NL} \rightarrow G_L$
- $src_{EL}^G : G_{EL} \rightarrow G_E$ and $trg_{EL}^G : G_{EL} \rightarrow G_L$

3. GRAPH-BASED MODELING



\mathcal{E} -GRAPH
MORPHISM

Definition 3.1.8 (\mathcal{E} -Graph morphism). Given the \mathcal{E} -graphs G and H , an \mathcal{E} -graph morphism $\phi : G \rightarrow H$ is a tuple, $(\phi_V : G_V \rightarrow H_V, \phi_L : G_L \rightarrow H_L, \phi_E : G_E \rightarrow H_E, \phi_{NL} : G_{NL} \rightarrow H_{NL}, \phi_{EL} : G_{EL} \rightarrow H_{EL})$ such that ϕ commutes with all of the source and target functions.

Also, \mathcal{E} -graphs and \mathcal{E} -graph morphism constitute a category (see [36]):

CATEGORY EGraph

Definition 3.1.9 (Category of \mathcal{E} -graphs¹). All \mathcal{E} -graphs G and all \mathcal{E} -graph morphisms $\phi : G \rightarrow H$ form the category of \mathcal{E} -graphs EGraph.

Labels in \mathcal{E} -graphs may be substituted, e.g. by data values as used in symbolic graphs:

LABEL
SUBSTITUTION IN
 \mathcal{E} -GRAPHS

Definition 3.1.10 (Label substitution in \mathcal{E} -graphs). Given an \mathcal{E} -graph $G = (G_V, G_L, G_E, G_{NL}, G_{EL}, \{src_j^G, trg_j^G\}_{j \in \{NL, NE\}})$, a set of labels H_L , and a function $h : G_L \rightarrow H_L$, we define the \mathcal{E} -graph $h(G)$ resulting from the substitution of G_L along h as $h(G) = H = (H_V, H_L, H_E, H_{NL}, H_{EL}, \{src_j^H, trg_j^H\}_{j \in \{NL, NE\}})$ with:

- $H_V = G_V, H_E = G_E, H_{NL} = G_{NL}, H_{EL} = G_{EL}, \{src_j^H = src_j^G\}_{j \in \{NL, NE\}},$ and $trg^H = trg^G$
- $trg_{NL}^H = trg_{NL}^G; h$
- $trg_{EL}^H = trg_{EL}^G; h$

Moreover, h induces the definition of the \mathcal{E} -graph morphism $h^* : G \rightarrow h(G)$, with $h^* = (id_V, h, id_E, id_{NL}, id_{EL})$.

To manipulate data values, data algebras are used that can be formalized by algebraic signatures:

ALGEBRAIC
SIGNATURE

Definition 3.1.11 (Algebraic signature). An algebraic signature $\Sigma = (S, \Omega)$ consists of a set of sorts S and a family of operation symbols Ω of the form (op, s_1, \dots, s_n, s) , where $n \geq 0$ and $s_i, s \in S$.

Σ -ALGEBRA

Definition 3.1.12 (Σ -algebra). A Σ -algebra \mathcal{A} consists of a S -indexed family of sets $\{A_s\}_{s \in S}$ and a function $op_{\mathcal{A}} = A_{s_1} \times \dots \times A_{s_n} \rightarrow A_s$ for each $op = s_1 \times \dots \times s_n \rightarrow s \in \Omega$.

¹Note that EGraph is simply the functor category of the minimal equivalence graph (MEG) shown in Definition 3.1.7 into Set (i.e. $[MEG \Rightarrow Set]$).

3.1. Graphs supporting Attribution

Example 3.1.2 (Algebraic signature and Σ -algebra). An example algebra Σ_{NAT} considers only natural numbers, boolean values and the operations $+, \wedge$ and \geq . The signature is given by:

Sorts: Nat, Bool
Operations: (add, Nat, Nat, Nat)
 (and, Bool, Bool, Bool)
 (geq, Nat, Nat, Bool)

The Σ_{NAT} -algebra is defined by:

$\mathcal{A}_{Bool} = \{true, false\} = \mathbb{B}$
 $\mathcal{A}_{Nat} = \mathbb{N}$
 $\mathcal{A}_{add} = + : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
 $\mathcal{A}_{and} = \wedge : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$
 $\mathcal{A}_{geq} = \geq : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$

For example, $exp = 1 + 1 \geq 2$ is a valid expression in the algebra above.

Algebras also constitute categories.

Definition 3.1.13 (Σ -algebra morphism). A Σ -algebra morphism $\phi : \mathcal{A} \rightarrow \mathcal{A}'$ consists of a S -indexed family of functions $\{\phi_s : A_s \rightarrow A'_s\}_{s \in S}$ commuting with the operations.

Σ -ALGEBRA
MORPHISM

Definition 3.1.14 (Category Alg_Σ). Σ -algebras and Σ -algebra morphisms constitute category- Alg_Σ .

CATEGORY Alg_Σ

Expressions in the algebra are defined to be in the same equivalence class if their evaluation leads to the same result:

Definition 3.1.15 (Σ -algebra congruence). A congruence \equiv on an algebra \mathcal{A} is a S -indexed set of equivalence relationships $\{\equiv_s\}_{s \in S}$, which are compatible with the operations. \mathcal{A}/\equiv denotes the quotient algebra whose elements are equivalence classes of values in \mathcal{A} . Between \mathcal{A} and \mathcal{A}/\equiv there is a canonical morphism mapping every element in \mathcal{A} into its equivalent class.

Σ -ALGEBRA
CONGRUENCE

Symbolic graphs combine the concepts of \mathcal{E} -graphs and Σ -Algebras:

Definition 3.1.16 (Symbolic graph). A symbolic graph over the data Σ -algebra \mathcal{D} , with $\Sigma = (S, \Omega)$, is a pair (G, Φ_G) , where G is an \mathcal{E} -graph over a S -sorted set of variables $X = \{X_s\}_{s \in S}$, i.e. $G_L = \cup_{s \in S} X_s$ and Φ is a set of first-order Σ -formulas built over the free variables in X and including the elements in \mathcal{D} as constants.

SYMBOLIC GRAPH

Definition 3.1.17 (Symbolic graph morphism). Given symbolic graphs (G, Φ_G) and (H, Φ_H) over the same data algebra \mathcal{D} , a symbolic graph morphism $h : (G, \Phi_G) \rightarrow (H, \Phi_H)$ is an \mathcal{E} -graph morphism $h : G \rightarrow H$ such that $\mathcal{D} \models \Phi_H \implies h^\#(\Phi_G)$, where $h^\#(\Phi_G)$ is the set of formulas obtained when replacing in Φ_G every variable x_G in the set of labels of G by $h_L(x_G)$.

SYMBOLIC GRAPH
MORPHISM

CATEGORY
SymbGraph \mathcal{D}

Definition 3.1.18 (Category of symbolic graphs). Symbolic graphs over \mathcal{D} together with their morphisms form the category SymbGraph \mathcal{D} .

If every label of a symbolic graph is associated with a variable bound to a value, the graph is called *grounded*.

GROUNDING
SYMBOLIC GRAPH

Definition 3.1.19 (Grounded symbolic graph). A symbolic graph (G, Φ_G) over a data algebra \mathcal{D} is grounded if

1. G_L includes a variable, which we denote by x_v , for each value $v \in \mathcal{D}$,
2. and for every substitution $\phi : G_L \rightarrow \mathcal{D}$, such that $\mathcal{D} \models \phi(\Phi)$, we have $\phi(x_v) = v$, for each variable $x_v \in G_L$.

CATEGORY
GSymbGraph \mathcal{D}

Moreover, we define category GSymbGraph \mathcal{D} as the full subcategory of category SymbGraph \mathcal{D} consisting of all grounded graphs.

Models usually do not contain variables but attribute values. Therefore they may be formalized by grounded graphs. Graphs used in transformation rules instead may require variables and might be formalized by graphs which are not grounded (see Example 3.1.3).

Example 3.1.3 (Symbolic graph morphism in category SymbGraph \mathcal{D}). Figure 3.2 (a) and (b) show symbolic graphs. In contrast to the symbolic graph in Figure 3.2 (a), the symbolic graph in Figure 3.2 (b) is grounded.

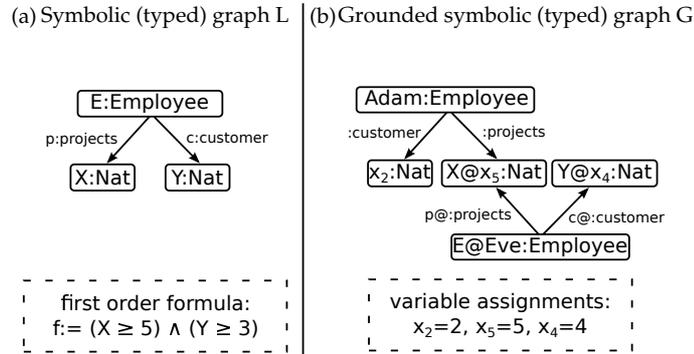


Figure 3.2: A (typed) symbolic graph morphism

In Figure 3.2 (a) characters have been used to denote element identifier. In Figure 3.2 (b) element identifiers have been neglected. We also leave them out in following examples where they are unimportant. Furthermore, element identifiers are used (prefixed by @) in Figure 3.2 (b) to denote morphism $m : L \rightarrow G$ by mapping elements. Note that only this mapping binds the variables of the given first order formula f in Figure 3.2 (a) so that f evaluates to true.

3.2 Graphs supporting Inheritance

Beside classes, references and attributes, there are more concepts that class models support, e.g. operations and interfaces. While operation signatures and interfaces can also be formalized straightforwardly using special types of vertices, it is frequently discussed in the literature about how the concept of inheritance can be formalized e.g. [36, 54, 89, 121, 122]. Some ideas are presented in the following.

Already typed (directed multi-)graphs are sufficient to formalize meta-models supporting inheritance. Inheritance edges in meta-models can be modeled as edges of specific type in the type graph. Models that use inherited features can be modeled by instance graphs that model subclasses as composition of particles similar to [122]. In such a formalization, types that present subclasses always have to be instantiated together with their superclasses and the corresponding inheritance edges (see Example 3.2.1). Tools that use such structures to store models internally can present them without particles by using an adequate visualization.

INHERITANCE BY
PARTICLES

Example 3.2.1 (Vertex inheritance in category Graph). Figure 3.3 (a) shows the Petri net meta-model of Example 3.1.1 extended by a superclass as a type graph. Figure 3.3 (b) shows an instance graph using particles to formalize an object typed by a subclass.

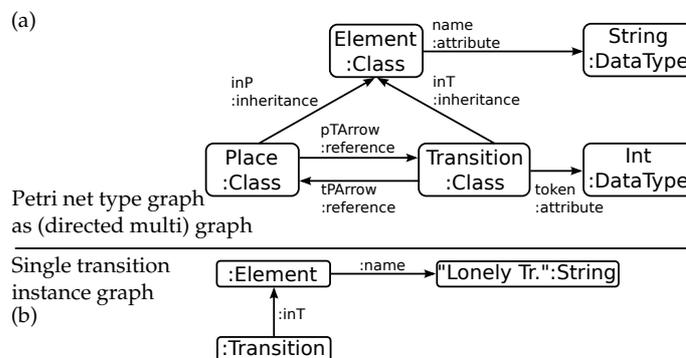


Figure 3.3: A type graph for a Petri net meta-model and an instance graph using particles to model inheritance

The type graph of the previous Petri net meta-model is extended by vertex Element. Vertices Place and Transition “inherit” a name attribute. The instance graph in Figure 3.3 (b) shows a single transition with name “Lonely Tr.” using particles.

INHERITANCE IN
DPF

An alternative formalization of the inheritance concept based on (directed multi-)graphs is presented in [121]. This formalization is called the Diagram Predicate Framework (DPF) [120, 121]. DPF builds on previous work on Generalized Sketches [32, 33, 150]. In DPF, inheritance relationships are again modeled by edges of specific type in type graphs. However, inheritance edges have specific semantics: edges formalizing references or attributes are automatically copied to all corresponding vertices formalizing subclasses. In contrast to the formalization before, models can be formalized without using particles. Inherited features can be typed by the corresponding “copied” edge types. The appendix of this thesis contains an additional Chapter B, where this work is generalized in a case study on adhesive categories [77]. In the following, basic DPF concepts are recalled from this chapter and [121]. A central concept in DPF is signatures. Signatures allow for annotating graphs by constraints. Herein, annotations are defined on the basis of (directed multi-)graphs that specify the scope of a constraint. Additionally, each constraint is named by a predicate symbol.

(DPF) SIGNATURE

Definition 3.2.1 ((DPF) Signature). A (DPF) signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$ consists of a set of predicate symbols Π^Σ and a mapping α^Σ that assigns a (multi-)graph to each predicate symbol $\pi \in \Pi^\Sigma$. $\alpha^\Sigma(\pi)$ is called the *arity* of the predicate symbol π .

A (directed multi-)graph that is annotated by constraints from a signature is called a generalized specification:

 (DPF)
GENERALIZED
SPECIFICATION

Definition 3.2.2 ((DPF) generalized specification). Given a (DPF) signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$, a generalized specification $\mathfrak{S} = (S, C^\mathfrak{S}, \text{sym}^\mathfrak{S}, \text{dia}^\mathfrak{S})$ consists of a multi-graph S , a set $C^\mathfrak{S}$ of “constraint identifiers” and two maps $\text{sym}^\mathfrak{S} : C^\mathfrak{S} \rightarrow \Pi^\Sigma$ and $\text{dia}^\mathfrak{S} : C^\mathfrak{S} \rightarrow \text{Graph}_0$, such that the diagram below commutes. Herein, Graph_0 denotes the set of objects in the category Graph and $(\text{Graph}/S)_0$ denotes the slice category as usual.

$$\begin{array}{ccc}
 & C^\mathfrak{S} & \\
 \text{sym}^\mathfrak{S} \swarrow & & \searrow \text{dia}^\mathfrak{S} \\
 \Pi^\Sigma & (=) & (\text{Graph}/S)_0 \\
 \alpha^\Sigma \searrow & & \swarrow \text{fst}^S \\
 & \text{Graph}_0 &
 \end{array}$$

For the definition of morphisms, we have to remind that any graph morphism $\phi_G : S \rightarrow S'$ induces a functor $\overline{\phi}_G : (\text{Graph}/S) \rightarrow (\text{Graph}/S')$ with $\overline{\phi}_G; \text{fst}^{S'} = \text{fst}^S$, which is defined by simple post-composition, i.e., $\overline{\phi}_G(\gamma) := \gamma; \phi_G$ for all objects $\gamma : G \rightarrow S$ in Graph/S .

3.2. Graphs supporting Inheritance

Definition 3.2.3 (Morphisms between generalized (DPF) specifications). Given two generalized (DPF) specifications $\mathfrak{S} = (S, C^{\mathfrak{S}}, sym^{\mathfrak{S}}, dia^{\mathfrak{S}})$ and $\mathfrak{S}' = (S', C^{\mathfrak{S}'}, sym^{\mathfrak{S}'}, dia^{\mathfrak{S}'})$, a specification morphism $f = (f_C, f_G) : \mathfrak{S} \rightarrow \mathfrak{S}'$ is given by a mapping $f_C : C^{\mathfrak{S}} \rightarrow C^{\mathfrak{S}'}$ and a graph morphism $f_G : S \rightarrow S'$, such that the following two diagrams commute:

MORPHISMS
BETWEEN
GENERALIZED
SPECIFICATIONS

$$\begin{array}{ccc}
 C^{\mathfrak{S}} & \xrightarrow{f_C} & C^{\mathfrak{S}'} \\
 \searrow^{sym^{\mathfrak{S}}} & & \swarrow_{sym^{\mathfrak{S}'}} \\
 & \Pi^{\Sigma} &
 \end{array}
 \qquad
 \begin{array}{ccc}
 C^{\mathfrak{S}} & \xrightarrow{f_C} & C^{\mathfrak{S}'} \\
 \downarrow^{dia^{\mathfrak{S}}} & & \downarrow^{dia^{\mathfrak{S}'}} \\
 (\text{Graph}/S)_0 & \xrightarrow{\overline{f_G}} & (\text{Graph}/S')_0
 \end{array}$$

In addition, generalized specifications and generalized specification morphism constitute a category:

Definition 3.2.4 (Category of generalized (DPF) specifications). Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, the category $\text{GSpec}(\Sigma)$ has all generalized specifications $\mathfrak{S} = (S, C^{\mathfrak{S}}, sym^{\mathfrak{S}}, dia^{\mathfrak{S}})$ as objects and all generalized specification morphisms $\phi : \mathfrak{S} \rightarrow \mathfrak{S}'$ as morphisms between generalized specifications \mathfrak{S} and \mathfrak{S}' .

CATEGORY
 $\text{GSpec}(\Sigma)$

The composition $\phi; \psi : \mathfrak{G} \rightarrow \mathfrak{R}$ of two (generalized) specification morphisms $\phi : \mathfrak{G} \rightarrow \mathfrak{S}$ and $\psi : \mathfrak{S} \rightarrow \mathfrak{R}$ is defined component-wise $\phi; \psi = (\phi_C, \phi_G); (\psi_C, \psi_G) := (\phi_C; \psi_C, \phi_G; \psi_G)$. The identity (generalized) specification morphism $id^{\mathfrak{G}} : \mathfrak{G} \rightarrow \mathfrak{G}$ is also defined component-wise $id^{\mathfrak{G}} = (id^{C^{\mathfrak{G}}}, id^G)$. This ensures that the composition of specification monomorphisms is associative and that identity specification morphisms are left and right neutral with respect to composition.

Naturally, the category of generalized (DPF) specifications can be restricted by typing constituting the category of typed generalized (DPF) specifications analogously as we have seen before. The semantics of an inheritance edge is formalized by a set of universal constraints on typed specifications.

Definition 3.2.5 (Universal constraint). Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, a universal constraint is a typed specification morphism $c : (\mathfrak{L}, t_L : L \rightarrow G) \rightarrow (\mathfrak{R}, t_R : R \rightarrow G)$ with $\mathfrak{L} = ((L, t_L : L \rightarrow G), C^{\mathfrak{L}}, sym^{\mathfrak{L}}, dia^{\mathfrak{L}})$ and $\mathfrak{R} = ((R, t_R : R \rightarrow G), C^{\mathfrak{R}}, sym^{\mathfrak{R}}, dia^{\mathfrak{R}})$.

(DPF) UNIVERSAL
CONSTRAINT

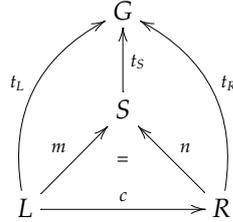
The universal constraints are assigned to a specification S formalizing a meta-meta-model and are required to be satisfied in all specifications formalizing meta-models:

Definition 3.2.6 (Satisfaction of universal constraints). A typed specification $(\mathfrak{S}, t_S : S \rightarrow G) = ((S, C^{\mathfrak{S}} : \Sigma), t_S : S \rightarrow G)$ satisfies a universal constraint

SATISFACTION OF
UNIVERSAL
CONSTRAINTS

3. GRAPH-BASED MODELING

$c : (\mathcal{Q}, t_L : L \rightarrow G) \rightarrow (\mathcal{R}, t_R : R \rightarrow G)$ iff, for any typed specification morphism $m : (\mathcal{Q}, t_L : L \rightarrow G) \rightarrow (\mathcal{S}, t_S : S \rightarrow G)$, there is a typed specification morphism $n : (\mathcal{R}, t_R : R \rightarrow G) \rightarrow (\mathcal{S}, t_S : S \rightarrow G)$, such that $c; n = m$.

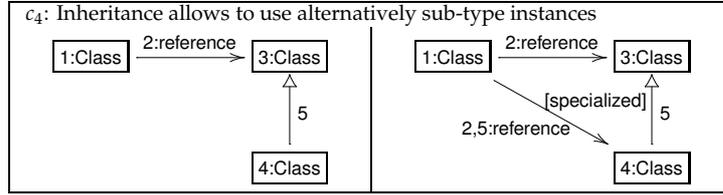


We call $(\mathcal{Q}, t_L : L \rightarrow G)$ and $(\mathcal{R}, t_R : R \rightarrow G)$ input and output patterns of the constraint c , respectively; and we call m and n matches of the patterns $(\mathcal{Q}, t_L : L \rightarrow G)$ and $(\mathcal{R}, t_R : R \rightarrow G)$ in $(\mathcal{S}, t_S : S \rightarrow G)$, respectively.

The universal constraints to enforce that all attributes and references are properly inherited by subclasses are shown in Table 3.1 (in concrete syntax). The universal constraints also use two atomic constraint annotations. The annotation `[inherited]` denotes that the edge presents a specific inherited edge. This correspondence is encoded in the identifier of the edge, which consists of the identifiers of the inheritance relationship and the inherited reference. Analogously, the atomic constraint `[specialized]` denotes that there is also a reference to a specific subclass of a referenced class.

Table 3.1: Universal constraints for modeling inheritance (adapted from [121])

$(\mathcal{Q}, t_L : L \rightarrow S)$	$(\mathcal{R}, t_R : R \rightarrow S)$
<p>c_1: Inheritances are transitive</p>	
<p>c_2: Inheritance leads to inheritance of attributes</p>	
<p>c_3: Inheritance leads to inheritance of references</p>	



Example 3.2.2 (Vertex inheritance in *category* GSpec). This example shows the same models as Example 3.2.1 using DPF. Figure 3.4 (a) shows a Petri net meta-model, while Figure 3.4 (b) shows an instance graph.

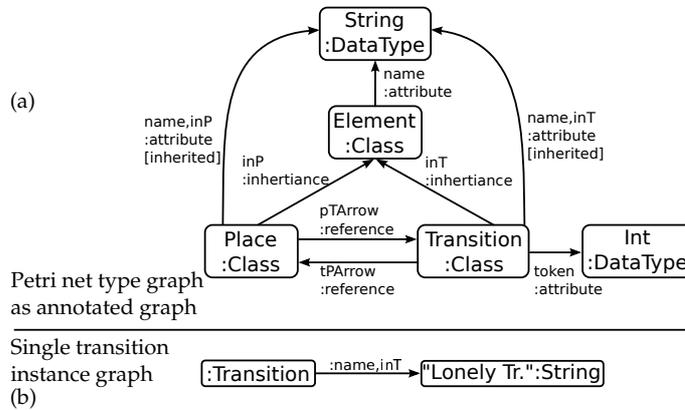


Figure 3.4: A type graph for a Petri net meta-model with instance graph in DPF

In the meta-model of Figure 3.4, universal constraint c_2 in Table 3.1 has been used to ensure that attribute `name` of class `Element` has been properly specified for its subclasses `Place` and `Transition`. The instance graph in Figure 3.4 (b) shows a single transition with the name “Lonely Tr.”, using a correspondingly copied edge.

A fundamental different formalization of models with inheritance is presented in [54]. A similar formalization can also be found in [89]. Instead of considering the inheritance concept on the level of graphs only, it is considered by changing the definition of typing morphisms. In [54], such graphs are called \mathcal{J} -graphs (see Example 3.2.3). An \mathcal{J} -graph extends a (directed multi-)graph by a set of inheritance relationships between vertices that preorder vertices. This in particular means every graph is also an \mathcal{J} -graph. In the following, we call a vertex A as usual in object-oriented programming a *sub*-vertex if it inherits from another vertex B , while we call vertex B a *super*-vertex of vertex A . The inheritance relation is transitive and the set of sub-vertices of a vertex is called its clan. Multiple inheritance

is allowed. However, in contrast to the concept of traditional inheritance, \mathcal{J} -graphs allow cycles in inheritance relationships, i.e. a vertex can be both a *sub* and *super* vertex of another vertex:

\mathcal{J} -GRAPH

Definition 3.2.7 (\mathcal{J} -graph). A graph with inheritance, short \mathcal{J} -graph, is given by $GI = (G, I)$. It consists of graph G and inheritance relation $I \subseteq G_V \times G_V$, where for $v \in G_V$ $clan_I(v) = \{v' \in G_V \mid (v', v) \in I^*\}$ with I^* being the reflexive and transitive closure of I . A vertex $v' \neq v$ with $v' \in clan(v)$ is called a *sub-vertex* of v or more *specifically* as v ($v' \leq v$), vertex v we call a *super vertex* of v' .

The typing relationships of “objects” by “classes” can be specified by clan morphisms. A clan morphism defines a mapping from a (directed multi-)graph to an \mathcal{J} -graph. Clan morphisms are based on graph morphisms, but weaken the homomorphism condition. Hence, every graph morphism is also a clan morphism. In particular, clan morphisms allow to type edges connecting two “object” vertices by edges connecting super vertices of their current source and target type.

CLAN MORPHISM

Definition 3.2.8 (Clan morphism). Given graph $G1$ and \mathcal{J} -graph $GI2 = (G2, I2)$, a pair of mappings $f = (f_V, f_E) : G1 \rightarrow G2$ is called clan-morphism, written $f : G1 \rightarrow GI2$, if $\forall e1 \in G1_E : (src^{G1}; f_V)(e1) \in clan_{I2}((f_E; src^{G2})(e1)) \wedge (trg^{G1}; f_V)(e1) \in clan_{I2}((f_E; trg^{G2})(e1))$.

Example 3.2.3 shows an example of an \mathcal{J} -graph and a (directed multi-)graph typed by a clan-morphism.

Example 3.2.3 (Vertex inheritance in *category* $\mathcal{I}Graph$). This example shows the same models as Example 3.2.1 and Example 3.2.2 using a (typed) \mathcal{J} -graph as formalization of the meta-model (Figure 3.5 (a)). The instance graph is still formalized by a (directed multi-)graph Figure 3.5 (b). Hence, the typing morphism is a clan-morphism (see Definition 3.2.8).

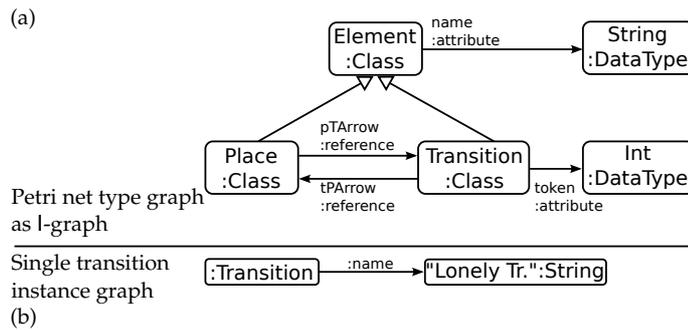


Figure 3.5: A type graph for a Petri net meta-model with instance graph in $\mathcal{I}Graph$

The instance graph in Figure 3.4 (b) shows a single transition with the name “Lonely Tr.” using the original attribute from the Petri net meta-model. This is possible because the typing morphism is no longer a graph morphism but a clan morphism. Note that the edges with hollow arrowheads in the type graph present inheritance relationships.

While clan-morphisms are morphisms between (direct multi-)graphs and \mathcal{J} -graphs, morphisms between \mathcal{J} -graphs are called \mathcal{J} -graph morphisms. \mathcal{J} -graph morphism based on clan-morphism and especially every clan-morphism is also an \mathcal{J} -graph morphism. Additionally, \mathcal{J} -graph morphisms have to ensure that inheritance relationships are preserved in the image of the morphism. This means the following: if two vertices are in the same clan in the pre-image of a morphism, their images are also required to be in the same clan.

Definition 3.2.9 (\mathcal{J} -graph morphism). Given \mathcal{J} -graphs $GI1 = (G1, I1)$ and $GI2 = (G2, I2)$, an \mathcal{J} -graph morphism $f : GI1 \rightarrow GI2$ is given by a clan-morphism $f : G1 \rightarrow G2$, which is \mathcal{J} -compatible, i.e. $(v, w) \in I1$ implies $(f(v), f(w)) \in I2^*$.

\mathcal{J} -GRAPH
MORPHISM

\mathcal{J} -graphs and \mathcal{J} -graph morphism constitute the category of \mathcal{J} -graphs [54]:

Definition 3.2.10 (Category of \mathcal{J} -graphs). All \mathcal{J} -graphs as objects and all \mathcal{J} -graph morphisms constitute the category of \mathcal{J} -graphs \mathbf{IGraph} . The composition of \mathcal{J} -graph morphisms $f : GI1 \rightarrow GI2$ and $g : GI2 \rightarrow GI3$ is defined by $f;g : GI1 \rightarrow GI3$ with $(f;g)_V = f_V;g_V : G1_V \rightarrow G3_V$ and $(f;g)_E = f_E;g_E : G1_E \rightarrow G3_E$.

CATEGORY OF
 \mathcal{J} -GRAPHS

3.3 Graphs supporting Language Constraints

Another frequently used concept in modeling is language constraints that restrict the set of valid instance models beyond typing. Such constraints are either integrated into the language or attached (see Chapter 2). The literature provides different solutions to equip model formalizations with constraints [36, 112, 121]. In the following, different solutions to *annotate* graph formalizations by constraints are presented. The semantics of such annotations are often not formally defined and given by validator programs in modeling tools. However, one way to formalize constraint semantics is graph constraints [113]. A generalization of graph constraints is DPF’s universal constraints, which have been presented before (see Definition 3.2.5).

MOF implements constraint annotation e.g. for multiplicities by the attribution of elements. Since meta-model edge types may also require

3. GRAPH-BASED MODELING

constraint attributes, MOF models various edge types by vertex meta-types (see Example 2.4.2). A different visualization is used to e.g. present associations, respectively references in meta-models by edges while they are typed by a class in the meta-meta-model. This idea can be reused to formalize constraint annotations in (directed multi-)graphs. Without changing the visualization, references could also be formalized in graphs presenting meta-models each with one vertex and two edges. Analogously links in graphs presenting models would be presented. A sample model formalized by graphs using this technique is shown in Example 3.3.1.

Example 3.3.1 (A model with multiplicity constraints in category Graph). Figure 3.6 shows a meta-meta-model supporting multiplicity constraints together with a simple Petri net meta-model as (directed multi-)graphs.

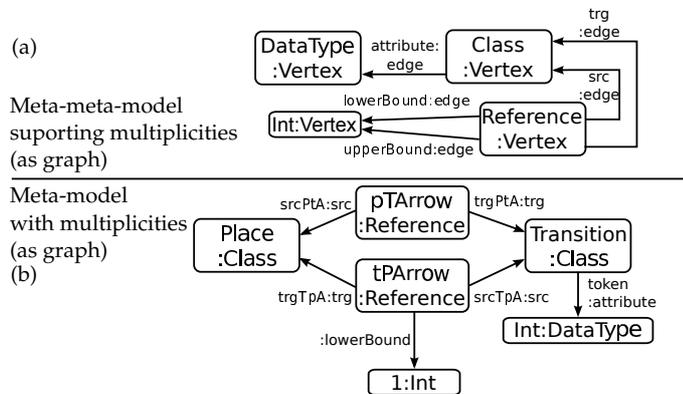


Figure 3.6: A meta-meta-model and Petri net meta-model formalized by (directed multi-)graphs

The Petri net meta-model below uses the `lowerBound` multiplicity attribute to annotate that transition instances require at least one outgoing arrow.

Alternatively, different types of graphs such as \mathcal{E} -graphs (presented above, see Definition 3.1.7) can be used to annotate graphs with constraints more directly. \mathcal{E} -graphs have the advantage that edge types of class diagrams must not be formalized by vertex types in type graphs, as edges can be annotated directly. Therefore, this formalization is more compact and closer to their traditional presentation as class and object diagrams. Example 3.3.2 shows a graph using edge attribution to support multiplicity constraints.

Example 3.3.2 (A model with multiplicity constraints in category EGraph). Figure 3.7 shows the models of Example 3.3.1 using (typed) \mathcal{E} -graphs instead of (directed multi-)graphs.

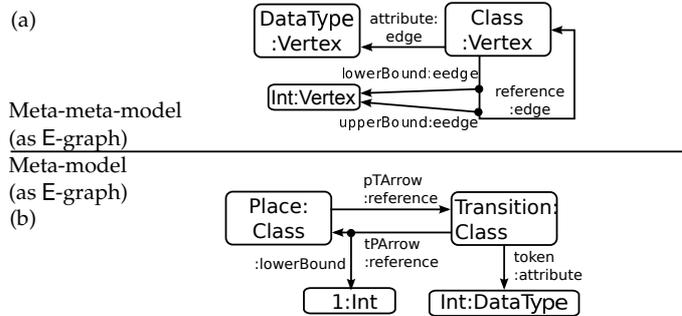


Figure 3.7: A meta-meta-model and Petri net meta-model formalized by attributed graphs with edge and vertex attribution

In addition to integrated constraint there are also attached constraints used in modeling. While integrated constraints are specified by setting meta-attribute values that are used in corresponding validation routines of a modeling tool, attached constraints are usually specified by use of a textual constraints language such as OCL (see Example 2.4.2). However, it is hard to incorporate textual constraints with their models on a formal level. Therefore, we prefer the annotation technique of DPF, which has been summarized in the last section. It has been shown in [120, 121] that DPF can be used in many cases where traditionally OCL is used. Models on all levels can be annotated by constraint predicates using a graph morphism from a shape graph to a sub-graph (of the model). Example 3.3.3 shows a graph using DPF's [120, 121] atomic constraints (see also Example 3.2.2). Atomic constraint annotations can be specified for arbitrary shape graphs. Their semantics can be defined using a suitable approach. This means that how atomic constraints are validated in DPF is not prescribed. For example, validation programs can be implemented for each predicate, which validate each subgraph of an instance graph typed by $(\text{Graph}/S)_0$ for each constraint identifier. Such subgraphs can be obtained by pullbacks of $G \xrightarrow{t_G} S \xleftarrow{\text{dia}^{\ominus}; \text{slice}} c$, where $c \in C^{\ominus}$ and slice denote morphism $\text{slice} : (\text{Graph}/S)_0 \hookrightarrow S$.

ATOMIC
CONSTRAINTS IN
DPF

Example 3.3.3 (A meta-model formalized by a DPF specification). Figure 3.8 shows the simplified Petri net meta-model from the previous example using DPF's formalization. Atomic constraints have been used to annotate constraints.

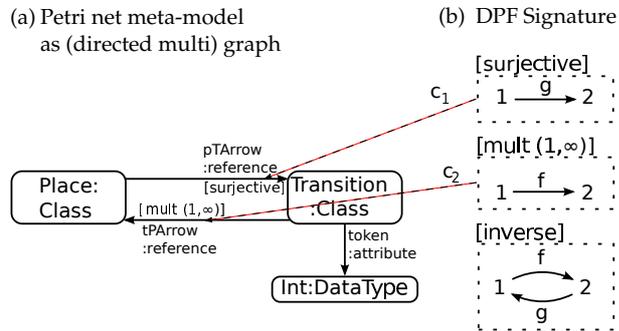


Figure 3.8: A simplified meta-model for Petri nets annotated with DPF atomic constraints

Edge `tPArrow` has been annotated with a `multiplicity` constraint specifying again that each transition should have at least one outgoing arrow. Edge `pTArrow` has in addition been annotated with a `surjective` constraint to define that every transition should be at least one target of one arrow. C_1 and C_2 denote the corresponding constraint identifier. The arrows going from the signature to the model graph present graph morphisms. Additional constraint annotations can be used in DPF such as `inverse` to not only annotate single elements but sub-graphs of a model as well. What type of constraint annotations exists is up to a DPF user.

The theory developed in this thesis is aimed to be applicable to a wide range of graph formalizations. Therefore, the theory widely abstracts away from a concrete formalization and employs adhesive (HLR) categories [36, 77] instead. All graphs presented in this chapter fit into this theory.

In this chapter, it has been shown how various types of graphs can be used to formalize models and modeling languages. In particular the concepts attributes, inheritance and (modeling) language constraints have been considered.

Adhesive Categories and Graph Transformations

Adhesive categories [77, 78] and (weak) adhesive High Level Replacement (HLR) categories [36] build a suitable categorical framework for formalizing graphs and algebraic graph transformations [36] in the more general case. Therefore, we consider them as adequate to formalize models. In particular, all model formalizations from the previous Chapter 3 fit into this framework. In this chapter, the basic theory of (weak) adhesive (HLR) categories and graph transformations based on cospans are summarized. In particular, we recall the cospan Double Pushout (cospan DPO) approach to graph transformation [37] and adapt the (span) sesqui Pushout (SqPO) approach [24] to cospan rules. Furthermore, we recall properties that are (partly) only valid in (weak) adhesive HLR categories in preparation for coupled transformations that are introduced in Chapter 6. Also, “new” propositions are presented: (1) the special pushout-pullback decomposition property [78] has been generalized, as well as (2) a theorem about the stability of Final Pullback Complements (FBPCs) [88]. To obtain a basic understanding of category theory and a deeper understanding of adhesive (HLR) categories, we refer to related works such as [11], [36, 77].

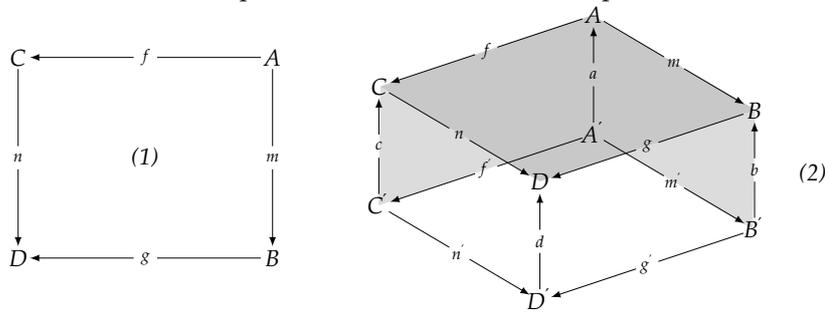
4.1 Adhesive Categories

The intuitive idea of adhesive categories is that of categories with compatible pushouts and pullbacks. More precisely, in adhesive categories, pushouts are stable under pullbacks and vice versa on the basis of van Kampen (VK) squares. (Weak) adhesive HLR categories are a generalization

of adhesive categories. While adhesive categories are based on all monomorphisms in a category \mathbf{C} , (weak) adhesive HLR categories are based on a suitable subclass \mathbf{M} of the monomorphisms in \mathbf{C} . Weak adhesive HLR categories in contrast to adhesive HLR categories require stronger preconditions for VK squares. There is an inclusion relation between all three types of adhesive categories: all adhesive categories are also adhesive HLR categories and all adhesive HLR categories are also weak adhesive HLR categories, but not vice versa. First, we recall the basic definitions [36, 77]:

VAN KAMPEN
SQUARE

Definition 4.1.1 (van Kampen square). A pushout (1) is a van Kampen square if, for any commutative cube (2) with (1) in the top face and where the back faces (light grey) are pullbacks, the following statement holds: the bottom face is a pushout iff¹ the front faces are pullbacks.



A sufficient² condition for the existence of VK squares relies on monomorphisms. Therefore, we recall the definition of a monomorphism next. Monomorphism generalizes injective morphism. In particular, in category \mathbf{Graph} , the injective morphisms are the monomorphisms.

MONOMORPHISM

Definition 4.1.2 (Monomorphism). Given a category \mathbf{C} , a morphism $m: B \rightarrow C$ is called a monomorphism (denoted by \rightarrowtail) if, for all morphisms $f, g: A \rightarrow B \in \mathbf{C}$, it holds that $f; m = g; m$ implies $f = g$:

$$A \begin{array}{c} \xrightarrow{f} \\ \xrightarrow{g} \end{array} B \rightarrowtail m \rightarrow C$$

Adhesive categories guarantee the VK property along monomorphism beside the existence of pushouts and pullbacks:

ADHESIVE
CATEGORY

Definition 4.1.3 (Adhesive category). A category \mathbf{C} is adhesive if:

1. \mathbf{C} has pushouts along monomorphism (i.e. pushouts where at least one of the given morphisms is a monomorphism).

¹As usual “iff” denotes “if and only if”.

²Note that there are also weaker sufficient conditions for VK squares [72]. However, in this thesis, only the condition relying on monomorphism is used, which is supported by the definition of (weak) adhesive (HLR) categories.

2. \mathbf{C} has pullbacks.
3. Pushouts along monomorphisms are VK squares.

(Weak) adhesive HLR categories guarantee VK squares only along a special class of monomorphism (and under stricter conditions):

Definition 4.1.4 ((Weak) adhesive HLR category). A category \mathbf{C} with a morphism class \mathbf{M} is called a (weak) adhesive HLR category if:

(WEAK) ADHESIVE
HLR CATEGORY

1. \mathbf{M} is a class of monomorphisms closed under isomorphisms, composition ($f : A \rightarrow B \in \mathbf{M}, g : B \rightarrow C \in \mathbf{M} \Rightarrow f;g \in \mathbf{M}$), and decomposition ($f;g \in \mathbf{M}, g \in \mathbf{M} \Rightarrow f \in \mathbf{M}$).
2. \mathbf{C} has pushouts and pullbacks along \mathbf{M} -morphisms (i.e. at least all pushout and pullbacks exist that include one \mathbf{M} -morphism), and \mathbf{M} -morphisms are closed under pushouts and pullbacks.
3.
 - a) *Adhesive HLR category*: pushouts in \mathbf{C} along \mathbf{M} -morphisms are VK squares.
 - b) *Weak adhesive HLR category*: pushouts in \mathbf{C} along \mathbf{M} -morphisms are weak VK squares. This means the VK square property holds for all commutative cubes with $m \in \mathbf{M}$ and $f \in \mathbf{M}$ or $b, c, d \in \mathbf{M}$ (see figure in Definition 4.1.1).

Many types of graphs fit into such categories that can be used to formalize problems in software engineering:

Example 4.1.1 (Examples of different types of adhesive categories). Table 4.1 lists examples for different types of adhesive categories together with the corresponding class of monomorphism.

Table 4.1: Adhesive, adhesive HLR and weak adhesive HLR categories

	Name	Morphism class \mathbf{M}
Adhesive categories	Set [77]	injective morphism
	Graph [77]	injective morphism
	Graph _{TRG} [36]	injective morphism
	EGraph (follows from [112])	injective morphism
	GSpec (Chapter B)	injective morphism
Adhesive HLR categories	SymbGraph _D [112]	injective morphism with equivalent formula in domain and codomain
	G [†] [89]	extremal monomorphisms
	HyperGraph [36]	injective hypergraph morphism
Weak adhesive HLR categories	PTNet [36]	injective morphism
	IGraph [54]	subtype reflecting I-graph morphism
	AIGraph [54]	subtype reflecting AI-graph morphism

In adhesive categories, morphism class \mathbf{M} contains all monomorphisms. Most of the categories in Table 4.1 have already been presented in Chapter 3. However, more categories are (weak) adhesive:

- **Set** is the category of sets. Its a well-known fact that many categories are based on **Set** e.g. **Graph**, Graph_{TG} , **EGraph** and **GSpec**.
- \mathbf{G}^{\top} is a category of graphs supporting inheritance similar to category **IGraph**. Vertices of type graphs are partially ordered and form a lattice (i.e. each type graph has a least an element “Everything” and a greatest element “Anything” by definition).
- Category **HyperGraph** denotes the category of hypergraphs. A hypergraph is a directed graph where each edge has instead of only one source and target vertex an arbitrary sequence of vertices as attachment points.
- Category **PTNet** denotes the category of place/transition (Petri) nets.
- Category **AIGraph** builds on category **IGraph**, and support edge attribution and data manipulation.

4.2 Properties of (Adhesive) Categories

Proofs that are presented in this thesis rely on several properties that are (partly) only valid in (weak) adhesive (HLR) categories. In Table 4.2, all of these properties are listed. We will refer to these properties by their numbers in subsequent chapters.

Table 4.2: Categorical properties and sufficient preconditions

	Property	Sufficient precondition
(1)	PB composition [11]	category with PBs
(2)	PB decomposition [11]	category with PBs
(3)	PO composition [11]	category with POs
(4)	PO along M -morphism is PB [36]	weak adhesive HLR category
(5)	M -morphism stable under PB [36]	weak adhesive HLR category
(6)	M -morphism stable under PO [36]	weak adhesive HLR category
(7)	Uniqueness of POCs [24, 36]	weak adhesive HLR category
(8)	VK Property [36]	adhesive HLR category
(9)	Special PB-PO property* [78]	adhesive HLR category with arbitrary PBs
(10)	Weak VK Property [36]	weak adhesive HLR category
(11)	Weak special PB-PO property* [78]	weak adhesive HLR category
(12)	FPBCs stable under PBs* [88]	category with PBs

*: The proof can be found in the appendix.

Pullback (PB), Pushout (PO), Pushout Complement (POC), Final Pullback Complement (FPBC)
 adhesive category \subseteq adhesive HLR category \subseteq weak adhesive HLR category
 \subseteq category with pullbacks

4.2. Properties of (Adhesive) Categories

Note that Properties 9, 11 and 12 are propositions from related work [78, 88] that have been generalized. Table 4.3 summarizes the facts of the properties listed in Table 4.2.

Table 4.3: Categorical properties

In the following, we always refer to the commuting diagram on the right (if not stated differently).	$ \begin{array}{ccccc} A & \xrightarrow{f} & C & \xrightarrow{p} & E \\ m \downarrow & & \downarrow n & & \downarrow l \\ B & \xrightarrow{g} & D & \xrightarrow{q} & F \end{array} $ <p style="text-align: center;">(1) (2)</p>
(1) PB composition	If squares (1) and (2) in the diagram are pullbacks then also square (1 + 2) is a pullback.
(2) PB decomposition	If squares (1 + 2) and (2) in the diagram are pullbacks then also square (1) is a pullback.
(3) PO composition	If squares (1) and (2) in the diagram are pushouts then also square (1 + 2) is a pushout.
(4) PO along M -morphism is PB	If square (1) is a pushout and m a M -morphism then square (1) is also a pullback.
(5) M -morphism stable under PB	If square (1) is a pullback and n a M -morphism then also m is a M -morphism.
(6) M -morphism stable under PO	If square (1) is a pushout and m a M -morphism then also n is a M -morphism.
(7) Uniqueness of POC	If square (1) is a pushout and m a M -morphism then the pushout complement $A \xrightarrow{f} C \xrightarrow{n} D$ of $A \xrightarrow{m} B \xrightarrow{g} D$ is unique (up to iso.).
(8) VK Property	see Definition 4.1.4, 3 a) and Definition 4.1.1
(9) Special PB-PO property	If square (1 + 2) is a pullback, square (1) a pushout and m, n and l are M -morphisms then square (2) is a pullback.
(10) Weak VK Property	see Definition 4.1.4, 3 b) and Definition 4.1.1
(11) Weak special PB-PO property	If square (1 + 2) is a pullback, square (1) a pushout and m, n, l, f and g are M -morphisms then square (2) is a pullback.
(12) FPBCs stable under PBs	Consider the cube of Definition 4.1.1 with all faces as pullbacks and $A \xrightarrow{f} C \xrightarrow{n} D$ being a Final Pullback Complement then also $A' \xrightarrow{f'} C' \xrightarrow{n'} D'$ is a Final Pullback Complement.

In the next section, we introduce graph transformations based on cospans that already rely on a subset of these properties.

4.3 Graph Transformations based on Cospans

One main contribution of this thesis is a theory of coupled transformations. Coupled transformations are build on top of new variants of algebraic graph transformations based on cospan rules. Generally, algebraic graph transformations manipulate graphs by use of categorical constructions. In the literature, there are three well-known variations of algebraic graph transformation approaches:

DIFFERENT GRAPH
TRANSFORMATION
APPROACHES

1. the Double-Pushout [36] (DPO),
2. the Single-Pushout [87] (SPO),
3. and the Sesqui Pushout [24] (SqPO) approach.

While all approaches use an elegant and compact description of graph rewriting, they differ with respect to the types of morphisms under consideration, the form of the rules, and the commuting diagrams on which the rewriting steps are based. A rule in the SPO approach is a morphism $p = L \rightarrow R$ in a category of graphs and partial graph morphisms, while a rule in the DPO approach and the SqPO approach is a span $p = L \xleftarrow{l} I \xrightarrow{r} R$ in a category of graphs and total graph morphisms [24].

The formalization of coupled transformations in this thesis is based on (adhesive) categories with total morphisms. Therefore, we only consider variants of the double and sesqui pushout approach here. While the DPO approach and the SqPO approach traditionally use span rules, we employ cospan rules $p = L \xrightarrow{l} I \xleftarrow{r} R$ instead. This switch to cospan rules is justified in Chapter 6, where it is shown that cospan transformations have several properties that are beneficial for synchronizing two related transformations.

The graph transformations we work with operate in a category of graphs, e.g. in the category of (directed multi-)graphs, \mathbf{Graph} . In \mathbf{Graph} , it is well known that the monomorphisms are the injective morphisms and the epimorphisms are the surjective morphisms [36]. A generalization of the given definitions can be obtained by substituting graphs by objects of a (weak) adhesive HLR category, injective morphisms by \mathbf{M} -morphisms and surjective morphisms by epimorphisms. One rule in the cospan DPO approach as well as in the cospan SqPO approach is a cospan between three graphs. The left-hand side L represents the pre-conditions of a rule, while the right-hand side R describes its post-conditions. The intermediate graph I clarifies how L and R overlap.

4.4. Cospan Double Pushout Approach

Definition 4.3.1 (Cospan transformation rule). A *cospan transformation rule* $p = L \xrightarrow{l} I \xleftarrow{r} R$ consists of graphs L (left-hand-side), I (intermediate), and R (right-hand-side) and two jointly surjective³ graph morphisms l and r .

COSPAN
TRANSFORMATION
RULE

If rule morphism $l = L \rightarrow I$ is injective, a rule p is called *left-linear*, if rule morphism $r = R \rightarrow I$ is injective, it is called *right-linear* and if both are injective, the rule is only called *linear*.

Should r be bijective, rule p is in addition called *non-deleting* and can be written as $p = l = L \rightarrow I$. For non-deleting rules, I is also called right-hand side. Analogously, should l be bijective, rule p is called *non-creating* and can be written as $p = r = I \leftarrow R$. For non-deleting rules, I is also called left-hand side.

4.4 Cospan Double Pushout Approach

A well-known approach is the (span) DPO approach [36], where two pushouts are used to describe graph changes by first deleting graph elements and then creating new elements or merging preserved items. In the following, we consider a new variant of the DPO approach called cospan Double-Pushout approach [37] (cospan DPO). In this approach, basic graph changes are performed in the opposite way, meaning that graph elements are created or merged first before elements are deleted. This implies that new graph elements are inserted in a richer context where graph elements to be deleted are still available. In [37], it is shown that the adjoint rule, that is obtained by taking the pullback of a cospan rule, respectively the pushout of a span rule, produces the same transformation result in case of linear rules in any weak adhesive HLR category.

Given a rule $p = L \xrightarrow{l} I \xleftarrow{r} R$, a graph transformation $t : G \xrightarrow{p,m} H$ transforming a graph G into a graph H is applied by first finding a morphism $m : L \rightarrow G$, called *match*, from the left-hand side L of the rule p to graph G and second, by adapting G in two steps. In the first step, elements corresponding to $r(R) \setminus l(L)$ are created in G (resulting in U), while elements of G that correspond to elements in L that are non-injectively mapped via rule morphism $l : L \rightarrow I$ are merged accordingly. In the second step, elements corresponding to $l(L) \setminus r(R)$ are deleted in U (resulting in H).

Definition 4.4.1 (Cospan DPO transformation). Given a right-linear

COSPAN DPO
TRANSFORMATION

cospan transformation rule $p = L \xrightarrow{l} I \xleftarrow{r} R$, together with a graph morphism $m : L \rightarrow G$, called *match*, rule p can be applied to G if a cospan double-pushout exists as shown in the diagram on the right. Transformation $t : G \xrightarrow{p,m} H$ is called a *cospan double pushout transformation*.

$$\begin{array}{ccccc} L & \xrightarrow{l} & I & \xleftarrow{r} & R \\ m \downarrow & & \downarrow i & & \downarrow m' \\ G & \xrightarrow{s} & U & \xleftarrow{h} & H \end{array}$$

(PO1) (PO2)

³i.e. $l(L) \cup r(R) = I$

INJECTIVE AND
NON-INJECTIVE
MATCHING

Pushout (1) in the diagram of Definition 4.4.1 can directly be constructed, pushout (2) is constructed by a pushout complement. As pushout complements are only unique⁴ if $r = R \twoheadrightarrow I$ is injective (see Property 7 in Table 4.3), cospan rules in the cospan double pushout approach are always right-linear. This implies that (cospan) DPO transformations do not support the splitting of elements. A match $m : L \rightarrow G$ can be injective or non-injective in general. Even injective matching is more frequently used since matches can be found easier; graph transformation tools such as AGG [30] also offer to search for non-injective matches. However, we primarily work with injective matching here. Therefore, we allow non-left linear rules that support the merging of graph elements. This is beneficial e.g. to specify refactoring “Merge Class”. Note, in (weak) adhesive (HLR) categories (see Definition 4.1.3 and Definition 4.1.4), only pushouts along \mathbf{M} -morphism must exist, therefore it is usual to restrict either to linear rules or injective matching. Under this restriction, double pushout approaches could also be called double pullback approaches, as pushouts along \mathbf{M} -morphism are pullbacks (Property 4).

GLUING CONDITION

A graph transformation rule can be applied if both pushouts can be uniquely constructed i.e. if a unique pushout complement exists. This is the case if the gluing condition is satisfied i.e. a rule application does not produce dangling edges (dangling condition) and all elements are uniquely identified (identification condition). In case injective matching is used, the gluing condition can be reduced to the dangling condition. The gluing condition for cospan DPO transformations can be found in [37].

The DPO approaches⁵ allow the performance of *create*, *delete* and *merge* operations. In contrast to the DPO approaches, the SqPO approaches⁶ additionally allow for *splitting* elements as well as for deleting elements in *unknown context*. Therefore, the cospan SqPO approach is introduced next.

Example 4.4.1 shows a sample graph transformation in the cospan DPO approach in category Graph_{TG} . Example 6.1.1 in Chapter 6 shows another sample graph transformation in category GSpec .

⁴in weak adhesive HLR categories

⁵DPO approach or cospan DPO approach

⁶SqPO approach or cospan SqPO approach

Example 4.4.1 (Cospan DPO transformation in category Graph_{TG}). Figure 4.1 shows a cospan DPO transformation that moves an attribute along a reference. Because graph transformations cannot change the target of edges, the “move” operation has to be modeled by create and delete. The transformation can also be seen as a cospan SqPO transformation.

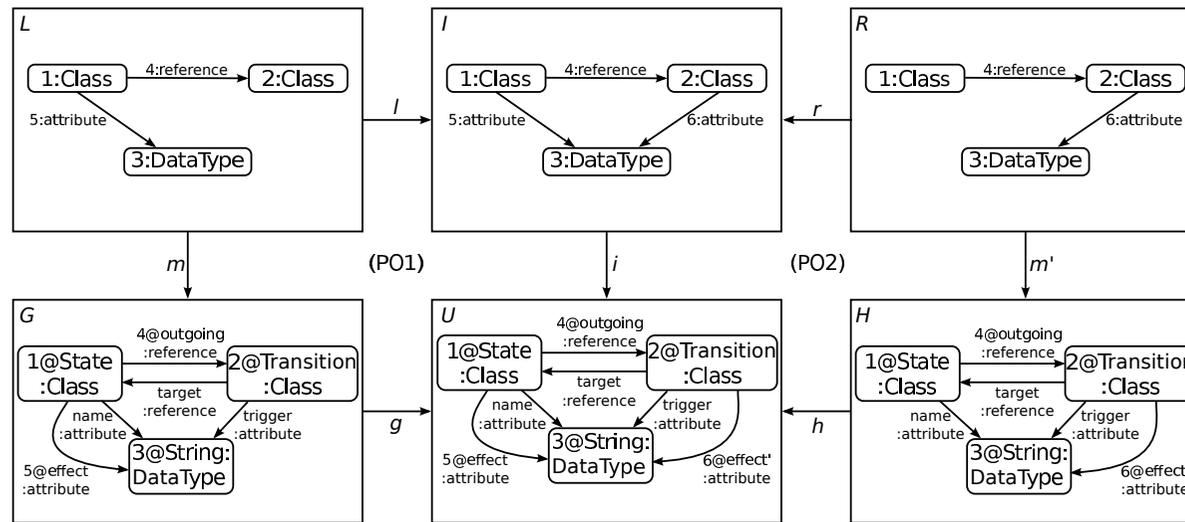


Figure 4.1: A cospan DPO transformation

The transformation evolves a type graph for Moore into one for Mealy automata. Mappings are indicated by numbers. New elements ids (names in the models) are defined by the transformation executor.

4.5 Cospan Sesqui Pushout Approach

The (span) SqPO approach has been introduced by Corradini and Heindel [24] as a deterministic and conservative extension of DPO rewriting. Rules are defined as in the traditional DPO approach as spans but are not required to be left-linear. This allows for the performance of split operations on elements. In this section, we adapt the definitions of [24] and also define sesqui pushout transformations for cospan rules. This means cospan rules are no longer required to be right-linear.

Instead of a pushout and a pushout complement, rules in the (cospan) SqPO approach are applied by a pushout and a Final Pullback Complement (FPBC). A FPBC can be characterized as the largest pullback complement, where largest means the pullback complement with the largest possible pullback object (see Definition A.2.1 in the appendix). Interestingly, in case of right-linear cospan rules, FPBCs are isomorph to pushout complements [24]. Hence, it is a real generalization. We define a cospan sesqui pushout transformation as follows:

COSPAN SQPO
TRANSFORMATION

Definition 4.5.1 (Cospan SqPO transformation). Given a cospan transformation rule $p = L \xrightarrow{l} I \xleftarrow{r} R$ together with a morphism $m : L \rightarrow G$, called *match*, rule p can be applied to G if the pushout in the left square in the diagram on the right exists and the pullback in the right square with $R \xrightarrow{m'} H \xrightarrow{h} U$ being the FPBC of $R \xrightarrow{r} I \xrightarrow{i} U$. Transformation $t : G \xrightarrow{p,m} H$ is called a *cospan sesqui pushout transformation*.

$$\begin{array}{ccccc}
 L & \xrightarrow{l} & I & \xleftarrow{r} & R \\
 m \downarrow & & (PO) \downarrow i & & (PB) \downarrow m' \\
 G & \xrightarrow{g} & U & \xleftarrow{h} & H
 \end{array}$$

CONFLICT FREENESS

A cospan SqPO transformation rule can be applied if the FPBC exists. Interestingly, FPBCs are always unique up to isomorphism, if they exist. In [24, 70] the existence of FPBCs is studied for \mathbf{Graph} as well as for arbitrary categories. For \mathbf{Graph} , a sufficient condition for the existence of FPBCs is characterized as *conflict freeness* in [24], a weaker condition as the gluing condition in DPO graph rewriting. In category \mathbf{Graph} , the condition is satisfied for every injective match [24], i.e. that only in a case of non-injective matching the conflict-freeness condition needs to be checked.

Example 4.5.1 shows a sample graph transformation in the cospan SqPO approach in category \mathbf{Graph}_{TG} .

Example 4.5.1 (Cospan SqPO transformation in category Graph_{TG}). Figure 4.2 shows a cospan SqPO transformation which creates a (super)class and split its (sub)class. Context elements are cloned due to the non-injective morphism $r = R \rightarrow I$.

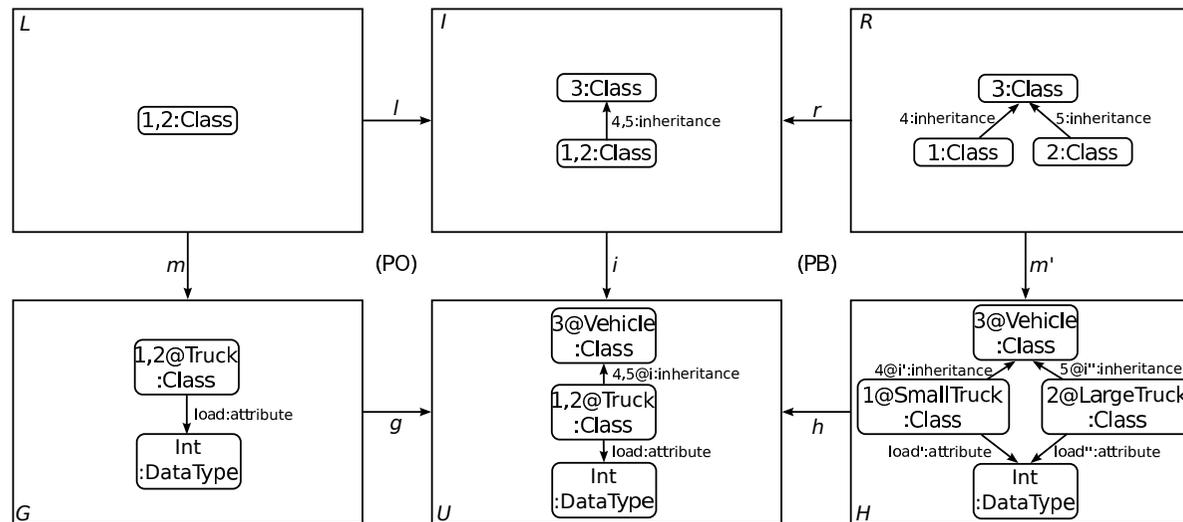


Figure 4.2: A cospan SqPO transformation

The transformation creates vertex `Vehicle` and splits vertex `Truck`. Attribute `load` and inheritance edge `i` are cloned. Mappings are again indicated by numbers.

4.6 Summary of Approach Differences

In the previous section, the cospan SqPO approach has been introduced as a generalization of the cospan DPO approach. In this section, we compare both approaches and the SPO approach. Most of the differences are independent from the formalization of the rules (i.e. if cospan rules or span rules are used for DPO or SqPO transformations).

DIFFERENCES OF
ALGEBRAIC GRAPH
TRANSFORMATION
APPROACHES

First, transformation rules are applicable under stricter conditions in the DPO approaches than in the SqPO approaches, as FPBCs may also exist if unique pushout complements do not exist. If a unique pushout complement exists, it is also a FPBC. If the pushout complement does not exist but the FPBC does, deletion in unknown context is modeled as in the SPO approach: dangling edges are removed. Therefore, the SqPO approaches do not require a dangling condition (which is part of the gluing condition) as the DPO approaches [24]. Instead of the gluing condition, the existence of FPBC has to be checked. In the SPO approach, only pushouts must exist. Hence, the SPO approach has the weakest precondition. This is generally ensured for pushouts along \mathbf{M} -morphisms (see Definition 4.1.4). Therefore, the SPO approach may delete identified elements preserved by the rule morphism, i.e. it prioritizes deletion over preservation. Furthermore, the approaches differ in the supported transformation operations. In contrast to the other approaches, the SqPO approaches support the splitting of elements (i.e. cloning).

Table 4.4 gives an overview about all three types of transformations approaches and their differences. While there are no differences between graph transformation approaches and their cospan equivalents in Table 4.4, we observed in [80] that cospan approaches have advantages should transformations be applied bidirectionally.

Table 4.4: Supported operations of graph transformation approaches

	(cospan) DPO	(cospan) SqPO	SPO
create	✓	✓	✓
delete	✓	✓	✓
merge	✓	✓	✓
split		✓	
delete in unknown context		✓	✓
precedence of deletion over preservation			✓

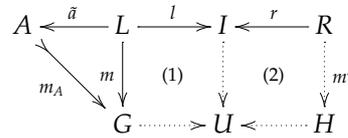
4.7 Application Condition

The left-hand side of a graph transformation rule can be considered as a precondition for their application i.e. if the left-hand side is matched a rule may be applied (i.e. also the gluing condition/conflict-freeness condition

is satisfied). However, further conditions to specify the applicability of a graph transformation rule are desired. In particular, it is common also to check for the non-existence of graph patterns before a graph transformation rule is applied. Such conditions can be specified by application conditions (see e.g. [36]). Additional application conditions as defined next can be used for cospan rules in the DPO or SqPO approach:

Definition 4.7.1 (Simple application condition). A simple application condition is in the form $AC(\tilde{a})$, where $\tilde{a} : L \rightarrow A$ is a morphism. Application conditions are either negative (NAC) or positive (PAC). A match $m : L \rightarrow G$ of a rule satisfies: (1) a PAC(\tilde{a}) if there *exists*, (2) a NAC(\tilde{a}) if there *does not exist*, an injective morphism $m_A : A \rightarrow G$ with $\tilde{a}; m_A = m$.

SIMPLE
APPLICATION
CONDITION



Simple application conditions can be composed to boolean formulas for more complex conditions:

Definition 4.7.2 (Application condition). An application condition is a simple application condition or a Boolean expression of application conditions. A morphism $m : L \rightarrow G$ satisfies an application condition $A1 \wedge A2$ ($A1 \vee A2$) if it satisfies $A1$ and (or) $A2$. Morphism m satisfies $\neg A$ if it does not satisfy A .

APPLICATION
CONDITION

4.8 Transformation Variants

Algebraic graph transformation is an abstract but also flexible transformation approach. Graph transformations can be used for *homogeneous* and *heterogeneous* transformations, as well as for *in-place* and *out-place* transformations (see Chapter 2).

Homogeneous transformations are transformations where the source and target modeling language (i.e. their type graphs) are the same. This is obviously supported by graph transformations in any category of typed graphs.

Heterogeneous transformations are transformations where the source and target modeling language are not the same. Such transformations can be specified by creating a joint modeling language first, containing both modeling languages as well as new elements relating types. Heterogeneous transformations become due to this construction homogeneous

transformations in a joint modeling language. After the complete transformation is finished, the target model can be extracted from the original transformation result by keeping only all elements that exist in the target modeling language. This can be formalized by a pullback construction.

In-place transformations are transformations where a model (i.e. graph) is changed. Usually, graph transformations are used for in-place transformations. However, a key concept of category theory is that object relationships are considered rather than their internal structures. In particular, this means algebraic graph transformations distinguish graph elements up to isomorphisms only, i.e. the theory of graph transformations abstracts from the question whether elements are reused or copied. Therefore, it is up to a tool designer if a graph transformation finally transforms in-place or out-place. If tools reuse the elements from G in the pushout, respectively pullback constructions transformations can be considered as in-place.

Out-place transformations are vice versa transformations that create new models by copying elements. Therefore, a tool only needs to use pushout and pushout/Final Pullback Complement constructions that create new elements instead of reusing existing ones. In addition, mixed constructions are possible. This can be helpful if element identifiers are also used as element names. A renaming is always possible even if transformations should transform primarily in-place.

In this chapter, (weak) adhesive (HLR) categories have been recalled. In particular, properties that are ensured in these categories have been listed. Two of them have been generalized. Those properties are used in proofs in subsequent chapters. Furthermore, the cospan DPO graph transformation approach has been recalled, while the SqPO graph transformation approach has been transferred to co-span rules. In addition, application conditions have been summarized.

Detecting Evolution Steps by Graph Transformation Rules

In Chapter 2, three different types of model migration approaches from the literature [118] have been discussed: manual specification, operator-based and matching approaches. In contrast to the first type of approach (manual specification), in which model migration needs to be specified manually, meta-model evolution steps are explicitly considered in the second (operator-based) and third (matching) types of approaches. In this chapter, we focus on the third type of approach. We introduce a new formal approach for detecting sequences of meta-model evolution steps (see Figure 5.1) by graph transformation rules based on cospans. Thereafter, we discuss its advantages over an alternative approach based on span rules. This chapter is based on [91].

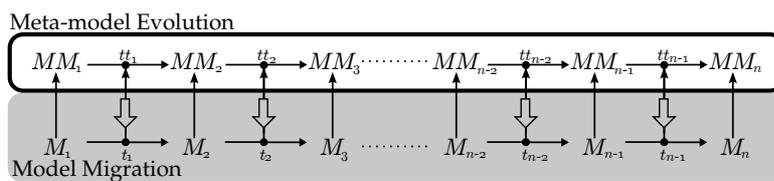


Figure 5.1: Coupled Evolution Steps

How models can be migrated along detected sequences of meta-model evolution steps is explained in subsequent chapters.

5.1 Introduction

Example 5.1.1 shows an introductory sample evolution that has been used previously in various articles [21, 118, 146]. A Petri net meta-model is evolved into one supporting weighted edges.

Example 5.1.1 (Petri net meta-model evolution). Figure 5.2 (a) shows a Petri net meta-model that does not support weighted edges while Figure 5.2 (b) shows a Petri net meta-model that does.

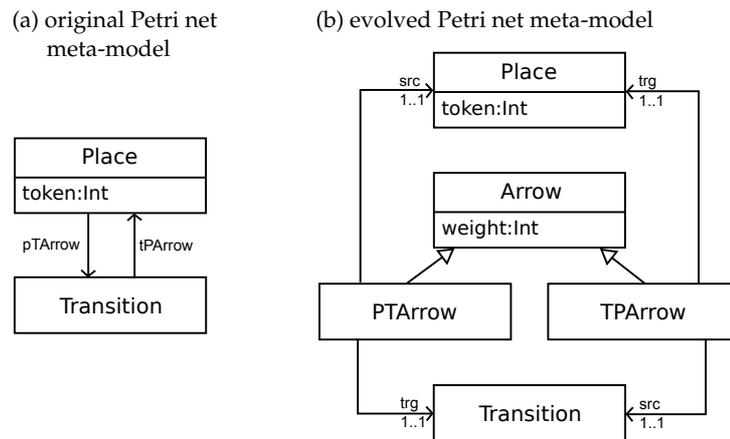


Figure 5.2: Petri net meta-model evolution

The Petri net meta-model in Figure 5.2 (a) might have been evolved e.g. in six steps:

1. Replace reference `pTArrow` by class `PTArrow` and two references `src` and `trg` with corresponding multiplicity constraints.
2. Add attribute `weight` to class `PTArrow`.
3. Replace reference `tPArrow` by class `TPArrow` and two references `src` and `trg` with corresponding multiplicity constraints.
4. Add attribute `weight` to class `TPArrow`.
5. Add superclass `Arrow`.
6. Pull up attribute `weight` to superclass `Arrow`.

However, other evolution sequences are possible. Evolution steps might have been performed by rules or by editing the model directly.

If a meta-model has been evolved by several cospan graph transformation rules, we can also relate the first and the final meta-model version by a cospan using pushouts and composition of morphisms (see Example 5.1.2). In the following, we call such cospans *meta-model cospans* (or *type graph cospans*).

Example 5.1.2 (Relating meta-models by a cospan). Figure 5.3 shows a sequence of meta-model evolution steps formally described by three graph transformations evolving a meta-model MM_1 to MM_7 . Meta-models MM_1 and MM_7 can be related by a meta-model cospan $MM_1 \xrightarrow{t_1; a_1, a_2} MM_{2,6} \xleftarrow{t_2; b_1, b_2} MM_7$.

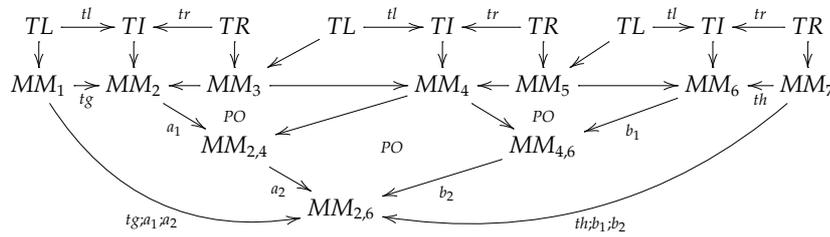


Figure 5.3: Type graph cospan composition

The idea is to decompose a given meta-model cospan again using graph transformation rules into a sequence of rule applications. A meta-model cospan may be directly derived by the element identifier in a suitable modeling framework (see Example 5.1.3).

Example 5.1.3 (Deriving a meta-model cospan). The example builds on Example 5.1.1 and shows a listing for each meta-model in Figure 5.2 (a) and (b). The textual presentation for the models is based on categories IGraph , DPF and SymbGraph_D . A tool, *DPF Text*, supporting such models has been developed as part of the PhD project. Listing 5.1 shows the meta-model of Figure 5.2 (a).

Listing 5.1: Figure 5.2 (a) in DPF Text

```

1 Specification:ClassModel<6> {
2   Graph {
3     Place@2:Class@0{
4       pTArrow@4:reference@1->Transition@1:Class@0,
5       token@5:*->Int
6     },
7     Transition@1:Class@0{
8       tPArrow@3:reference@1->Place@2:Class@0
9     }
10  }
11 }

```

Each element in the listings has a name and in addition an identifier automatically assigned by the tool. In addition, each element has a type

consisting of type name and type identifier. The tool automatically formats the textual presentation similar to class specifications in textual programming languages. Listing 5.2 shows the meta-model of Figure 5.2 (b).

Listing 5.2: Figure 5.2 (b) in DPF Text

```

1 Specification:(ClassModel ,MySig)<14> {
2   Graph {
3     Arrow@12:Class@0{
4       weight@13,14:*->Int
5     },
6     PTAArrow@6:Class@0 extends Arrow@12:Class@0{
7       src@8:reference@1->Place@2:Class@0,
8       trg@7:reference@1->Transition@1:Class@0
9     },
10    Place@2:Class@0{
11      token@5:*->Int
12    },
13    TPAArrow@9:Class@0 extends Arrow@12:Class@0{
14      src@10:reference@1->Transition@1:Class@0,
15      trg@11:reference@1->Place@2:Class@0
16    }
17  }
18  Constraints {
19    minMax@4("1"){PTAArrow@6:Class@0-src@8:reference@1->Place@2:Class@0},
20    minMax@4("1"){PTAArrow@6:Class@0-trg@7:reference@1->Transition@1:Class@0},
21    minMax@4("1"){TPAArrow@9:Class@0-src@10:reference@1->Transition@1:Class@0},
22    minMax@4("1"){TPAArrow@9:Class@0-trg@11:reference@1->Place@2:Class@0}
23  }
24 }

```

DPF Text automatically reduces unnecessary duplications. For example, class `Transition` is only presented as a target of a reference since the class does not have any outgoing edges (i.e. attributes or references). Furthermore, identifier can be sets of integer values. The identifier of the weight attribute consists, for example, of two integers. Such identifiers can indicate e.g. a merge of two elements¹. Hence, also, merges and splits (identifier consists of a new integer and the previous integer) may be detected by help from such model presentations. Listing 5.3 shows the DPF signature used in Listing 5.2 containing additional atomic constraints.

Listing 5.3: DPF signature used in Listing 5.2

```

1 Signature<6, OCL> {
2   abstract@1(min){x:_}="context #x# inv: false",
3   irr@2(){x:_-y:_->x:_}="context #x# inv: not #y#->includes(self)",
4   min@3(min){x:_-y:_->z:_}="context #x# inv: #y#->size() >= #min#",
5   minMax@4(minMax){x:_-y:_->z:_}="context #x# inv: #y#->size() = #minMax#",
6   sur@5(){x:_-y:_->z:_}="context #z# inv: not 0#y#->isEmpty()"
7 }

```

The semantics of the signature has been specified by OCL templates. In such a signature, arbitrary new constraints can be specified. Note also that the weight attribute could have been supported by such a signature predicate alternatively to the shown meta-model evolution. Predicates can

¹In `IGraph`, it is possible to specify the “Pull Up Reference” refactoring by a merge.

be specified for arbitrary shape graphs. Types in the signature are omitted (shown as “_”), as the signature is untyped.

Obviously, the meta-model cospan shown in Figure 5.4 can be derived by mapping the identifier. Due to space restrictions in subsequent examples, a more compact graph presentation of the meta-models is used. Only the first letter(s) of names are shown and the typing has been omitted. However, element identifiers are shown either as subscripts in vertices or postfixes in edges.

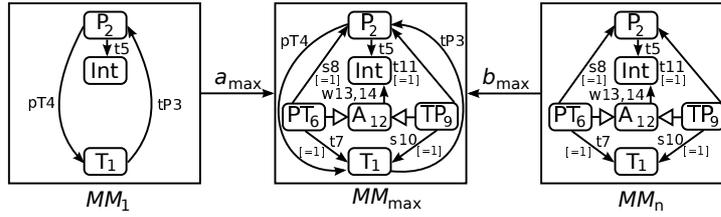


Figure 5.4: Petri net meta-model cospan

5.2 Detecting Evolution Steps with Cospan Rules

In this section, Procedure 5.2.1 is developed, which can be used to decompose meta-model cospans by cospan rules. The Procedure 5.2.1 is built on the idea of triple matches. This means instead of matching only the left-hand side of the rule, we match the whole cospan rule in the meta-model cospan.

Definition 5.2.1 (Triple match of a cospan transformation rule).

Assume a cospan $MM_1 \xrightarrow{a} MM_{max} \xleftarrow{b} MM_n$

relating three graphs (respectively objects in a (weak) adhesive (HLR) category) MM_1 , MM_{max} and MM_n : a triple match (m_{TL}, m_{TI}, m_{TR}) of a

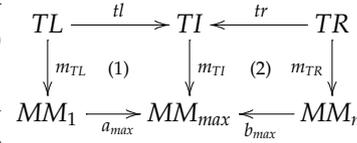
cospan rule $tp = TL \xrightarrow{tl} TI \xleftarrow{tr} TR$ is defined by three matches $m_{TL} : TL \rightarrow MM_1$,

$m_{TI} : TI \rightarrow MM_{max}$ and $m_{TR} : TR \rightarrow MM_n$ so that squares (1) and (2) in the diagram on the right commute.

Having a meta-model cospan, the cospan may be incrementally decomposed:

Procedure 5.2.1 (Type graph cospan decomposition by cospan rules). Given type graph cospan $TCS = M_1 \xrightarrow{a_{max}} M_{max} \xleftarrow{b_{max}} M_n$ (meta-model cospan) and a set of cospan rules R in a (weak) adhesive (HLR) category, then cospan TCS may be decomposed into a sequence of cospan rule applications (evolution steps) by applying the following procedure (see figure below):

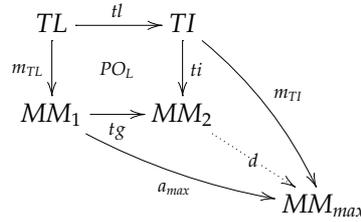
TRIPLE MATCH OF A
COSPAN
TRANSFORMATION
RULE



5. DETECTING EVOLUTION STEPS BY GRAPH TRANSFORMATION RULES

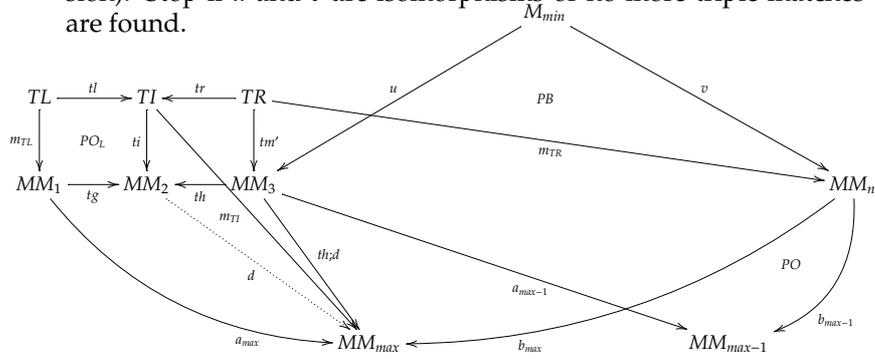
TYPE GRAPH
COSPAN
DECOMPOSITION BY
COSPAN RULES

1. Find a triple match (m_{TL}, m_{TI}, m_{TR}) of a cospan rule $tp \in R$ in cospan TCS . Note: the rules in R should have a priority. Rules with a higher priority are matched first.
2. Apply² cospan rule tp to type graph MM_1 at match $m_{TL} : TL \rightarrow MM_1$. Note, if the rule cannot be applied continue with Step 1. If the cospan SqPO approach is used, it must be ensured in addition that the rule does not delete unmatched context elements.
3. Obtain $d : MM_2 \rightarrow MM_{max}$ as mediating morphism from the left pushout PO_L of tp 's rule application.



The inner square commutes since it is a pushout. The outer square commutes since it is a triple match.

4. Construct $MM_3 \xleftarrow{u} MM_{min} \xrightarrow{v} MM_n$ as a pullback of $MM_3 \xrightarrow{th;d} MM_{max} \xleftarrow{b_{max}} MM_n$.
5. Construct a new type graph cospan $MM_3 \xrightarrow{a_{max-1}} MM_{max-1} \xleftarrow{b_{max-1}} MM_n$ as a pushout of $MM_3 \xleftarrow{u} MM_{min} \xrightarrow{v} MM_n$.
6. Continue to decompose cospan $MM_3 \xrightarrow{a_{max-1}} MM_{max-1} \xleftarrow{b_{max-1}} MM_n$ (recursion). Stop if u and v are isomorphisms or no more triple matches are found.



Example 5.2.1 applies Procedure 5.2.1 to the sample meta-model cospan from Example 5.1.3.

Example 5.2.1 (Petri net meta-model cospan decomposition). Figure 5.5 shows the first detected evolution step in detail (relying on the cospan DPO approach), while Figure 5.6 shows the next three detected evolution steps.

²cospan DPO or cospan SqPO approach

5.2. Detecting Evolution Steps with Cospan Rules

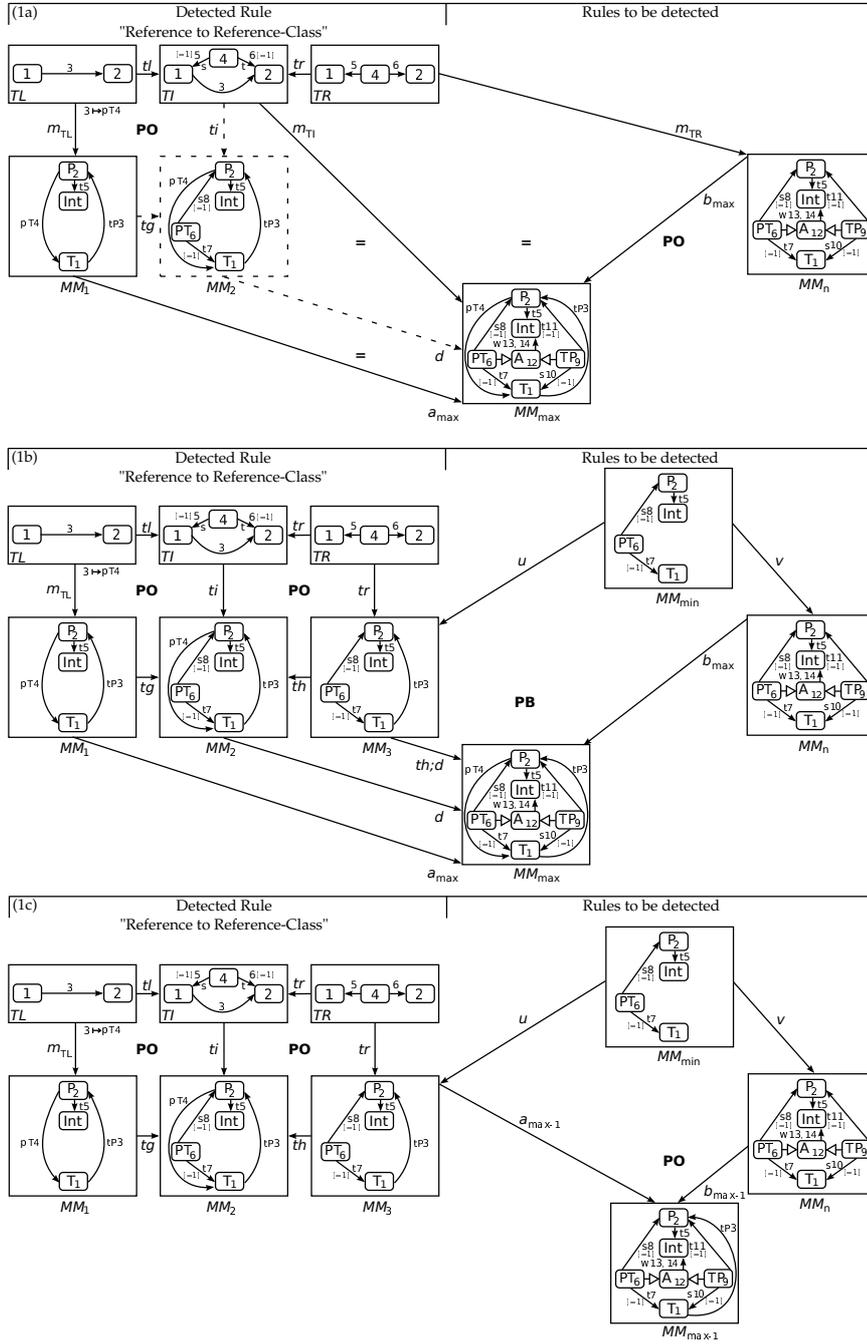


Figure 5.5: Detecting Evolution Step 1

5. DETECTING EVOLUTION STEPS BY GRAPH TRANSFORMATION RULES

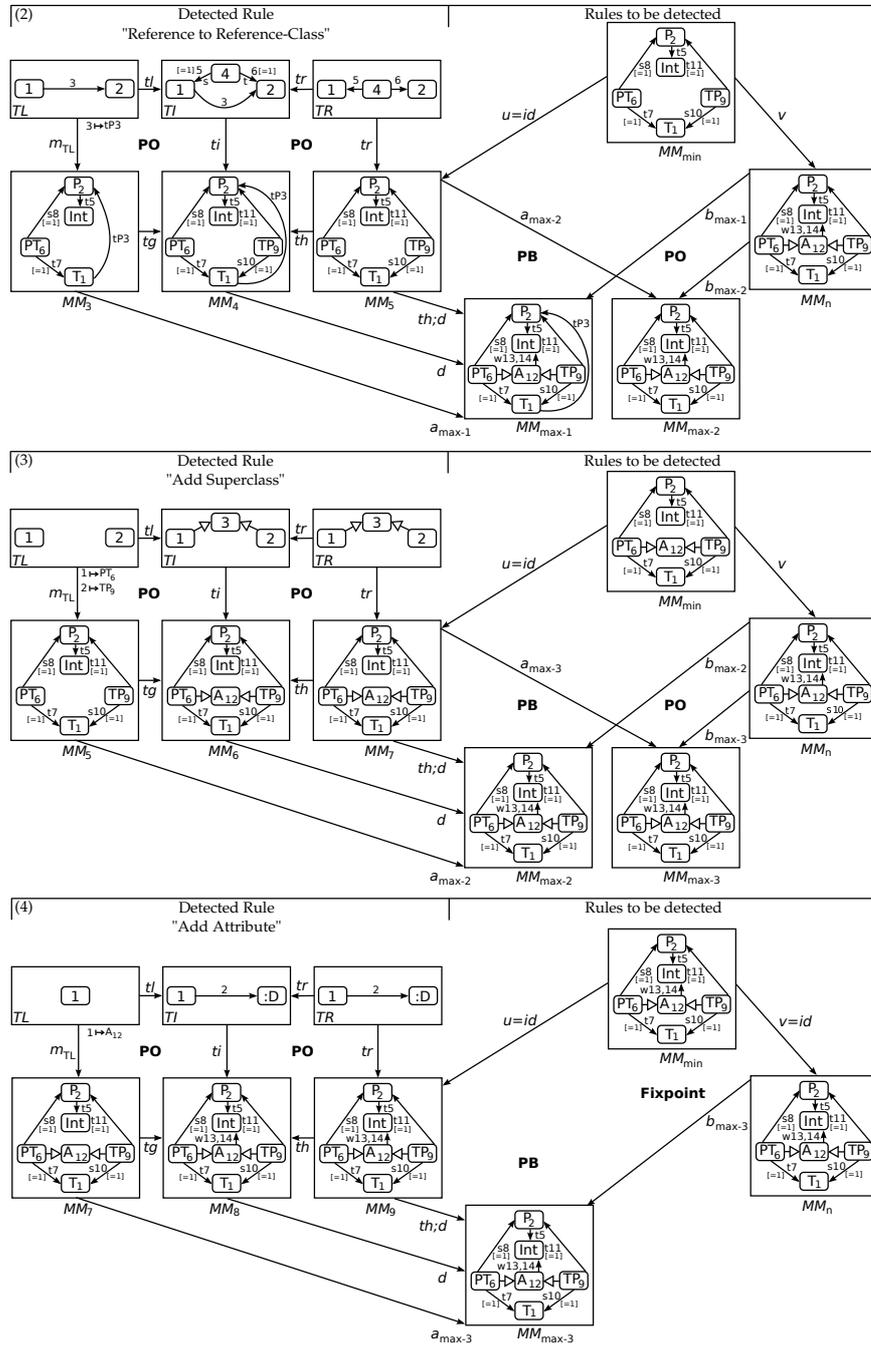


Figure 5.6: Detecting Evolution Steps 2-4

Figure 5.5 (1a) shows a detected triple match of Rule “Reference to Reference-Class”. Note the rule also makes use of the reference names s (src) and t (trg) to correctly detect the corresponding reference class for the edge. Rule matches should be obvious in the example, however, some mappings (e.g. $3 \mapsto pT4$) are denoted to make it easier for the reader. Figure 5.5 (1a) shows in addition the left pushout of the rule application and the deduced morphism $d : MM_2 \rightarrow MM_{max}$. The figure shows Step 1, (partly) Step 2 and Step 3 of Procedure 5.2.1. Figure 5.5 (1b) shows Step 4 of Procedure 5.2.1 and Figure 5.5 (1c) shows Step 5. Step 6 is the recursion step.

Figure 5.5 (2) shows another detected evolution step by Rule “Reference to Reference-Class”. Starting from this sub-figure, detected triple matches are neglected while Steps 2 to 5 of Procedure 5.2.1 are shown. After this step, no elements need to be deleted or merged anymore and morphism $u = id$. Figure 5.5 (3) shows a detected evolution step “Add Superclass”. Such a rule may be described by an amalgamated graph transformation rule [17, 133] (see Chapter 7) i.e. by a generic rule for an arbitrary number of subclasses. Figure 5.5 (4) shows a detected evolution step “Add Attribute”³. Such simple rules should have a low priority during matching to detect more complex evolution steps instead of simple ones. After these steps, both morphisms u, v are $u = v = id$ and the procedure stops.

Hence, the detected sequence of evolution steps is: (1. and 2.) “Reference to Reference-Class,” (3.) “Add Superclass,” (4.) “Add Attribute”. In addition, all required rule “parameter” e.g. names for new classes are detected so that the meta-model evolution sequence can be replayed.

Note that Procedure 5.2.1 can also be used if elements became merged or split, even if not shown in the example above. However, the procedure may be improved for a practical application. For example, evolution steps should be detected that create elements that are deleted by a next step (e.g. a property is moved along two references). This challenge may be solved by automatically extending the rule set by generated rules combining evolution rules. Furthermore, rule sets may not be confluent and backtracking may in addition be useful to stop in Step 6 of Procedure 5.2.1 with u and v isomorphisms instead of “no more matches found”. Both challenges, we consider as future work.

Remark 5.2.1 (Rule priorities). Due to the nature of the problem, the procedure may not always provide the desired result. However, rule priorities may help. In Example 5.2.1, e.g. rule priorities would help to not detect the following sequence:

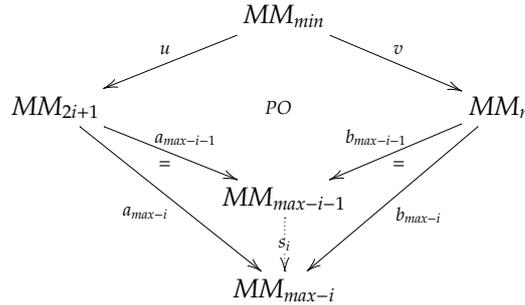
1. Replace reference `pTArrow` by class `PTArrow` and two references `src` and `trg` with corresponding multiplicity constraints.

³Vertex `:D` denotes a data type.

2. Add superclass `Arrow` to class `PTArrow`.
3. Add attribute `weight` to class `Arrow`.
4. Replace reference `tPArrow` by class `TPArrow` and two references `src` and `trg` with corresponding multiplicity constraints.
5. Add superclass `Arrow'` to class `TPArrow`.
6. Merge superclass `Arrow` and superclass `Arrow'`.

Although the sequence above is also a valid sequence of evolution steps, the detected sequence in Example 5.2.1 is more efficient as it has less steps. Note that depending on the specification of corresponding migration steps, migration results may differ.

Remark 5.2.2 (Termination of Procedure 5.2.1). We assume that we only consider triple matches of rules that actually have an effect (i.e. MM_i is not isomorph to MM_{i+1}). This implies that the delta between MM_i and MM_n becomes smaller with each rule application. Even if not proven here, this should be ensured by the commuting condition of triple matches. That MM_{max-i} converges against MM_n can be shown by the existence of a morphism $s_i : MM_{max-i-1} \rightarrow MM_{max-i}$.



Note that the outer square trivially commutes, as it is a pullback (Step 4).

Next, an alternative approach to detecting evolution sequences based on span rules is discussed.

5.3 Detecting Evolution Steps with Span Rules

We did not succeed in decomposing a meta-model span by span rules (see Figure 5.7, morphism $d : MM_{max} \rightarrow MM_3$ could not be constructed using category theory).

However, meta-model cospans can also be decomposed by graph transformation rules based on spans. Again, we assume an appropriate meta-model cospan as given.

5.3. Detecting Evolution Steps with Span Rules

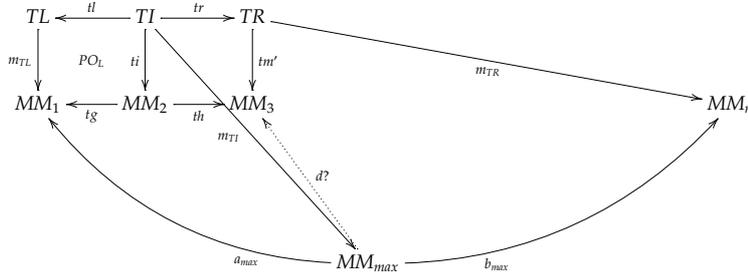


Figure 5.7: Meta-model span decomposition problem

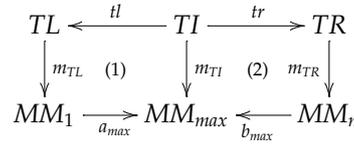
First, we define triple matches for span rules in meta-model cospans:

Definition 5.3.1 (Triple match of a span transformation rule).

Assume a cospan $MM_1 \xrightarrow{a} MM_{max} \xleftarrow{b} MM_n$ relating three graph (respectively objects in a (weak) adhesive (HLR) category) MM_1 , MM_{max} and MM_n : a triple match (m_{TL}, m_{TI}, m_{TR}) of a

span rule $tp = TL \xleftarrow{tl} TI \xrightarrow{tr} TR$ is defined by three matches $m_{TL} : TL \rightarrow MM_1$, $m_{TI} : TI \rightarrow MM_{max}$ and $m_{TR} : TR \rightarrow MM_n$ so that squares (1) and (2) in the diagram on the right commute.

TRIPLE MATCH OF A
SPAN
TRANSFORMATION
RULE



Next, we revise Procedure 5.2.1 for span rules:

Procedure 5.3.1 (Type graph cospan decomposition by span rules).

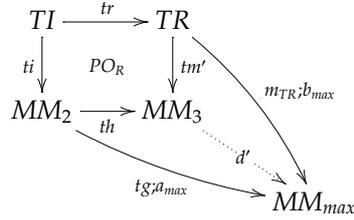
Given type graph cospan $TCS = M_1 \xrightarrow{a_{max}} M_{max} \xleftarrow{b_{max}} M_n$ (meta-model cospan) and a set of span rules R in a (weak) adhesive (HLR) category, then cospan TCS may be decomposed into a sequence of span rule applications (evolution steps) by applying the following procedure (see figure below):

TYPE GRAPH
COSPAN
DECOMPOSITION BY
SPAN RULES

1. Find a triple match (m_{TL}, m_{TI}, m_{TR}) of a span rule $tp \in R$ in cospan TCS . Rules are again matched using priorities.
2. Apply⁴ span rule tp to type graph MM_1 at match $m_{TL} : TL \rightarrow MM_1$. Note, if the rule cannot be applied continue with Step 1. If the cospan SqPO approach is used, it must be ensured in addition that the rule does not delete unmatched context elements.
3. Obtain $d' : MM_3 \rightarrow MM_{max}$ as mediating morphism from the right pushout PO_R of tp 's rule application.

⁴DPO or SqPO approach

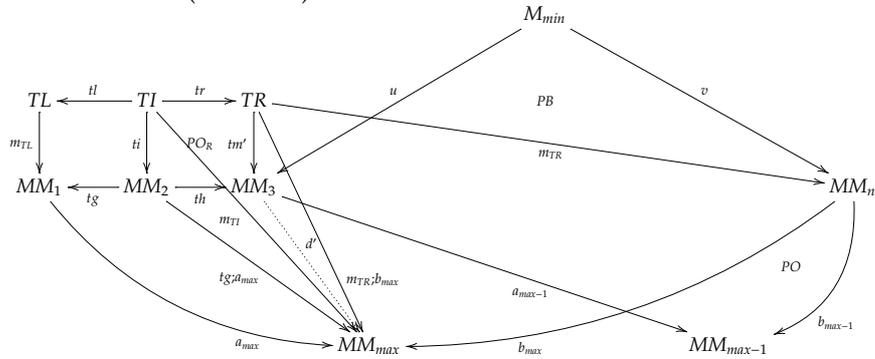
5. DETECTING EVOLUTION STEPS BY GRAPH TRANSFORMATION RULES



The inner square commutes since it is a pushout. The outer square commutes:

$$\begin{array}{lcl}
 tl; m_{TL}; a_{max} & = & m_{TI} \\
 ti; tg; a_{max} & = & m_{TI} \\
 tr; m_{TR}; b_{max} & = & m_{TI}
 \end{array}
 \left| \begin{array}{l}
 \text{triple match (left square)} \\
 \text{left square of transformation commutes} \\
 \text{triple match (right square)}
 \end{array} \right.$$

4. Construct $MM_3 \xleftarrow{u} MM_{min} \xrightarrow{v} MM_n$ as a pullback of $MM_3 \xrightarrow{d'} MM_{max} \xleftarrow{b_{max}} MM_n$ (as before).
5. Construct a new type graph cospan $MM_3 \xrightarrow{a_{max-1}} MM_{max-1} \xleftarrow{b_{max-1}} MM_n$ as a pushout of $MM_3 \xleftarrow{u} MM_{min} \xrightarrow{v} MM_n$ (as before).
6. Continue to decompose cospan $MM_3 \xrightarrow{a_{max-1}} MM_{max-1} \xleftarrow{b_{max-1}} MM_n$ (recursion). Stop if u and v are isomorphisms or no more triple matches are found (as before).



5.4 Advantages of Cospan Rule Detection

Triple matches need to be detected in both approaches i.e. independent of if cospan rules or span rules are used.

A possible implementation could use a flattening construction that could be formalized by a Grothendieck construction in category theory [11]. Graph transformation rules can be transferred to graphs where

the rule morphisms are presented by special edges⁵. Such graphs can be matched using the same techniques that are used to match the left-hand-side of graph transformation rules.

However, there may be a more efficient way to find triple matches without using a flattening construction. The idea is to match only the intermediate graph TI of a rule first and then deduce partial matches for graphs TL and graph TR . Afterward, such partial matches may be efficiently completed. This can be done in both approaches for elements that are only injectively mapped by morphisms of the meta-model cospan. We think this technique is more efficient in case of cospan rules, as the intermediate graphs in cospan rules contain more elements compared to equivalent span rules. In case of span rules, the probability of false-positive matches is higher than in case of cospan rules, meaning more often detected matches cannot be completed to triple matches (see Example 5.4.1). Furthermore, non-injective matches are more often required in case of span rules than in case of cospan rules (see Example 5.4.2).

DETECTING TRIPLE MATCHES

Example 5.4.1 (Number of false positive matches). In this example, we assume that a triple match is detected by first matching rule graph TI and deducing partial matches thereafter. Typings are again neglected as in the previous examples. Figure 5.8 shows a meta-model cospan and a span and a cospan rule. Both rules specify a “Move Attribute” operation. While graph TI of the cospan rule can only be matched once in the meta-model cospan, graph TI of the span rule can be matched four times. Furthermore, matches m_{TL} and m_{TR} can be completely deduced in the case of the cospan rule, while only partial matches m_{TL} and m_{TR} can be deduced in the case of the span rule.

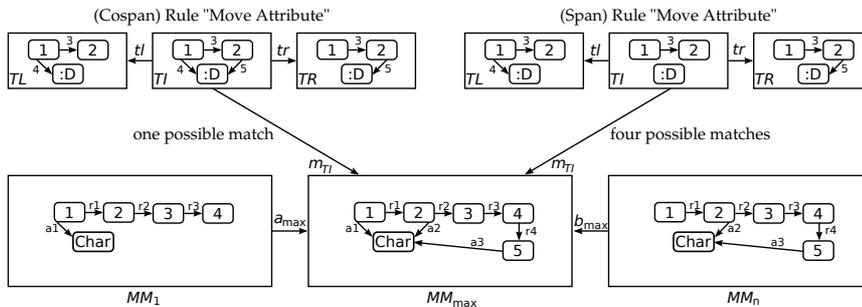


Figure 5.8: Finding triple matches: number of false positive matches

⁵similar to [66]

Example 5.4.2 (Number of non-injective matches). In this example, we assume again that a triple match is detected by first matching rule graph TI and deducing partial matches thereafter. Types are neglected analog to the previous examples. Figure 5.9 shows a type graph cospan and a span and a cospan rule. Both rules specify a “Merge Class” operation. While graph TI of the cospan rule can be matched injectively in the meta-model cospan, graph TI of the span rule must be matched non-injectively.

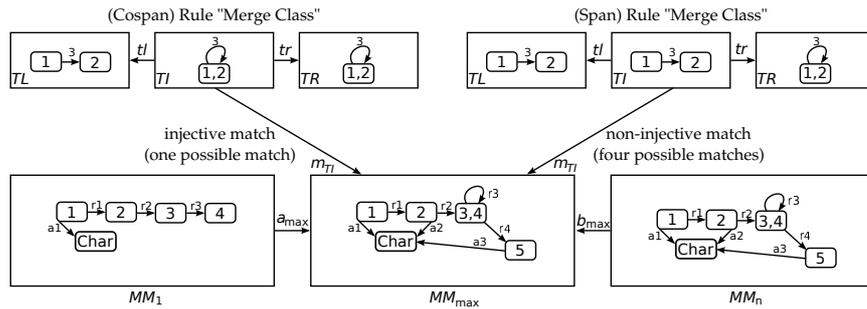


Figure 5.9: Finding triple matches: number of non-injective matches

In this chapter, a new approach to detect meta-model changes by cospan rules has been introduced and compared to an alternative (also new) approach based on span rules. The approach has been illustrated on a well-known model co-evolution example from the literature.

Coupled Transformations based on Graph Transformations

In the following, coupled transformations¹ are introduced. They can be used to evolve meta-models and migrate models correspondingly. First, different categorical constructions are discussed before a standard construction that fits well with meta-model evolution with model migration challenge is chosen. In addition, we justify the decision to use graph transformations based on cospans instead of traditional span-based transformations. This chapter is based on [92, 94, 136].

6.1 Coupled Transformations

We define a coupled transformation by two coupled cospan graph transformations, where each graph of the instance graph transformation (e.g. specifying a model migration step) is coupled to the corresponding graph of the type graph transformation (e.g. specifying a meta-model evolution step) by a morphism. Example 6.1.1 shows a sample coupled transformation. The transformation in Figure 6.1 shows a meta-model evolution step and the transformation in Figure 6.2 shows a model migration step. While the upper transformation is typed by the meta-meta-model (type graph of the type graphs), the lower transformation is typed by the upper transformation.

¹In our previous work also called co-transformations [94, 95, 136].

Example 6.1.1 (Example coupled transformation in category GSpec). Figure 6.1 shows the meta-model evolution step of Example 2.6.1 on page 22 (without adding the new weight attribute). A reference is replaced by a class and two corresponding references. DPF predicates are used to equip the graphs with constraints. In the sample transformation, type names correspond to identifiers. Therefore suitable identifiers for new elements have been chosen.

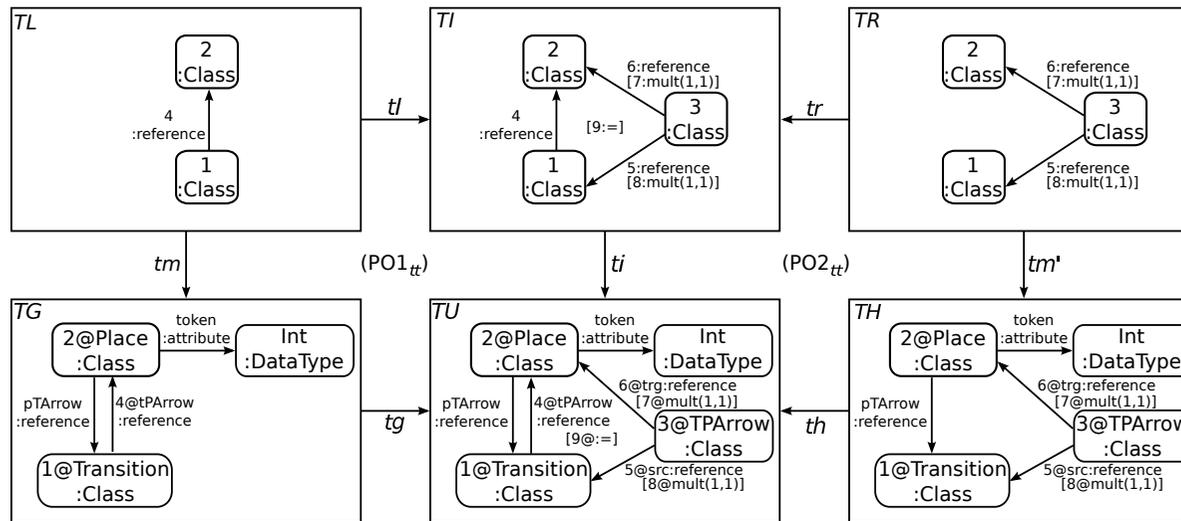


Figure 6.1: A meta-model evolution step

Figure 6.2 shows a similar migration step as in Example 2.6.1. In contrast to the model in Example 2.6.1, the model has two outgoing arrows from the transition to two different places.

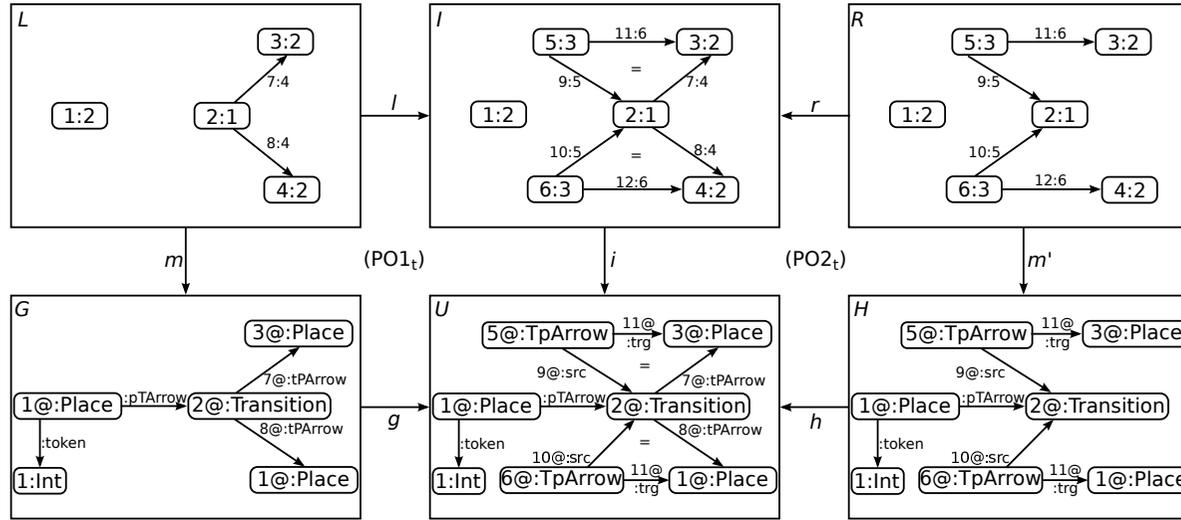


Figure 6.2: A model migration step

Note: while the vertex labeled by "1:2" is not required in the transformation, it is added so that the match of the coupled transformation rule is complete (see Definition 6.1.2). Furthermore, graph I and graph U each contain two commuting subgraphs corresponding to the constraint defined in the evolution rule of Figure 6.1.

elements that cannot be typed anymore. Usually, it is desired that coupled transformation rules are at least deletion-reflecting to ensure that instance graphs can be typed after migration.

Example 6.1.2 (A coupled transformation rule in category GSpec). This example builds on Example 6.1.1 and shows the corresponding coupled transformation rule.

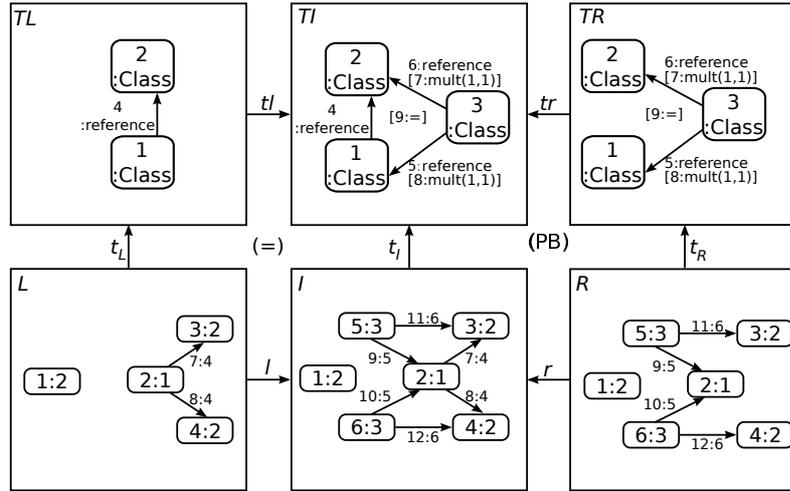


Figure 6.4: Coupled transformation rule

The coupled transformation rule is deletion-reflecting, as its right square is a pullback.

A match of a coupled transformation rule is defined by two coupled matches:

Definition 6.1.2 (Match of a coupled transformation rule).

A match (tm, m) of a coupled transformation rule (tp, p) is given by a match $tm : TL \rightarrow TG$ of the (evolution) rule tp and a match $m : L \rightarrow G$ of the (migration) rule p such that the two matches together with typing morphisms $t_L : L \rightarrow TL$ and $t_G : G \rightarrow TG$ constitute a commuting square: $t_L; tm = m; t_G$.

If $G \xleftarrow{m} L \xrightarrow{t_L} TL$ is a pullback then the match (tm, m) is called *complete*.

A complete coupled transformation match ensures that all elements of the instance graph G that are typed by the match of the evolution rule are considered in the migration step. This means in particular that it is not possible that the type of any element not considered in the migration step is deleted, merged or split.

MATCH OF A
COUPLED
TRANSFORMATION
RULE

$$\begin{array}{ccc}
 TL & \xrightarrow{tm} & TG \\
 \uparrow t_L & = & \uparrow t_G \\
 L & \xrightarrow{m} & G
 \end{array}$$

Example 6.1.3 (A coupled transformation match in category GSpec). This example builds on Example 6.1.1 and shows the corresponding coupled match.

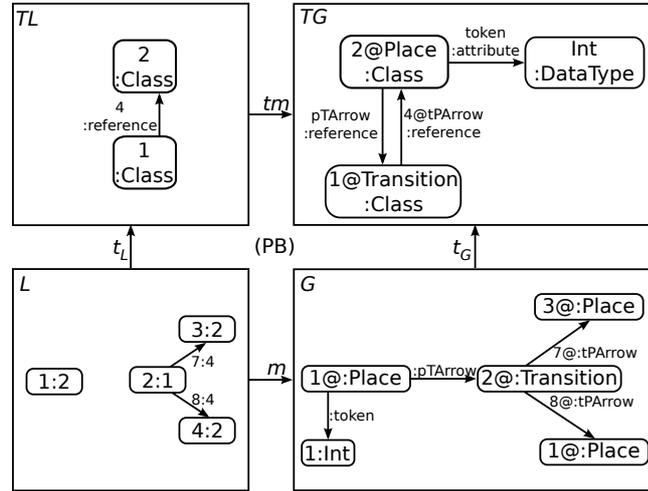


Figure 6.5: Coupled transformation match

The coupled match is complete since its diagram is a pullback.

A coupled transformation rule applied via a coupled transformation match results in a coupled transformation. Formally, we define a coupled transformation as follows:

Definition 6.1.3 (Coupled transformation). A coupled transformation (tt, t) consists of either two cospan DPO transformations (see Definition 4.4.1) or two cospan SqPO transformations (see Definition 4.5.1), $tt : TG \xrightarrow{tp, tm} TH$ and $t : G \xrightarrow{p, m} H$, that apply a coupled transformation rule (tp, p) at match (tm, m) to type graph TG and its instance graph G , such that there are morphisms $t_U : U \rightarrow TU$ and $t_H : H \rightarrow TH$ and all faces of Figure 6.3 commute.

COUPLED
TRANSFORMATION

If such a coupled transformation (tt, t) is used for meta-model evolution with model migration, the type graph transformation $tt : TG \xrightarrow{tp, tm} TH$ is called an *evolution step*, while the instance graph transformation $t : G \xrightarrow{p, m} H$ is called a *migration step* wrt. tt .

Up to now, we have not considered under which conditions coupled transformation rules are applicable. A necessary, but not a sufficient precondition for an applicable coupled transformation rule is that both cospan

graph transformation rules can be applied. In the next sections, sufficient conditions are examined, which result in different constructions of coupled transformations. In the following, we will assume an evolution step (i.e. the top face of the double cube in Figure 6.3) and an instance graph G typed by the type graph TG being evolved as given. First, we will consider different constructions for the left cube of the coupled transformation before we examine its right cube. Constructions and proofs are presented on the level of (weak) adhesive HLR categories. The constructions rely on the properties presented in Chapter 4.

6.2 Constructing Coupled Transformations (Left Part)

In this section, two different procedures to construct the left part of coupled transformations are examined. The first relies on the (weak) VK property (Property 8, respectively Property 10), the second one relies on the (weak) special PO-PB property (Property 9, respectively Property 11). For each procedure, the requirements in different types of adhesive categories are stated as well as which variants are possible. In both procedures, a pushout in the top face and a type graph TG with instance graph G (see Figure 6.6) is assumed. To be able to construct the right cube afterward, the procedures ensure that the right face of the cube in Figure 6.6 is constructed as a pullback.

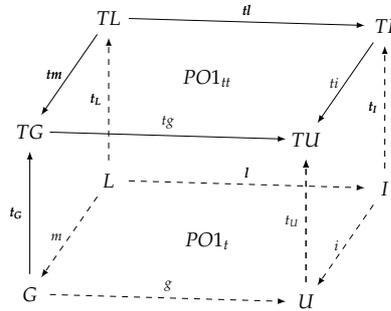


Figure 6.6: Left cube of coupled transformation

Constructing the left part of a coupled transformation as VK cube:

Procedure 6.2.1 (Left cube construction based on (weak) VK property).

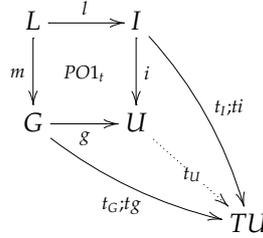
Let \mathbf{C} be a (weak) adhesive HLR category, given a pushout square $TL \xrightarrow{tl} TI \xrightarrow{ti} TU \xleftarrow{tg} TG \xleftarrow{tm} TL$ with $tm : \text{or}/(\text{and})^2 tl \in \mathbf{M}$ and an object $G \in \mathbf{C}$

²“And” in weak adhesive HLR categories. If pullbacks along arbitrary morphisms do not exist, tm must be a \mathbf{M} -morphism.

LEFT CUBE
CONSTRUCTION
BASED ON (WEAK)
VK PROPERTY

typed by $t_G : G \rightarrow TG$ (see Figure 6.6). A VK cube can be constructed as follows:

1. Construct the left face of the cube by taking a pullback $G \xleftarrow{m} L \xrightarrow{tl} TL$ of $G \xrightarrow{t_G} TG \xleftarrow{tm} TL$. This results in a coupled match (tm, m) that is complete.
2. Construct the back face of the cube by choosing a suitable pullback complement $L \xrightarrow{l} I \xrightarrow{ti} TI$ of $L \xrightarrow{tl} TL \xrightarrow{ti} TI$. Notice that pullback complements must not always exist. Furthermore, notice that variants are possible in this step as pullback complements are not unique.
3. Construct pushout $G \xrightarrow{g} U \xleftarrow{i} I$ of span $G \xleftarrow{m} L \xrightarrow{l} I$ ($PO1_t$).
4. Construct $t_U : U \rightarrow TU$ as a mediating morphism of pushout $PO1_t$ with $g; t_U = t_G; tg$ and $i; t_U = t_I; ti$ (see figure below) such that the cube commutes.



PROCEDURE 6.2.1 **Lemma 6.2.1.** *Procedure 6.2.1 is well-defined if the pullback complement in Step 2 exists.*

Proof. Step 1 can be constructed if tm is a \mathbf{M} -morphism (see Definitions 4.1.3 and 4.1.4) or if arbitrary pullbacks exist. We assume that Step 2 can be constructed. Step 3 can be constructed: if tm (or tl) is a \mathbf{M} -Morphism then also m (or tl) is a \mathbf{M} -Morphism (Property 5). In (weak) adhesive categories, pushouts along \mathbf{M} -Morphisms exist. For Step 4, it is required to show that t_U can be constructed as a mediating morphism. Therefore, the outer square in the pushout diagram of Procedure 6.2.1 needs to commute:

$$\begin{array}{l}
 l; t_I; ti \\
 = t_L; tl; ti \\
 = t_L; tm; tg \\
 = m; t_G; tg
 \end{array} \left| \begin{array}{l}
 \text{back face is commuting (Step 2)} \\
 \text{top face is commuting by assumption} \\
 \text{left face is commuting (Step 1)}
 \end{array} \right.$$

Next, it can be shown that the constructed commuting cube is a VK cube (which implies that its right face is a pullback as required). Morphisms $tm : TL \rightarrow TG$ and/or $tl : TL \rightarrow TI$ are \mathbf{M} -morphisms. Because the top face and the bottom face of the cube are pushouts and the left face and the back face are pullbacks, the (weak) VK property (Property 8, respectively Property 10) applies:

6.2. Constructing Coupled Transformations (Left Part)

1. in adhesive (HLR) categories, one \mathbf{M} -morphisms is sufficient that Property 8 applies. This means either morphism tm or tl need to be in \mathbf{M} .
2. in weak adhesive HLR categories, tm and tl must be \mathbf{M} -morphisms so that Property 10 applies.

□

Table 6.1 summarizes all requirements and variants of Procedure 6.2.1 for different types of categories. While weak adhesive categories require left-linear rules and “injective” matches (i.e. \mathbf{M} -morphisms), adhesive HLR categories require the existence of arbitrary pullbacks if “non-injective” matches should be supported. In Table 6.1 and all following tables, concepts supported by the construction are marked by “✓” and requirements not already supported in the kind of category by “•”. Supported properties are meant to be additive. This means if a table row contains a marker in column “non-left-linear rules” and “non-monic matches,” rules are allowed to be both. If there is a marker missing e.g. in column “left-linear rules,” then there may be another record with weaker preconditions. Note, we assume the type graph transformation as given even if supported concepts apply always for the type graph and instance graph rule if not stated differently.

Table 6.1: Procedure 6.2.1 requirements and variants

	Supports			Requires (in addition)		
	Left-linear rules	Non-left-linear rules	Non-monic matches	Existence of suitable PBCs along \mathbf{M} -morphisms (back face)	Existence of suitable PBCs along arbitrary morphisms (back face)	Existence of PBs along arbitrary morphisms (left face)
Adhesive categories	✓		✓	•		
		✓			•	
Adhesive HLR categories	✓			•		
	✓		✓	•		•
		✓			•	
Weak adhesive HLR categories	✓			•		

Constructing the left part of a coupled transformation with help from the (weak) special PO-PB property:

Procedure 6.2.2 (Left cube construction based on (weak) special PB-PO Property). Let \mathbf{C} be a (weak) adhesive HLR category, given a pushout square $TL \xrightarrow{tl} TI \xrightarrow{ti} TU \xleftarrow{tg} TG \xleftarrow{tm} TL$ with $tm, ti \in \mathbf{M}$ and an object $G \in \mathbf{C}$ typed by

LEFT CUBE
CONSTRUCTION
BASED ON (WEAK)
SPECIAL PB-PO
PROPERTY

$t_G : G \rightarrow TG$ (see Figure 6.6). A commuting cube with a pushout in the bottom face and a pullback in the right face can be constructed as follows:

1. Construct the left face of the cube by taking a pullback $G \xleftarrow{m} L \xrightarrow{t_L} TL$ of $G \xrightarrow{t_G} TG \xleftarrow{tm} TL$ (as before).
2. Construct the back face of the cube by complementing $L \xrightarrow{t_L} TL \xrightarrow{t_I} TI$ to a commuting square $L \xrightarrow{t_L} TL \xrightarrow{t_I} TI \xleftarrow{i} I \xleftarrow{l} L$. Notice that even more variants are possible in this step (compared with the previous Step 2) and that such a complement always exists (e.g. $L \xrightarrow{l=id} I \xrightarrow{t_I=t_L \circ t_I} TI$).
3. Construct pushout $G \xrightarrow{s} U \xleftarrow{i} I$ ($PO1_i$) of span $G \xleftarrow{m} L \xrightarrow{l} I$ (as before).
4. Construct $t_U : U \rightarrow TU$ by using the pushout property of the bottom face so that the cube commutes (as before).

PROCEDURE 6.2.2 **Lemma 6.2.2.** *Procedure 6.2.2 is well-defined.*
 IS WELL-DEFINED

Proof. Because tm is a \mathbf{M} -morphism, Step 1 can always be constructed. Step 2 can always be constructed. Step 3 and Step 4 can always be constructed as before (see Proof of Lemma 6.2.2). It remains to show that the right face of the cube is a pullback. Morphism $tm : TL \rightarrow TG$ is a \mathbf{M} -morphism. It follows that morphism $m : L \rightarrow G$ is also a \mathbf{M} -morphisms, as \mathbf{M} -morphisms are stable under pullbacks (Property 5). In addition, it follows that morphisms ti and i are in \mathbf{M} , as \mathbf{M} -morphisms are also stable under pushout (Property 6). The top and bottom faces are pushouts along \mathbf{M} -morphisms, hence they are also pullbacks (Property 4). Consider figures below: the left face (1) and the top face (2) of the cube can be composed to a pullback (1+2).

$$\begin{array}{ccc}
 L & \xrightarrow{t_L} & TL & \xrightarrow{t_I} & TI \\
 \downarrow m & & \downarrow tm & & \downarrow ti \\
 G & \xrightarrow{t_G} & TG & \xrightarrow{t_g} & TU
 \end{array}
 \quad
 \begin{array}{ccc}
 L & \xrightarrow{l} & I & \xrightarrow{t_I} & TI \\
 \downarrow m & & \downarrow i & & \downarrow ti \\
 G & \xrightarrow{s} & U & \xrightarrow{t_U} & TU
 \end{array}$$

Since the bottom face of the cube (3) is a pushout (Step 3), it can be deduced:

1. In any adhesive HLR category with pullbacks that the right face (4) (in Figure 6.6) of the cube is a pullback as required by using the special PO-PB property (Property 9).

6.3. Constructing Coupled Transformations (Right Part)

- In any weak adhesive HLR category, it must in addition be assumed that morphism l has been constructed as a \mathbf{M} -morphism (e.g. as id). Then the weak special PO-PB property (Property 11) applies and the right face (4) of the cube is a pullback as required. That morphism g is then also a \mathbf{M} -morphism follows by $l \in \mathbf{M}$ (Property 6).

□

Table 6.2 summarizes all requirements and variants of the construction above. In comparison to Procedure 6.2.1, Procedure 6.2.2 is only applicable if “injective” matching (i.e. $tm, m \in \mathbf{M}$) is used. Therefore the construction does not require the existence of a suitable pullback complement in the back face of the cube.

Table 6.2: Procedure 6.2.2 requirements and variants

	Supports			Requires (i. a.)		
	Left-linear rules	Non-left-linear rules	Non-monic matches	Existence of suitable PBCs along \mathbf{M} -morphisms (back face)	Existence of suitable PBCs along arbitrary morphisms (back face)	Existence of PBs along arbitrary morphisms (Theorem A.1.1)
Adhesive categories	✓	✓				
Adhesive HLR categories	✓					
Weak adhesive HLR categories	✓	(✓) ¹				•

¹: Type graph transformations are not required to be left-linear, while instance graph transformations are.

6.3 Constructing Coupled Transformations (Right Part)

In this section, two different procedures to construct the right part of coupled transformations are examined. The first one uses again the (weak) VK property (Property 8, respectively Property 10), and the second one uses the stability of FPBCs along pullbacks (Property 12). For each construction, the requirements in different types of adhesive categories are stated as well as which construction variants are possible as before. In both constructions, a PO respectively a PB (with a final complement) in the top face and a PB in the left face (see left cube construction) of the cube (see Figure 6.7) to be constructed are assumed as given.

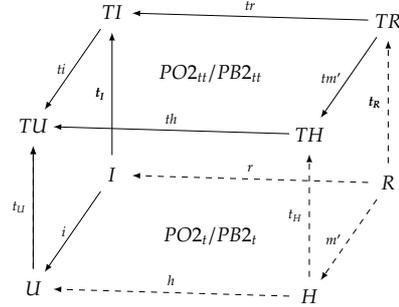


Figure 6.7: Right cube of coupled transformation

Next, the right part of a coupled transformation is constructed as VK cube. The (weak) VK Property only allows for deducing that the bottom face of a commuting cube with a pushout in its top face is also a pushout (and not a FPBC in the general case). Therefore, the procedure is restricted to the cospan DPO approach. Hence, graph transformation rules are right linear i.e. $tr, r \in \mathbf{M}$.

Procedure 6.3.1 (Right cube construction based on (weak) VK property).

RIGHT CUBE
CONSTRUCTION
BASED ON (WEAK)
VK PROPERTY

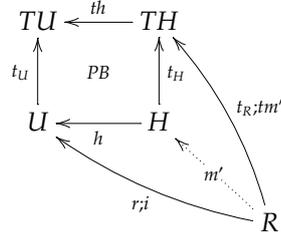
Let \mathbf{C} be a (weak) adhesive HLR category, given a pushout square $TR \xrightarrow{tr} TI \xrightarrow{ti} TH \xleftarrow{th} TR \xleftarrow{tm'} TR$ with tr (and³ ti) $\in \mathbf{M}$ and a pullback square $I \xrightarrow{t_i} TI \xrightarrow{t_{ii}} TU \xleftarrow{t_{iu}} U \xleftarrow{i} I$ (see Figure 6.7). A VK cube can be constructed as follows:

1. Construct the back face of the cube by taking a pullback $I \xleftarrow{r} R \xrightarrow{t_R} TR$ of $I \xrightarrow{t_i} TI \xleftarrow{tr} TR$. Note that the coupled transformation rule is deletion-reflecting (see Definition 6.1.1) by this construction. In addition, this construction is fully deterministic i.e. the construction is unique up to isomorph.
2. Construct also the front face of the cube by taking a pullback $U \xleftarrow{h} H \xrightarrow{t_H} TH$ of $U \xrightarrow{t_{iu}} TU \xleftarrow{th} TH$.
3. Construct $m' : R \rightarrow H$ as a mediating morphism of the front face pullback (see figure below) with $m'; h = r; i$ and $m'; t_H = t_R; tm'$ such

³“And” in weak adhesive HLR categories.

6.3. Constructing Coupled Transformations (Right Part)

that the cube commutes.



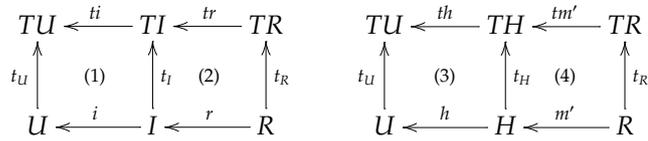
Lemma 6.3.1. *Procedure 6.3.1 is well-defined.*

PROCEDURE 6.3.1
IS WELL-DEFINED

Proof. Step 1 requires only the existence of pullbacks along \mathbf{M} -morphism and $tr \in \mathbf{M}$, as the evolution rule is right-linear. Furthermore, $th \in \mathbf{M}$ since \mathbf{M} -morphism are stable under pushout (Property 6). Therefore, also Step 2 can be applied. Step 3 requires that the pullback diagram of Procedure 6.3.1 commutes:

$$\begin{array}{l} t_R; tm'; th = t_R; tr; ti \\ = r; t_I; t_i \\ = r; i; t_U \end{array} \left| \begin{array}{l} \text{top face is commuting by assumption} \\ \text{back face is commuting (see Step 1)} \\ \text{left face is a commuting (by left cube)} \end{array} \right.$$

It is left to show that the cube is a VK cube. By composing the left face (1) and the back face (2) to a pullback (1+2) and decomposing it by the front face (3), it can be shown that the right face face (4) of the cube is also a pullback (see figures below).



Because all side faces of the commuting cube are pullbacks and the top face is a pushout along a \mathbf{M} -morphism, the (weak) VK property applies:

1. in any adhesive HLR category Property 8 applies and the bottom face is a pushout as required.
2. in any weak adhesive HLR category Property 10 applies if also $tm' \in \mathbf{M}$, i.e., the bottom face is a pushout as required. Morphism $tm' \in \mathbf{M}$ if tm is a \mathbf{M} -morphism: $tm \in \mathbf{M} \implies ti \in \mathbf{M}$ (Property 6), $ti \in \mathbf{M} \implies tm' \in \mathbf{M}$ (Property 4 and 5).

□

Table 6.3 summarizes all requirements and variants of the procedure above. Graph transformation rules are in this construction always right-linear. In weak adhesive HLR categories, furthermore, “injective” matching is required.

Table 6.3: Procedure 6.3.1 requirements and variants

	Supports			Requires (i. a.)	
	Right-linear rules	Non-right-linear rules (cospan SqPO approach)	Non-monic matches	Existence of PBs along monomorphism (back and front faces)	Existence of PBs along arbitrary morphisms (back and front faces)
Adhesive categories	✓		✓		
Adhesive HLR categories	✓		✓		
Weak adhesive HLR categories	✓				

Next, Property 12 is used to construct the right cube with a FPBC in the bottom face. However, the construction below can also be used if evolution steps are given as cospan DPO transformations due to the fact that cospan DPO transformations are always also cospan SqPO transformations.

Procedure 6.3.2 (Right cube construction based on the stability of FPBCs). Let \mathcal{C} be a (weak) adhesive HLR category with arbitrary pullbacks, given a pullback square $TR \xrightarrow{tr} TI \xrightarrow{ti} TU \xleftarrow{th} TH \xleftarrow{tm'} TR$ with $TR \xrightarrow{tm'} TH \xrightarrow{th} TU$ being a FPBC and a pullback square $I \xrightarrow{ti} TI \xrightarrow{ti} TU \xleftarrow{tu} U \xleftarrow{i} I$ (see Figure 6.7). A commuting cube with a pullback in the bottom face with $R \xrightarrow{m'} H \xrightarrow{h} U$ being a FPBC can be constructed as follows:

RIGHT CUBE
CONSTRUCTION
BASED ON THE
STABILITY OF
FPBCs

1. Construct the back face of the cube by taking a pullback $I \xleftarrow{r} R \xrightarrow{tr} TR$ of $TI \xrightarrow{tr} TR \xleftarrow{tr} R$ as in Procedure 6.3.1. Notice that this may require the existence of pullbacks along non \mathbf{M} -morphisms in case of non-right-linear cospan rules.
- 2.-3. Also, construct Step 2 and Step 3 as in Procedure 6.3.1.

PROCEDURE 6.3.2
IS WELL-DEFINED

Lemma 6.3.2. *Procedure 6.3.2 is well-defined.*

Proof. That the cube is commuting and all side faces pullbacks has been shown in the proof of Lemma 6.3.1. Notice that this does not require any \mathbf{M} -morphism. By Property 12, it follows directly that $R \xrightarrow{m'} H \xrightarrow{h} U$ is a FPBC. \square

6.3. Constructing Coupled Transformations (Right Part)

Procedure 6.3.2 only differs from Procedure 6.3.1 in its precondition. This means Procedure 6.3.1 is also applicable if evolution steps are given as cospan SqPO transformations. Table 6.4 summarize all requirements and variants of Procedure 6.3.2.

Table 6.4: Procedure 6.3.2 requirements and variants

	Supports			Requires (i. a.)	
	Right-linear rules	Non-right-linear rules (cospan SqPO approach)	Non-monic matches	Existence of PBs along monomorphism (back and front faces)	Existence of PBs along arbitrary morphisms (back and front faces)
Adhesive categories	✓	✓	✓		
(Weak) Adhesive HLR categories	✓		✓		
		✓	✓		•
(Any category)	✓		✓	•	
		✓	✓		•

6.4 Standard Construction for Coupled Transformations

In this section, the standard procedure to construct coupled transformations is defined based on the procedures presented previously. The standard procedure is used in subsequent chapters. Table 6.5 summarizes all possible constructions for complete coupled transformations by combining the results of Procedure 6.2.1, Procedure 6.2.2 and Procedure 6.3.2. Procedure 6.3.2 is a generalization of Procedure 6.3.1 and therefore does not need to be considered.

Table 6.5: Construction coupled transformations: requirements and variants

	Supports				Requires (in addition)			
	Linear rules	Non-left-linear rules	Non-right-linear rules	Non-monic matches	Existence of suitable PBCs along \mathbf{M} -morphisms (left-back face)	Existence of suitable PBCs along arbitrary morphisms (left-back face)	Existence of PBs along arbitrary morphisms (left face, right-back face, right-front face, Theorem A.1.1)	Left cube constructed by ¹
Adhesive categories	✓		✓	✓	•			1
		✓	✓			•		1
Adhesive HLR categories	✓	✓	✓					2
	✓				•			1
		✓				•		1
		✓	✓			•	•	1
	✓		✓	✓	•			1
	✓						•	2
Weak adhesive HLR categories		✓	✓				•	2
	✓				•			1
	✓		✓		•		•	1
	✓	(✓) ²						2
	✓	(✓) ²	✓				•	2

¹: “1” denotes Procedure 6.2.1, “2” denotes Procedure 6.2.2

²: Type graph transformations are not required to be left-linear while instance graph transformations are.

We define the standard procedure based on Procedure 6.2.2 and Procedure 6.3.2. We choose Procedure 6.2.2 to be more flexible when defining the left side of the coupled transformation rule, even if this choice restricts us to “injective” matching (i.e. $tm, m \in \mathbf{M}$). However, this is useful as already a simple retyping (see Example 6.4.1) may not be expressible by a pullback complement as required in Step 2 of Procedure 6.2.1. Furthermore, Procedure 6.2.2 allows for also having non-left linear evolution rules

in weak adhesive HLR categories. We choose Procedure 6.3.2, as it is a generalization of Procedure 6.3.1.

This choice is also supported by the results of [52]. In [52], four variants of the traditional DPO approach to graph transformation are examined. Beside the usual DPO graph transformation approach with arbitrary matching and right-linear span rules, Habel et al. consider three variations employing injective matching and/or non-right linear span rules. They show that the graph transformation approach with non-right linear span rules and injective matching is the best approach with respect to expressiveness. We support, in the standard construction, the equivalent variant with non-left linear cospan rules and injective matching. Also note that even if the standard construction does not use the (weak) VK property explicitly, it is based on the (weak) VK property due to Property 9, respectively Property 11 (see proof of Proposition A.1.1).

Example 6.4.1 (Retyping in category Graph). Figure 6.8 shows a left side of a simple coupled transformation rule in category Graph_{TG} that retypes one object due to a merge in its type graph.

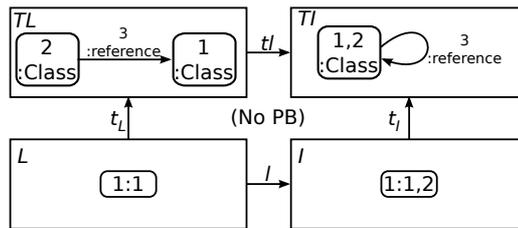


Figure 6.8: Left side of a coupled transformation rule that retypes one object

The diagram in Figure 6.8 is not a pullback⁴ and hence it is not a valid left side of a coupled transformation rule according to Procedure 6.2.1. However, because the diagram is commuting it is a valid left side according to Procedure 6.2.2.

Procedure 6.4.1 (Standard construction for coupled transformation). Let \mathbf{C} be a (weak) adhesive HLR category (with arbitrary pullbacks), given a type graph TG (being an object of \mathbf{C}), a type graph cospan graph transformation $tt : TG \xrightarrow{tp, tm} TH$ with $tm \in \mathbf{M}$ (evolution step, see Figure 6.3), and a graph G (being an object of \mathbf{C}) typed by $t_G : G \rightarrow TG$, the standard construction of a coupled transformation rule (tp, p) with a complete match (tm, m) is defined as follows:

STANDARD
CONSTRUCTION FOR
COUPLED
TRANSFORMATION

1. Construct $G \xleftarrow{m} L \xrightarrow{t_L} TL$ as a pullback of $G \xrightarrow{t_G} TG \xleftarrow{tm} TL$. This results in a complete match (tm, m) .

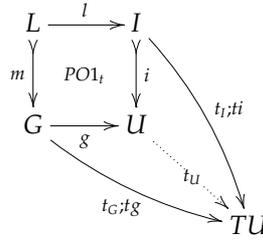
⁴To be a valid pullback, the diagram requires a second node "2:2" in graph L .

6. COUPLED TRANSFORMATIONS BASED ON GRAPH TRANSFORMATIONS

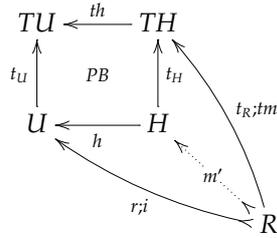
2. Complete $L \xrightarrow{t_L} TL \xrightarrow{t_I} TI$ by $L \xrightarrow{l} I \xrightarrow{t_I} TI$ to a commuting square⁵. If the category is only weak adhesive, choose a \mathbf{M} -morphism for l (e.g. $l = id$).
3. Construct $I \xleftarrow{r} R \xrightarrow{t_R} TR$ as pullback of $I \xrightarrow{t_I} TI \xleftarrow{t_r} TR$. If morphism tr is not a \mathbf{M} -morphism, the category must in addition support pullbacks along arbitrary morphism.

Such a coupled transformation rule (tp, p) with evolution step t and match (tm, m) can be completed to a coupled transformation (tt, t) by the following steps:

4. Construct $G \xrightarrow{g} U \xleftarrow{i} I$ as pushout of $G \xleftarrow{m} L \xrightarrow{l} I$ ($PO1_t$ in Figure 6.3). Type morphism $t_U : U \rightarrow TU$ is obtained as mediating morphism of this pushout as shown in the figure below.



5. Construct $TH \xleftarrow{t_H} H \xrightarrow{h} U$ as pullback of $U \xrightarrow{t_U} TU \xleftarrow{t_H} TH$. Morphism $m' : R \rightarrow H$ is obtained as mediating morphism of this pullback as shown in the figure below.



PROCEDURE 6.4.1
IS WELL-DEFINED

Theorem 6.4.1 (Procedure 6.4.1 is well-defined). *Let \mathbf{C} be a (weak) adhesive HLR category and TG an object in \mathbf{C} , an evolution step $tt : TG \xrightarrow{tp, tm} TH$ with match $tm \in \mathbf{M}$ (see Figure 6.3) and morphism $t_G : G \rightarrow TG$, Procedure 6.4.1 yields a match-complete and deletion-reflecting coupled transformation rule (tp, p) . Furthermore, rule (tp, p) is applicable, i.e. rule (tp, p) can always be completed to a*

⁵Here the procedure leaves some freedom to define a desired construction.

6.4. Standard Construction for Coupled Transformations

coupled transformation (tt, t) in a canonical way by Procedure 6.4.1 under the following assumptions⁶:

- If \mathbf{C} is an adhesive HLR category, it has arbitrary pullbacks.
- If \mathbf{C} is a weak adhesive HLR category, $l : L \rightarrow I$ can be constructed as M -morphism.

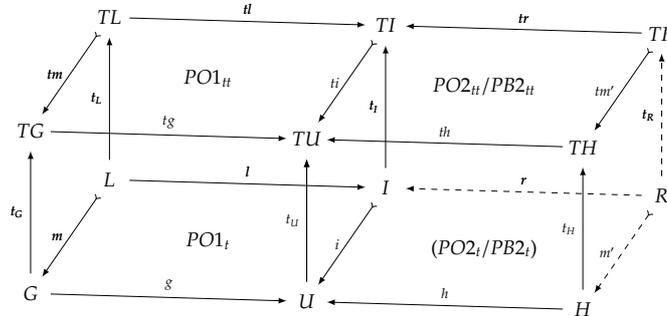
Proof. The proof follows directly from Lemma 6.2.2 and Lemma 6.3.2. \square

The standard construction defined by Procedure 6.4.1 provides language designers with the following possibilities to evolve meta-models and migrate models correspondingly: type elements can be *created*, *deleted*, *merged* or *split*, instance elements can be *created*⁷ or *merged*⁸. However, as the right part of a coupled transformation rule is determined by a pullback (Step 3), only exactly those instance elements are deleted that have a corresponding type which is deleted. If a type is split, instance elements are corresponding split. Further, *deletions* or *splittings* must be defined in a usual graph transformation on the instance graph level thereafter.

SUPPORTED
PRIMITIVE
OPERATIONS OF
PROCEDURE 6.4.1

Remark 6.4.1 (Incomplete and alternative construction).

1. In Procedure 6.4.1, Step 3 is fully deterministic and can be omitted in practice i.e. it is enough to take the pullback in Step 5. In any case, the incomplete double cube (see figure below) can be uniquely completed to a coupled transformation.



2. In practice, the instance graph transformations can alternatively be applied directly. By the uniqueness of POCs⁹/FPBCs $(PO2_t/PB2_t)$, this yields isomorphic results. In this case, the typing morphisms $t_H : H \rightarrow TH$ have to be constructed differently (in case of cospan DPO transformations e.g. by reducing $t_U : U \rightarrow TU$ to t_H).

⁶To support non-right linear rules and to use Lemma 6.2.2 in the proof.
⁷independent of if their types are created or preserved
⁸in adhesive (HLR) categories only
⁹In graph transformations, only unique pushout complements are used.

In Procedure 6.4.1, the pushout complement, respectively FPBC is not constructed directly. This implies that the instance graph transformation can be constructed as long as the type graph transformation can be constructed.

APPLICABILITY OF
MIGRATION
TRANSFORMATIONS

Corollary 6.4.1 (Applicability of migration transformations). *Let \mathbf{C} be a (weak) adhesive HLR category with:*

- arbitrary pullbacks if \mathbf{C} is an adhesive HLR category or
- $l : L \rightarrow I$ being constructed as \mathbf{M} -morphism if \mathbf{C} is a weak adhesive HLR category.

If the evolution step (with match $tm : TL \rightarrow TG$ being a \mathbf{M} -morphism) of a coupled transformation satisfies the (cospan) gluing condition (respectively conflict-freeness condition), its migration step can be constructed by Procedure 6.4.1 for any instance object, i.e. the gluing condition (respectively conflict-freeness condition) does not need to be verified for migration steps.

Proof. The proof follows directly from Theorem 6.4.1 and Procedure 6.4.1. The second pushout $PO2_t$ of the migration transformation $t : G \xrightarrow{p,m} H$ is not constructed by a complement but by a pullback in the right-front face (Step 5 of Procedure 6.4.1, see Figure 6.3). Because we assume that coupled transformations are applied in categories where required pushouts ($PO1_t$) and pullbacks (right back and front faces) always exist, Corollary 6.4.1 holds. \square

In addition, we can state that applied coupled transformation rules have a unique result:

UNIQUE RESULT OF
COUPLED
TRANSFORMATION

Corollary 6.4.2 (Unique result of coupled transformation). *Let \mathbf{C} be a (weak) adhesive HLR category, applying a deletion-reflecting coupled transformation rule (tp, p) along a complete match (tm, m) with tm and m being \mathbf{M} -morphisms results in a unique coupled transformation.*

Proof. Graph transformations have unique transformation results [36], hence transformations $tt : TG \xrightarrow{tp,tm} TH$ (top faces, see Figure 6.3) and $t : G \xrightarrow{p,m} H$ (bottom faces, see Figure 6.3) have unique results each. Hence, it has to be shown only that transformation t is uniquely typed i.e. that $t_U : U \rightarrow TU$ and $t_H : H \rightarrow TH$ are unique. (All other morphism are given.) It has been shown before that transformation t can be applied by Steps 4 and 5 of Procedure 6.4.1. Thus, it follows directly that $t_U : U \rightarrow TU$ and $t_H : H \rightarrow TH$ are unique:

1. The uniqueness of morphism t_U follows by Step 4 of Procedure 6.4.1, i.e. by the fact that the left cube is commuting and by the universal pushout property of the bottom face $PO1_t$.

6.4. Standard Construction for Coupled Transformations

- The uniqueness of morphism t_H follows by Step 5 of Procedure 6.4.1, i.e. the uniqueness of t_H follows directly from the fact that $U \xleftarrow{h} H \xrightarrow{t_H} TH$ is a pullback of $U \xrightarrow{t_U} TU \xleftarrow{t_H} TH$, t_U is unique and th is unique (as th is constructed by a graph transformation).

□

Remark 6.4.2 (Deletion-reflecting coupled transformations). Note that deletion-reflection for coupled transformations applies to elements of deleted types only. However, a migration rule may also delete or split elements of preserved types. In this case, the alternative construction described in Remark 6.4.1 has to be used. Figure 6.9 shows an extended migration transformation. Due to the uniqueness of pushout complements (cospan DPO approach), respectively FPBCs (cospan SqPO approach), pushout (1 + 2), respectively pullback (1 + 2), can also be constructed in one step i.e. by one complement construction.

$$\begin{array}{ccccccc}
 L & \xrightarrow{l} & I & \xleftarrow{r} & R & \xleftarrow{r'} & R' \\
 \downarrow m & & \downarrow i & (1) & \downarrow m' & (2) & \downarrow m'' \\
 G & \xrightarrow{g} & U & \xleftarrow{h} & H & \xleftarrow{h'} & H'
 \end{array}$$

Figure 6.9: Extended migration graph transformation t'

Figure 6.10 shows the migration transformation of Figure 6.9 in an extended coupled transformation.

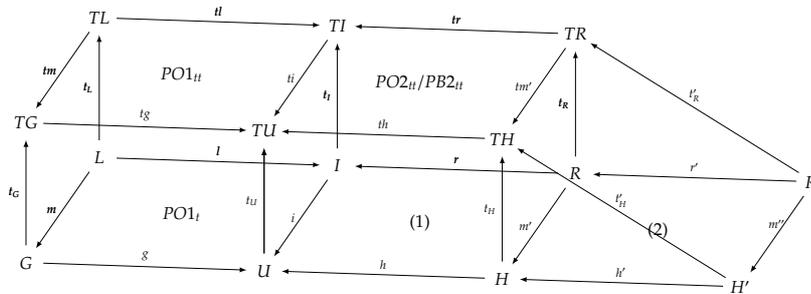


Figure 6.10: Extended coupled transformation

The migration rule is in this case $p' = L \xrightarrow{l} I \xleftarrow{r'} R'$. Note that for such migration rules, Corollary 6.4.1 cannot be applied. This means the gluing condition, respectively the conflict freeness condition has to be checked

for migration steps. This is undesired if we want to migrate many models in a batch. Therefore, we propose to delete or split such elements by traditional graph transformations after applying coupled transformations. In some categories, however, the problem does not occur if the cospan SqPO approach is used: in category Graph e.g. the existence of FPBCs is ensured if matches are injective [24].

To put it in a nutshell, Procedure 6.4.1 allows language engineers to define model migration rules that are always applicable if the corresponding meta-model evolution rule is applicable (see Corollary 6.4.1). Migration rules are model-specific, as the left-hand side of each migration rule in Procedure 6.4.1 is created by a pullback of $G \xrightarrow{t_G} TG \xleftarrow{t_m} TL$ in Step 1. Furthermore, Step 2 of Procedure 6.4.1 leaves space for migration variants, as the left face of the coupled transformation rule requires to be a commuting square only (see Figure 6.11). The amount of addition and merges can be specified according to *custom* needs. Without this freedom, only *one* fixed model migration variant would be possible, which may not reflect the language designer's intention of model migration. The right face of a coupled transformation rule, however, does not allow variants, as it is fully determined by the pullback construction in Step 3 of Procedure 6.4.1.

$$\begin{array}{ccccc}
 TL & \xrightarrow{tl} & TI & \xleftarrow{tr} & TR \\
 \uparrow t_L & & \uparrow t_I & & \uparrow t_R \\
 & \text{variants} & & PB & \\
 & \text{possible} & & & \\
 L_{G_1} & \xrightarrow{l_{G_1}} & I_{G_1} & \xleftarrow{r_{G_1}} & R_{G_1}
 \end{array}$$

Figure 6.11: Coupled transformation rule for graph G_1

Furthermore, all three types of model migration approaches (see Chapter 2) fit into the framework of coupled transformations:

- Trivially, coupled transformation rules can be considered as *coupled operators* that evolve meta-models and migrate models correspondingly.
- In Chapter 5, meta-model changes have been detected with the help of graph transformation rules. Such detected evolution steps can be completed to coupled transformations according to Procedure 6.4.1. Hence, coupled transformations also suites as formalization of *matching approaches*.

- If morphisms $tm : TL \rightarrow TG$, $ti : TI \rightarrow TU$ and $tm' : TR \rightarrow TH$ are identity morphisms, migrations rules are directly typed by the meta-model cospan. This formalization fits with *manual specification approaches*. How morphism $L \rightarrow I$ can be systematically constructed by rules is explained in Chapter 7.

6.5 Span versus Cospan Transformations

While it is common to work with graph transformations based on span rules, cospan transformations have been rarely used. In this section, we provide arguments as to why we use cospan rules instead of span rules. That cospan rules have advantages when evolution steps should be detected has already been examined in Chapter 5.

Cospan transformations in contrast to span transformations create first and delete thereafter. This is an advantage for synchronizing model migrations. In Procedure 6.4.1, a coupled transformation rule is constructed by first completing $L \xrightarrow{t_L} TL \xrightarrow{t_I} TI$ to a complete square (Step 2) and by second taking a pullback (Step 3). In case of span rules, this order also needs to be reversed, which has undesired side effects (see Example 6.5.1).

MIGRATION
SYNCHRONIZATION

Example 6.5.1 (Migration Synchronization in category Graph). Figure 6.13 and Figure 6.12 each show a coupled transformation rule in category Graph that moves an attribute along a reference and attribute values correspondingly along links. While the coupled transformation rule in Figure 6.13 employs cospan rules, the coupled transformation rule in Figure 6.12 employs span rules.

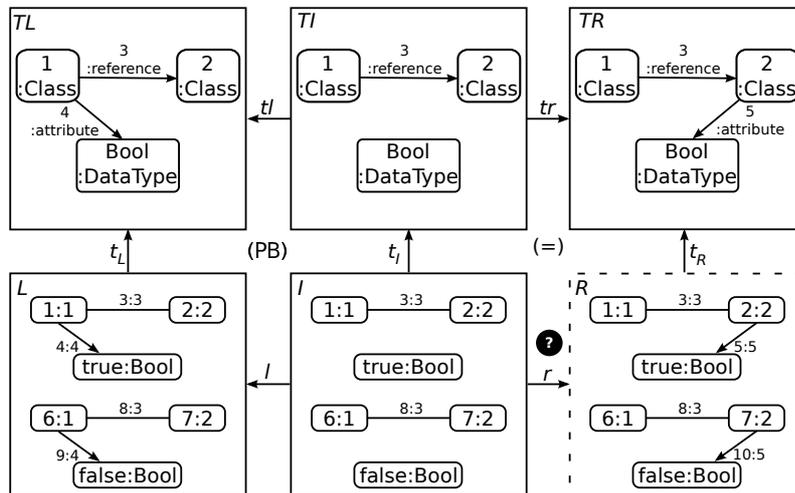


Figure 6.12: Coupled transformation rule with spans

In Figure 6.12, the right square of the coupled transformation rule has to be constructed to a commuting square. The construction of graph R requires considering the whole morphism $l : I \rightarrow L$, as attribute values are not connected in graph I due to the pullback in the left face.

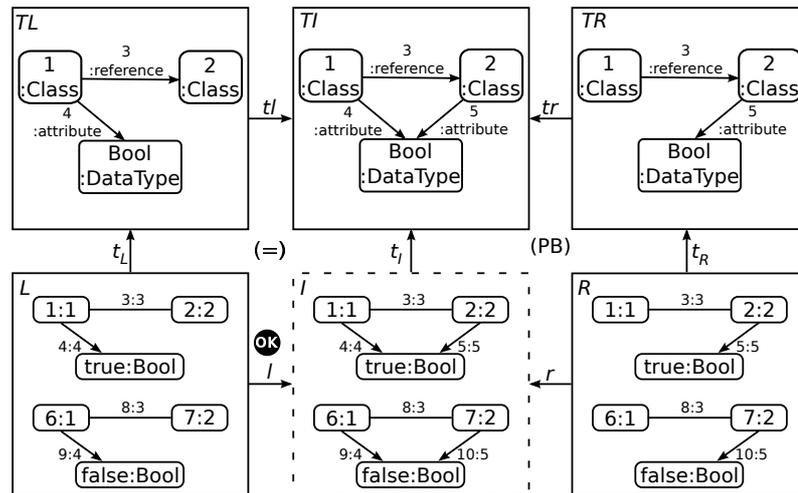


Figure 6.13: Coupled transformation rule with cospans

In Figure 6.13, the left square of the coupled transformation rule has to be constructed to a commuting square. Constructing graph I is easy because attribute values are still connected to their source vertices in graph L .

The second advantage is closely connected to the previous one. Cospan transformations do not only have advantages when synchronizing type and instance graph transformations, instance graph transformations can also be easier validated using constraints. Recall Example 6.1.1, in which a reference is replaced by an association class. In the desired migration step, each link typed by the reference to be deleted is replaced by an object of the new association class and two corresponding links. Herein, “corresponding” can be formalized by adding a commuting condition to intermediate type graph TI (see Figure 6.1, visualized by a DPF predicate “[=]”). The condition demands that all corresponding patterns in the instance graph I commute (see Example 6.5.2). Graph I and hence rule t can be validated using a suitable validator for the added constraint. In span transformations, intermediate graphs instead contain only the elements to be preserved. As a result, a commuting condition as in Example 6.1.1 cannot be formulated using type graph TI only.

Example 6.5.2 (Migration rule validation in category GSpec). Figure 6.14 shows the intermediate graphs of the coupled transformation rule of

the “Replace reference by reference class” example introduced in Example 6.1.1. Figure 6.15 shows the intermediate graphs for the same example for the case that span rules would have been used.

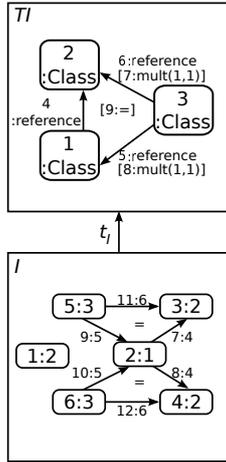


Figure 6.14: Intermediate rule graphs with cospan rules

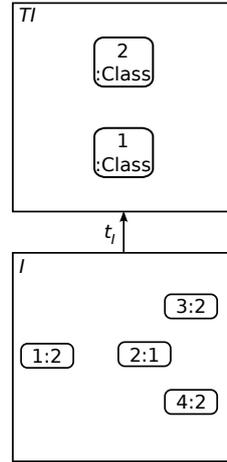


Figure 6.15: Intermediate rule graphs with span rules

Figure 6.14 uses a constraint in type graph TG (formalized by a DPF predicate, see Example 3.3.3) to ensure the desired behavior of the migration step. In Figure 6.15, this is not possible, as the intermediate graphs contain only the common parts.

In this chapter, coupled transformations have been introduced as coupled graph transformations. Different possibilities to construct such coupled transformations in (weak) adhesive (HLR) categories have been discussed. Finally, a standard construction has been defined which fits well with the challenge of model co-evolution.

Model Migration Schemes based on Coupled Transformations

In the previous chapter, coupled transformations have been introduced that are suitable for evolving meta-models and migrating models correspondingly. In addition, Procedure 6.4.1 has been developed, which can be used to complete meta-model evolution steps by migration steps to coupled transformations. Unfortunately, migration rules are *model-specific* and *not completely determined* according to this procedure, i.e. migration rules need to be specified individually for each model to be migrated. In this chapter, we develop a procedure to generate such model-specific rules systematically on the basis of *model-independent rule schemes*, called *model migration schemes*. Furthermore, a construction of *default migration schemes* that are *customizable* to specific needs of language engineers is presented.

7.1 Migration by Amalgamated Graph Transformations

In this section, we develop a strategy to specify coupled transformation rules by *model-independent* rule schemes. This allows us to migrate *all* models over a meta-model along a *generic* migration specification. The presented strategy constructs coupled transformation rules that satisfy the preconditions of Theorem 6.4.1. Such model-independent rule schemes are called *model migration schemes*. Their formalization is based on amalgamated graph transformations [17, 133, 135]. Amalgamation in graph transformation is a technique to construct graph transformation rules with similar structures from smaller rules before they are applied as usual. This has been highly beneficial in practice. Consider e.g. the well-known “Pull

AMALGAMATED
GRAPH
TRANSFORMATIONS

Up Attribute” refactoring: an attribute with a specific name and type occurs in all subclasses of a selected class. They shall be replaced by one new attribute in the selected class. Because a class can have an arbitrary number of subclasses, there are two possibilities to describe this refactoring with “classical” graph transformations (i.e. without amalgamation): (1) Apply a graph transformation system containing several graph transformation rules according to some control structure. We need to address the application logic in addition including the classical challenges¹ of graph transformation systems. (2) Define a separate rule for each possible number of subclasses that may occur. This solution is simple but not scalable. Both possibilities have drawbacks that are avoidable by using amalgamation.

An amalgamated graph transformation is constructed by an *interaction scheme* consisting of a *set of multi-rules* and a *set of kernel rules* (see e.g. [133]). Each multi-rule specifies a multi-object structure that shall be applied as often as possible. Kernel rules are used to synchronize multi-rule matches. Therefore, kernel and multi-rules are related by injective morphisms. After all matches of multi-rules have been found, the amalgamated graph transformation rule is constructed by taking a rule copy for each multi-rule match and gluing them at kernel rule matches. This results in a “classical” graph transformation rule that applies all multi- and kernel rule matches in *one parallel step*. This technique can be used e.g. to describe the “Pull Up Attribute” refactoring by one multi- and one kernel rule: (1) The multi-rule deletes an attribute (with specific name and type) in each subclass and creates a copy of it in the selected class. (2) The kernel rule declares all attribute copies in multi-rules to be identical. For model migrations, we have to adapt the existing amalgamation concept to cover the following additional requirements:

REQUIRED
ADAPTATIONS OF
THE AMALGAMATED
GRAPH
TRANSFORMATION
TECHNIQUE

1. Match-complete coverings of models have to be guaranteed (see Definition 6.1.2).
2. Multi-rules may be synchronized not only by kernel rules but also by match overlappings. This is sufficient if multi-rule applications have to be synchronized along preserved elements only. It avoids that all² possible kernel rules need to be defined.
3. The interaction scheme should be more flexible. Usually, all multi-rule matches overlapping in kernel rule matches are considered. We want to prioritize multi-rules such that different migration cases can be distinguished. Therefore, a specification of different *matching strategies* would be useful.

¹i.e. confluence and termination

²In traditional application areas of amalgamated graph transformations, usually not all possible kernel rules are required. For example, in the case of the “Pull Up Attribute,” refactoring one kernel rule is enough. In tools e.g. Henshin [3], the kernel rule would be matched first, before its match would be extended to multi-rule matches as often as possible.

4. So far, amalgamated graph rules are normal transformation rules that are typed over one (fixed) type graph. Because we want to construct coupled transformations, model migration rules are not typed over type graphs but over evolution rules.

7.2 Migration Rules from Migration Schemes

In this section, we first explain a *new* construction of coupled transformations by rule schemes, called *migration schemes*, on a semi-formal level before we elaborate the construction on a formal level thereafter. We focus on amalgamating the *left* part of the migration rule, as its *right* part is completely determined according to Procedure 6.4.1. Hence, we consider interaction schemes of non-deleting rules only. These are called *migration schemes*. Migration schemes are used to synchronize creation and merge operations, even if its rule matches may overlap. The construction of amalgamated migration rules works as follows (see Figure 7.1):

1. The match-complete left-hand side L of the migration rule is constructed by a pullback (Step 1 in Procedure 6.4.1).
2. All multi-rules are matched to L as often as possible using a suitable *matching strategy*. For each match found, the corresponding multi-rule is copied³, considering rule elements matched to the same instance model element to be identical. However, not all possible multi-rule matches need to be taken, e.g. a match that is completely covered by a match of a larger rule may be ignored by the used matching strategy.
3. All multi-rule matches are pairwise checked for overlappings. Overlapping parts of multi-rule matches may be covered by kernel rule matches to synchronize the creation of new elements in an interleaved application of multi-rules.
4. The migration rule morphism $l : L \rightarrow I$ is completed by the unmatched part of rule graph L using a standard construction. It solely adds unmatched elements of L and retypes them along corresponding type changes (if needed).
5. The construction of a coupled transformation rule is completed by Step 3 in Procedure 6.4.1.

CONSTRUCTING
COUPLED
TRANSFORMATION
RULES

³Note that implementing a rule “copy” may lead to reusing existing model elements. On the formal level, however, proper rule copies are considered.

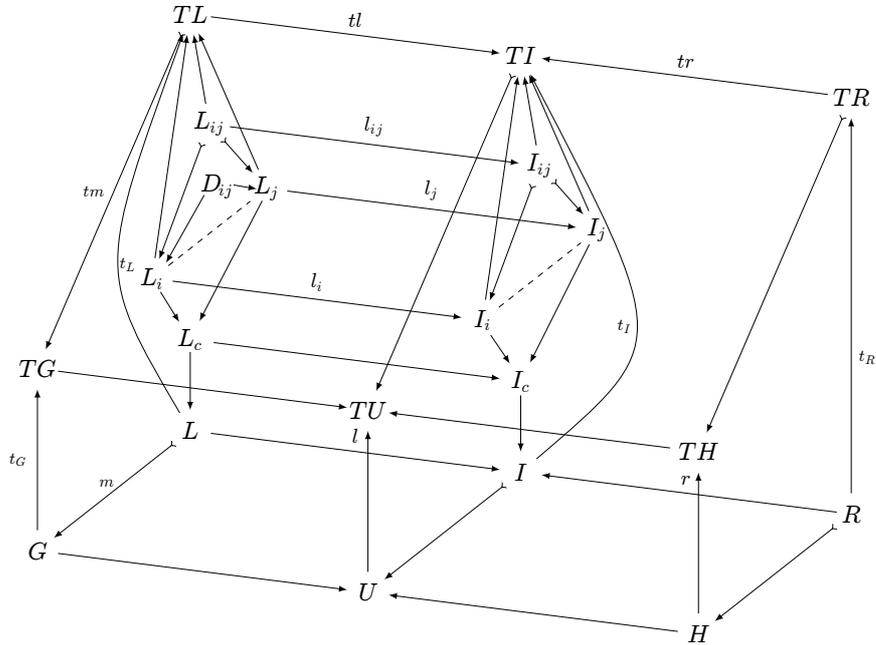


Figure 7.1: Model migration by the application of a migration scheme

Example 7.2.1 shows a sample migration scheme for a variant of meta-model refactoring “Move Attribute”:

Example 7.2.1 (Migration scheme: “Move Attribute”⁴ in category Graph). Figure 7.2 shows a meta-model from a Moore automata that is evolved to a Mealy automata [86] in Figure 7.3. The meta-models are presented in concrete syntax, however, it is straight forward to formalize them in terms of (typed) multi-graphs as in previous examples. Moore and Mealy automata are the basic computation models of finite state machines. A state machine that uses entry actions, so that its outputs depend on the current state only, is called a Moore automaton. In contrast, a Mealy automaton is a state machine whose output values are determined by both its current state and the current inputs. Therefore, in Moore automata, each state is labeled by an output value while in Mealy automata, each transition is labeled by an output value. The selected computation model for a state machine influences its design. However, there are no general indications on which computation model is better. The choice of the computation

⁴The example is based on an example that has been originally presented by Herrmannsdörfer et al. to illustrate the Cope [55, 57] tool.

7.2. Migration Rules from Migration Schemes

model depends on the application domain and on personal preferences of the modeler. Fortunately, Moore automata can be transformed into equivalent Mealy automata by moving the effects of each state to all of its incoming transitions (step from Figure 7.2 to Figure 7.3).



Figure 7.2: Original meta-model

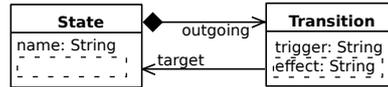


Figure 7.3: Move attribute

The migration scheme to migrate models from Moore to Mealy automata is shown in Figure 7.4.

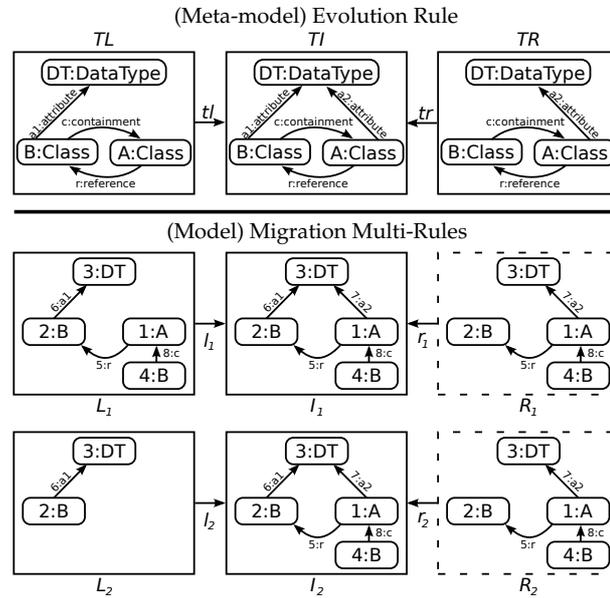


Figure 7.4: Evolution rule and migration scheme: “Move Attribute”

In the upper part of Figure 7.4, the evolution rule for the “Move Attribute” operation is presented. In the lower part of Figure 7.4, the (model) migration scheme is presented consisting of two multi-rules. Kernel rules are not required here. Furthermore, right-hand sides of migration rules are shown (in dashed boxes) for illustration purposes only. The first multi-rule is nearly isomorphic to the (meta-model) evolution rule and moves an attribute from one object to a referred one (in the opposite direction of a link). The second rule considers a case where the attribute value cannot be

7. MODEL MIGRATION SCHEMES BASED ON COUPLED TRANSFORMATIONS

moved to a target object since it does not exist. Hence, a new target object is created. In addition, in this variant of the “Move Attribute” refactoring also a new container is created for the target object. The migration scheme is used together with a matching strategy matching the second multi-rule only if the first one cannot be matched. This means the second rule has a lower priority as the first rule.

Figure 7.5 shows an instance model of the meta-model in Figure 7.2. Its migration (see Figure 7.6) illustrates all of the challenges the migration scheme in Figure 7.4 can handle. The model describes a simple Moore automaton for processing jobs in a batch. A job may be e.g. a phone bill that needs to be computed for each customer of a phone company. When a new job arrives, the automaton leaves the *Idle* state and computes the job in the *Busy* state. If the system runs into any problems while generating a new job or computing a job, it simply restarts the application (a very simple and often used recovery strategy). To migrate the model correspondingly to the meta-model in Figure 7.3, several effects have to be moved:

- Effect `compute` needs to be moved.
- Effect `restart` needs to be moved and duplicated.
- Effect `init` needs to be moved to a transition coming from a new state.

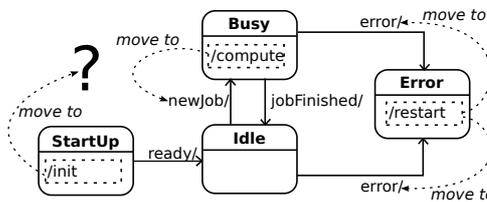


Figure 7.5: State machine for simple batch processing

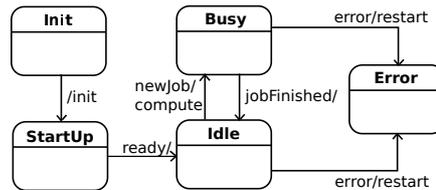


Figure 7.6: State machine for simple batch processing (migrated)

The amalgamated migration rule constructed to co-evolve the model of Figure 7.5 uses three copies of the first multi-rule and one copy of the second multi-rule. Because two matches of the first multi-rule overlap in state *Error* and attribute *restart*, the corresponding copies are accordingly glued.

Earlier, we stated that cospan transformations have advantages over span transformations, when migrating models. Here, we can see that the amalgamation procedure would also not work well with span rules. It amalgamates the left part of a migration rule, which would be the right part if we had span rules. In case of example “Move Attribute,” this would mean that all attribute values would be disconnected from objects first. In the amalgamation procedure, these values have to again be reconnected to objects, missing the information about the original connections.

In the following, we formally define migration schemes and the *amalgamation* procedure (see Procedure 7.2.1) for category Graph to build the left part of a coupled transformation rule. A generalization of the procedure can be deduced analog as earlier. The proof that the construction is well-defined is presented on the level of (weak) adhesive (HLR) categories.

First, rule morphisms are defined to formally embed kernel rules in multi-rules:

Definition 7.2.1 (Rule morphism).

A rule morphism $k: l_1 \rightarrow l_2$ between two non-deleting rules $l_1: L_1 \rightarrow I_1$ and $l_2: L_2 \rightarrow I_2$ is a pair (a, b) of injective morphisms $a: L_1 \rightarrow L_2$ and $b: I_1 \rightarrow I_2$ such that (1) is commuting.

$$\begin{array}{ccc} L_1 & \xrightarrow{l_1} & I_1 \\ a \downarrow & (1) & \downarrow b \\ L_2 & \xrightarrow{l_2} & I_2 \end{array}$$

RULE MORPHISM

An interaction scheme is defined as two sets of multi- and kernel rules related by rule morphisms:

Definition 7.2.2 (Interaction scheme of non-deleting rules). An interaction scheme of non-deleting rules is a triple (K, M, KM) with K being a set of non-deleting rules called *kernel rules*, M a set of non-deleting rules called *multi-rules*, and KM a set of rule morphisms $k: l_k \rightarrow l_m$ with $l_k \in K$ and $l_m \in M$ called *kernel morphisms*.

INTERACTION
SCHEME OF
NON-DELETING
RULES

Finally, we define migration schemes based on such interaction schemes:

Definition 7.2.3 (Migration scheme). A *migration scheme* is a tuple $((K, M, KM) \triangleright tp, prio)$ consisting of an interaction scheme (K, M, KM) of non-deleting rules typed over evolution rule tp and a priority function $prio: M \rightarrow \mathbb{N}$ for rules in M . Each rule $l_i \in M \cup K$, l_i is well-typed by the non-deleting part $tl: TL \rightarrow TI$ of rule tp , i.e. $t_{L_i}; tl = l_i; t_{I_i}$. Furthermore, each kernel morphism $(a, b) \in KM$ between kernel rule $l_K: L_K \rightarrow I_K$ and multi-rule $l_M: L_M \rightarrow I_M$ is type compatible i.e. $a; t_{L_M} = t_{L_K}$ and $b; t_{I_M} = t_{I_K}$.

MIGRATION
SCHEME

$$\begin{array}{ccc} TL & \xrightarrow{tl} & TI \\ \uparrow t_{L_i} & = & \uparrow t_{I_i} \\ L_i & \xrightarrow{l_i} & I_i \end{array} \qquad \begin{array}{ccccc} & t_{L_K} & L_K & \xrightarrow{l_K} & I_K & t_{I_K} & & \\ & & \downarrow a & & \downarrow b & & & \\ TL & \xleftarrow{=} & & \xrightarrow{=} & & \xrightarrow{=} & TI \\ & t_{L_M} & L_M & \xrightarrow{l_M} & I_M & t_{I_M} & & \end{array}$$

7. MODEL MIGRATION SCHEMES BASED ON COUPLED TRANSFORMATIONS

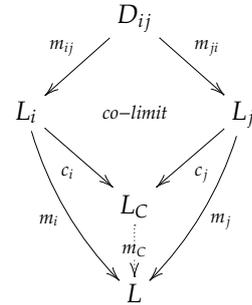
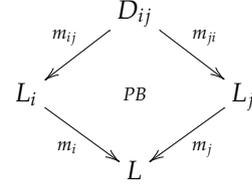
For each pair of rules l_1 and l_2 in M , we may have zero or finitely many triples of a kernel rule (l_k, k_1, k_2) with $l_k : L_k \rightarrow I_k$ in K and kernel morphisms $k_1 : l_k \rightarrow l_1$ and $k_2 : l_k \rightarrow l_2$ in KM .

Migration schemes are applied in the following amalgamation procedure, which is an improved version of the corresponding construction in [95], taking ideas from [80] into account.

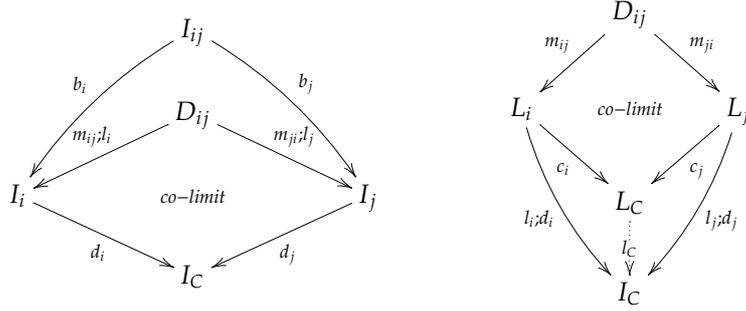
RULE
AMALGAMATION
FOR MIGRATION
SCHEMES

Procedure 7.2.1 (Rule amalgamation for migration schemes). Given a migration scheme $((K, M, KM) \triangleright tp, prio)$, an evolution step $tt : TG \xrightarrow{tp, tm} TH$ and a graph G with typing $t_G : G \rightarrow TG$: the non-deleting part of the migration rule $l : L \rightarrow I$ for G can be constructed by the following amalgamation procedure (see also Figure 7.1 and Figure 7.7):

1. Construct a pullback $G \xleftarrow{m} L \xrightarrow{t_l} TL$ of $G \xrightarrow{t_G} TG \xleftarrow{tm} TL$.
2. Match rules of M to L along a chosen matching strategy. The default strategy matches the left-hand sides of all multi-rules $l_u, l_v \in M$ injectively as often as possible with one restriction: let m_u be the match of rule l_u to G and m_v be the match of rule l_v to G , if one match is completely covered by the other one, i.e. $m_u(L_u) \supseteq m_v(L_v)$, match m_v is used only if $prio(L_v) \geq prio(L_u)$. The result is a set M' consisting of pairs (l_i, m_i) with each $l_i : L_i \rightarrow I_i$ being a copy of a rule in M and with $m_i : L_i \rightarrow L$ being the corresponding match for $1 \leq i \leq n$. If no match has been found, stop the procedure here with $l = id$ and $t_l = t_L; tl$ constructing trivially a commuting diagram. Otherwise, continue with Step 3.
3. For each two pairs $(l_i, m_i), (l_j, m_j) \in M'$, with $1 \leq i < j \leq n$, construct the overlapping part as a pullback $L_i \xleftarrow{m_{ij}} D_{ij} \xrightarrow{m_{ji}} L_j$ of the cospan $L_i \xrightarrow{m_i} L \xleftarrow{m_j} L_j$. By d_L , we denote the diagram consisting of all spans $L_i \xleftarrow{m_{ij}} D_{ij} \xrightarrow{m_{ji}} L_j$, $1 \leq i < j \leq n$.
4. Construct L_C as a co-limit graph of diagram d_L . For all $1 \leq i < j \leq n$, we obtain cospans $L_i \xrightarrow{c_i} L_C \xleftarrow{c_j} L_j$ with $m_{ij}; c_i = m_{ji}; c_j$. Moreover, there exists a unique mediating morphism $m_C : L_C \rightarrow L$ with $c_i; m_C = m_i$ for all $1 \leq i \leq n$, as cospans $L_i \xrightarrow{m_i} L \xleftarrow{m_j} L_j$ constitute a commutative co-cone of the diagram d_L , by construction.

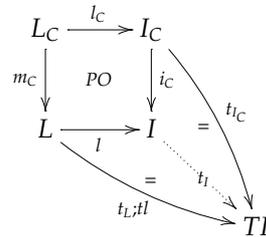


5. Next, we construct a diagram d_I : For each two pairs $(l_i, m_i), (l_j, m_j) \in M'$ with $l_i : L_i \rightarrow I_i$ and $l_j : L_j \rightarrow I_j$, $1 \leq i < j \leq n$:
- We add all interfaces I_i and I_j to d_I .
 - We relate such interfaces by adding all spans $I_i \xleftarrow{m_{ij}^i} D_{ij} \xrightarrow{m_{ij}^j} I_j$, $1 \leq i < j \leq n$ synchronizing overlapping matches.
 - We add further spans to d_I synchronizing kernel rule matches: For any triple (l_{ij}, k_i, k_j) of a kernel rule $l_{ij} : L_{ij} \rightarrow I_{ij}$ in K and kernel morphisms $k_i = (a_i, b_i) : l_{ij} \rightarrow l_i$ and $k_j = (a_j, b_j) : l_{ij} \rightarrow l_j$ in KM that relates multi-rule matches in M' such that $a_i; m_i = a_j; m_j$, we add a fresh copy of I_{ij} together with the relating span $I_i \xleftarrow{b_i} I_{ij} \xrightarrow{b_j} I_j$.
6. Construct I_C by the co-limit of diagram d_I . For all $1 \leq i < j \leq n$, we obtain a cospan $I_i \xrightarrow{d_i} I_C \xleftarrow{d_j} I_j$ with $m_{ij}; l_i; d_i = m_{ji}; l_j; d_j$ and $b_i; d_i = b_j; d_j$. This ensures that morphisms $l_i; d_i : L_i \rightarrow I_C$ constitute a commutative co-cone of the diagram d_L . Thus, there exists a unique mediating morphisms $l_C : L_C \rightarrow I_C$ with $c_i; l_C = l_i; d_i$ for all $1 \leq i \leq n$.



7. Construct the non-deleting rule $l : L \rightarrow I$ by building the pushout of span $L \xleftarrow{m_C} L_C \xrightarrow{l_C} I_C$.
8. Construct the typing morphism $t_I : I \rightarrow TI$:

- We can deduce the existence of a unique mediating morphism $t_{I_C} : I_C \rightarrow TI$ from the co-limit of diagram d_I in Step 6.
- The existence of unique mediating morphism $t_I : I \rightarrow TI$ we can deduce from the pushout in Step 7.



migration multi-rule copies are used in the migration scheme. Hence, the migration scheme itself consists already of rule copies before any migration step is amalgamated. However, we do not elaborate this extension here.

Procedure 7.2.1 has a (worse) complexity of $O(n^2)$, where n is the number of found multi-rule matches. In practice, the complexity may be better as kernel rules only need to be matched when multi-rule matches overlap. In the worst case, all multi-rule matches completely overlap. Then, all of these matches have to be checked pairwise for kernel rule matches. This case, however, is highly unlikely. We assume that the real bottleneck of this approach is the match finding problem, which is a general challenge in the field of graph transformations. Finding matches of multi-rules, however, can be performed in parallel.

Now we show that Procedure 7.2.1 indeed provides the required result.

Theorem 7.2.1 (Procedure 7.2.1 is well-defined).

Let \mathbf{C} be a (weak) adhesive HLR category with arbitrary pushouts⁵: given a migration scheme $((K, M, KM) \triangleright tp, prio)$, an evolution step $tt : TG \xrightarrow{tp, tm} TH$ and a morphism $t_G : G \rightarrow TG$, the left face of the coupled transformation rule for G can be constructed by Procedure 7.2.1 so that:

1. the coupled transformation rule is match-complete and
2. the left face of the coupled transformation rule is commuting i.e. $t_L; tl = l; t_l$.

Proof.

1. The pullback construction in Step 1 of Procedure 7.2.1 ensures that the match is complete and that the typing morphism t_L exists.
2. We have to show that the left back face of the double cube is commuting:
 - a) There exists a unique typing morphism $t_{L_C} : L_C \rightarrow TL$: a unique morphism $m_C : L_C \rightarrow L$ is deduced by the co-limit construction in Step 4 of Procedure 7.2.1. Hence, t_{L_C} is uniquely given by $t_{L_C} = m_C; t_L$.
 - b) There exists a unique typing morphism $t_{I_C} : I_C \rightarrow TI$: by showing that all graphs in diagram d_I are well-typed over TI , the existence of a unique typing morphism t_{I_C} is ensured by the co-limit construction over diagram d_I in Step 6 of Procedure 7.2.1. The definition of migration schemes ensures that I_i, I_j as well as I_{ij} are well-typed. Hence, all D_{ij} have to be shown to be

⁵ Although not proven here, the existence of arbitrary pushouts is only required if merging of model elements should be supported.

Example 7.2.2 (Migration scheme: “Introduce Container” in category Graph). This example takes up Example 7.2.1 and considers the addition of container StateMachine. In some modeling frameworks, as e.g. EMF, all model elements are required to belong to some container. Figure 7.8 shows the concrete evolution step (depicting the added container in a box).

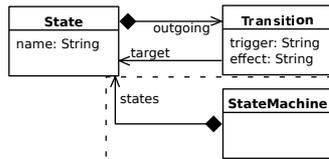


Figure 7.8: Evolution step: “Introduce Container”

Figure 7.9 shows the evolution rule and a migration scheme for the meta-model evolution step above. In the evolution rule, a new container class is created. The migration scheme consists of one multi- and one kernel rule. The multi-rule introduces a new container object for each object that can be contained in the container class. A kernel rule is used to specify that all of these newly introduced container objects are identical. Since the container object shall be unique over the whole model, the left-hand-side of the kernel rule is the empty graph.

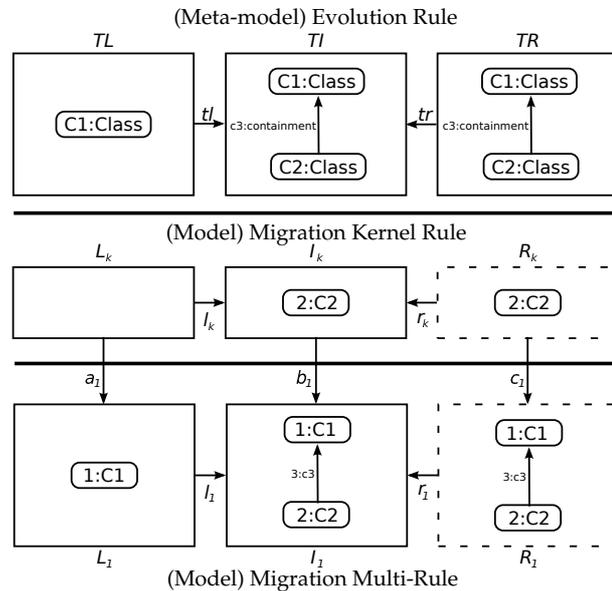


Figure 7.9: Evolution rule and migration scheme: “Introduce Container”

7. MODEL MIGRATION SCHEMES BASED ON COUPLED TRANSFORMATIONS

Figure 7.10 sketches an application of the migration scheme for the evolution step shown above. The models are shown in concrete syntax, while the rule is shown in abstract syntax. Because the model is only changed on the level of the abstract syntax, the added new elements are denoted in model U by dashed lines.

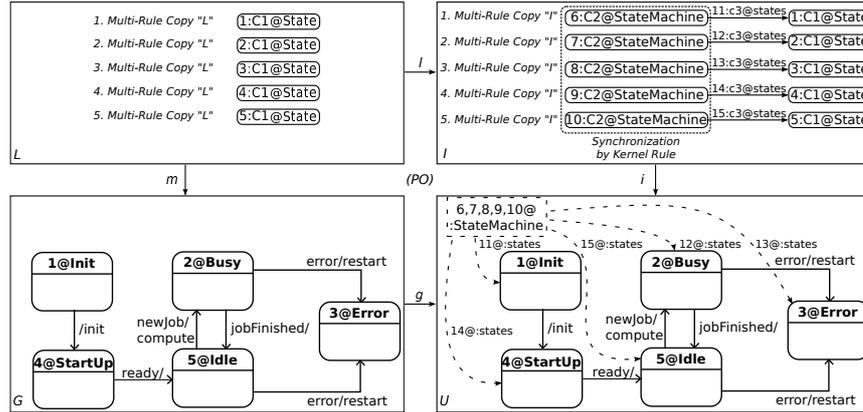


Figure 7.10: Migration step: “Introduce Container”

7.3 Default Migration Schemes

Having specified an evolution rule for meta-models, the automatic generation of model migration schemes shall be supported as far as possible. In this section, we introduce a strategy for generating default migration schemes from meta-model evolution rules. They may be customized to specific needs. The central question to be discussed is: given an evolution rule, which migration schemes shall be considered as the default? A trivial choice would be the *empty* migration scheme (i.e. $M \cup K = \emptyset$, see Definition 7.2.3). Empty migration schemes result in coupled transformation rules that retype, delete and split only. However, empty migration schemes are often not considered as adequate, see e.g. Example 7.2.1, where an attribute shall be moved. The empty migration scheme would not move but just delete the attribute value relations to objects. Fortunately, more adequate migration schemes can be deduced. Considering different examples, we observed that useful migration multi-rules are often identical to their evolution rule besides different typing. For evolution rule “Move Attribute,” for example, this is a good strategy. In this case, a meta-model evolution step is simply repeated on the model level as

often as possible. Hence, we would take a migration scheme containing the retyped evolution rule as multi-rule. The migration scheme for “Move Attribute” in Figure 7.4 contains two migration multi-rules. The first multi-rule is isomorphic to the evolution rule if we neglect the new container and containment relationship to be created. But the second one differs under the same assumption. It is needed to cover a migration case where the whole evolution pattern cannot be found on the model level. Rules for the second case can be generated by creating all possible connected subgraphs of the left-hand side of the evolution rule while keeping the intermediate graph as it is. This results in a set of multi-rules that can deal also with sub-patterns. If we keep only those multi-rules that replace model parts (i.e. add and implicitly delete, when completed by Step 3 in Procedure 6.4.1 on page 91), we get the migration scheme shown in Figure 7.4⁶.

In the following, we define a standard construction to automatically generate migration schemes from meta-model evolution rules for graphs using the strategy explained above. In many cases, this strategy provides useful migration schemes. In addition, generated migration schemes may be *customized* to fit to specific needs. To define default migration schemes, connected subgraphs are defined first:

Definition 7.3.1 (Connected subgraph). Given a graph H

- For any subgraph $G \sqsubseteq H$, $\equiv_G \subseteq G_V \times G_V$ denotes the equivalence relationship generated by the set $\{(src^G(e), trg^G(e)) \mid e \in G_E\}$.
- A subgraph $G \sqsubseteq H$ is called connected iff $\equiv_G = G_V \times G_V$

CONNECTED
SUBGRAPH

For category **Graph**, we construct default migration schemes as follows. We assume that a meta-model evolution rule has a left-hand-side that is a connected graph.⁷

Definition 7.3.2 (Default migration scheme). Given a non-deleting evolution rule $tp = tl: TL \rightarrow TI$, the default migration scheme $((\overline{K}, \overline{M}, \overline{KM}) \triangleright tp, \overline{prio})$ for rule tp is constructed by the following steps:

DEFAULT
MIGRATION
SCHEME

1. Construct a set of non-deleting multi-rules \overline{M} :

$$\begin{aligned} \overline{M} = & \{tl: TL \rightarrow TI\} \wedge \{l_i: L_i \rightarrow I_i \mid \\ & L_i \text{ is a connected subgraph of } TL (L_i \neq TL \text{ and } L_i \neq \emptyset), \\ & I_i = TI, l_i = tl|_{L_i}, (l_i \neq id \text{ (merging)}) \vee \\ & (\exists y \in I_i : \nexists x \in L_i \text{ with } l_i(x) = y \text{ (creating)}) \wedge \\ & (\exists w \in I_i : \exists u \in L_i \text{ with } l_i(u) = w \wedge \\ & \nexists v \in TR \text{ with } tr(v) = w \text{ (deleting)}) \} \end{aligned}$$

⁶if we neglect again the containment

⁷A connected graph is a connected subgraph of itself.

7. MODEL MIGRATION SCHEMES BASED ON COUPLED TRANSFORMATIONS

For each $l: L \rightarrow I \in \overline{M}$, the inclusions (denoted by \hookrightarrow) $in_L: L \hookrightarrow TL$ and $in_I: I \hookrightarrow TI$ provide the typing morphisms, where $l; in_I = in_L; tl$ is ensured by construction.

2. \overline{K} is the empty set of non-deleting kernel rules.
3. Consequently, \overline{KM} is also the empty set.
4. The default migration scheme defines a partial order on \overline{M} : the default rule priority is $\overline{prio}(l_i) > \overline{prio}(l_j)$ if $L_i \sqsupset L_j$ and $\overline{prio}(l_i) = \overline{prio}(l_j)$ if $L_i \not\supset L_j$ and $L_j \not\supset L_i$.

Rule priorities are used by the *matching strategy* to easily support distinct migration cases. Default migration schemes may be further customized:

- Multi-rules and kernel rules may be added, deleted or changed.
- The priority of multi-rules may be adapted.
- Application conditions may be added to rules (multi-rules and kernel rules), as specified in Chapter 4. Note that application conditions may be typed by either the left-hand side of the evolution rule or directly by the meta-model to be evolved. In the second case, the migration scheme becomes meta-model dependent (as in Example 8.5.1 in Chapter 8) but potentially more understandable.

The migration scheme in Example 7.2.1 is based on the default migration scheme of refactoring “Move Attribute”, but has been extended by a containment relationship and a new containment object. The one in Example 7.2.2 is also based on the default migration scheme but has been extended by a kernel rule synchronizing the addition of one default container.

In this chapter, migration schemes have been introduced based on an adaptation of amalgamated graph transformations. Migration schemes can be used to construct coupled transformation rules by migration patterns. In addition, a heuristic to deduce default migration schemes in category \mathbf{Graph} , given a meta-model evolution rule, has been presented.

Co-Evolution of Object-Oriented Models

A key feature of object-oriented modeling is class inheritance. It can be formalized by graphs with inheritance support. For example, \mathcal{J} -graphs as defined in [54] support inheritance between vertices (see Chapter 3). Together with corresponding morphisms, \mathcal{J} -graphs form a category, called \mathbf{IGraph} [54]. In this chapter, we consider coupled transformations on the basis of category \mathbf{IGraph} ¹. We identify extensions to our current framework that are desired in an application with \mathcal{J} -graphs and reconsider migration schemes. Finally, we discuss how migration schemes can be used to also formally define non-breaking, breaking and resolvable and breaking and unresolvable changes (see Chapter 2).

8.1 Introduction

Category \mathbf{IGraph} is a weak adhesive HLR category [54] for a class of \mathbf{M} -morphisms called *subtype-reflecting* (short \mathbf{M}_{S-refl}). A subtype-reflecting morphism is an injective graph morphism in which all sub-vertices of a vertex with a preimage also have preimages (see Example 8.1.1).

Definition 8.1.1 (Subtype-reflecting morphism). A subtype-reflecting morphism $f1 : GI0 \rightarrow GI1$ is a \mathcal{J} -graph morphism, where $f1$ is an injective graph morphism² and has the subtype reflection property: $\forall (v_{11}, v_1) \in I1^*, v_0 \in GI0_V : v_1 = f1_V(v_0) \implies \exists v_{01} \in GI0_V : f1_V(v_{01}) = v_{11} \wedge (v_{01}, v_0) \in I0^*$.

SUBTYPE-
REFLECTING
MORPHISM

¹This means meta-models are considered to be \mathcal{J} -graphs and models to be \mathcal{J} -graphs, which are directed multi-graphs as in Chapter 3.

²according to Definition 3.1.2 on page 30

($\text{IGraph}, \mathbf{M}_{S\text{-refl}}$) IS
WEAK ADHESIVE
HLR CATEGORY

Fact 8.1.1 (($\text{IGraph}, \mathbf{M}_{S\text{-refl}}$) is Weak Adhesive HLR Category). *The category IGraph of graphs with inheritance together with the class $\mathbf{M}_{S\text{-refl}}$ of subtype-reflecting morphisms is a weak adhesive HLR category.*

Example 8.1.1 (\mathcal{J} -graph morphism). Figure 8.1 and Figure 8.2 show sample \mathcal{J} -graph morphisms. \mathcal{J} -graph morphism are clan morphism between two \mathcal{J} -graphs and are e.g. used in meta-model evolution transformations. The meta-model presented by graph TG shows a hierarchy of classes modeling different types of cars: ECar (electrical car), TCar (traditional car) and HCar (hybrid car).

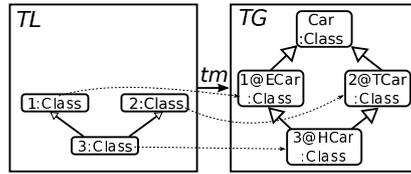


Figure 8.1: $\mathbf{M}_{S\text{-refl}}$ -morphism

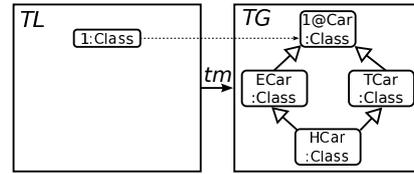


Figure 8.2: $\neg\mathbf{M}_{S\text{-refl}}$ -morphism

Figure 8.1 shows a subtype-reflecting \mathcal{J} -graph morphism, while the \mathcal{J} -graph-morphism in Figure 8.2 is not subtype-reflecting. Element mappings are again shown by dashed arrows. Inheritance edges in these figures are not mapped, as they are not “normal” graph edges. However, Definition 3.2.9 requires that all vertices that are in the same clan in the domain of the morphism are in its co-domain also in the same clan.

8.2 Supported Change Operations

Category IGraph is a weak adhesive HLR category with subtype-reflecting morphisms forming the class of \mathbf{M} -morphisms. This means that all constructions introduced in Chapter 6 and Chapter 7 can be directly applied.

To construct coupled transformations by Procedure 6.4.1, several morphisms need to be in \mathbf{M} . This means:

1. Morphism $tm : TL \rightarrow TG$ is subtype-reflecting³ and $m : L \rightarrow G$ is an injective graph morphisms.
2. Because the category is *weak* adhesive, morphism $l : L \rightarrow I$ also needs to be an injective graph morphism (see Theorem 6.4.1).
3. In [54], only a construction for pullbacks along subtype-reflecting morphisms is presented. Hence, morphism $tr : TR \rightarrow TI$ needs to

³Note that evolution rules that cannot be matched subtype reflective in a meta-model, may be automatically completed by a tool (by using 1:1 mappings).

be subtype-reflecting as well (and hence $r : R \rightarrow I$ an injective graph morphism), as typing morphisms cannot be assumed to be injective (see Step 3 in Procedure 6.4.1).

Item 3 restricts coupled transformations so that they are based on the cospan DPO approach. Splitting is not supported as long as no constructions for general pullbacks and FPBCs along subtype-reflecting morphisms have been identified. We can state immediately which evolution operations are safe to be applied:

1. Class vertices of type graphs may be *added*, *deleted* or *merged*⁴.
2. Object vertices of instance graphs may be *added* or *deleted*.

Note that subtype-reflection prevents evolution rules to delete vertices that are not on the top of an inheritance hierarchy. In addition, subtype-reflection ensures that pullbacks i.e. in particular the left-hand side L of a coupled transformation rules can be constructed component-wise based on category Set ⁵. Other types of operations, however, may also be desired:

1. Vertices and edges of instance graphs may be merged by e.g. an encoding of the “Inline Class” operation.
2. Vertices of instance graphs may be retyped to more special types, being sub-vertices of its current type vertex (called subtyping here).

In the next sections, we consider these two operations above and show under which conditions these operations are safe to apply. Note, in addition, a language engineer may want to retype objects to supertypes. This is supported the same way as subtyping, however, only if all incoming and outgoing edges that are directly typed by edges of the current type are explicitly deleted before with a “classical” graph transformation on the model level. Without this preceding operation, the typing morphism $t_U : U \rightarrow TU$ in the corresponding coupled transformation would violate the definition of a clan morphism (see Definition 3.2.8 on page 42). We neglect this operation in the following.

8.3 Merging of Model Elements

Merging of model elements is not supported in category IGraph , because the category is only a *weak* adhesive HLR category (see Theorem 6.4.1). However, category Graph is an adhesive category and merging of model

⁴This applies to sub- and super-vertices as well.

⁵analog to the construction of pullbacks in category Graph

elements is supported in this category. Fortunately, in restricted cases, \mathcal{J} -graphs can be considered as objects in category \mathbf{Graph} by help of a flattening construction. For such cases merging of model elements can be allowed. In this section, sufficient conditions for such a restriction are examined.

CLOSURE OR
FLATTENING OF
 \mathcal{J} -GRAPH

Definition 8.3.1 (Closure or Flattening of \mathcal{J} -graph [54]). Given \mathcal{J} -graph $GI = (G, I)$, then the closure \overline{GI} is a graph $\overline{GI} = (\overline{GI}_V, \overline{GI}_E, src^{\overline{GI}}, trg^{\overline{GI}})$ with $\overline{GI}_V = G_V, \overline{GI}_E = \{(v_1, e, v_2) \in G_V \times G_E \times G_V \mid v_1 \in \text{clan}_I(src^G(e)), v_2 \in \text{clan}_I(trg^G(e))\}, src^{\overline{GI}}(v_1, e, v_2) = v_1, trg^{\overline{GI}}(v_1, e, v_2) = v_2$. The closure \overline{GI} is also called flattening of GI .

By the construction above, \mathcal{J} -graphs can be reduced uniquely to graphs in category \mathbf{Graph} and \mathcal{J} -graph morphisms correspondingly into graph morphisms [54]. The construction makes the inherited edges explicit. However, even the functor leading from \mathbf{IGraph} to \mathbf{Graph} is cofree [54], i.e. preserve pullbacks; it unfortunately does not preserve pushouts (see Example 8.3.1).

Example 8.3.1 (Pushouts in \mathbf{IGraph} versus pushouts in \mathbf{Graph} ⁶). Figure 8.3 shows two pushout examples. A pushout of $TG \xleftarrow{tm} TL \xrightarrow{ti} TI$ in cate-

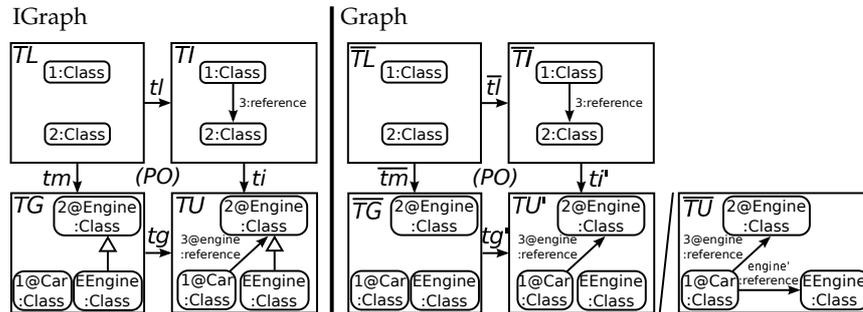


Figure 8.3: Pushouts in \mathbf{IGraph} versus pushouts in \mathbf{Graph}

gory \mathbf{IGraph} is shown at left of Figure 8.3, while in its right a pushout in category \mathbf{Graph} of the flattened span $\overline{TG} \xleftarrow{\overline{tm}} \overline{TL} \xrightarrow{\overline{ti}} \overline{TI}$ (see Definition 8.3.1) is shown. Both pushouts simply add an edge formalizing a reference between two vertices formalizing classes. However, both pushouts are not equivalent, as flattening the pushout graph TU of the first pushout (see graph \overline{TU} in the right of the figure) creates an additional edge 'engine' that is not in graph TU' .

⁶adopted from [54]

Fact 8.3.1 (Pushouts in category $\mathbf{IGraphs}$ along \mathbf{M}_{S-refl} morphisms[54]).

Given a subtype-reflecting morphism $f1 : GI0 \rightarrow GI1$ and a general \mathcal{J} -graph morphism $f2 : GI0 \rightarrow GI2$, then the pushout (1) in category \mathbf{IGraph} exists and can be constructed component-wise^a for the vertex and edge components with $I3 = (g1_V \times g1_E)(I1) \cup (g2_V \times g2_E)(I2)$. Moreover, $g2 : GI2 \rightarrow GI3$ becomes a subtype-reflecting morphism.

$$\begin{array}{ccc} GI0 & \xrightarrow{f1} & GI1 \\ f2 \downarrow & (1) & \downarrow g1 \\ GI2 & \xrightarrow{g2} & GI3 \end{array}$$

PUSHOUTS IN
CATEGORY $\mathbf{IGraphs}$
ALONG \mathbf{M}_{S-refl}
MORPHISMS

^aComponent-wise means component-wise in category \mathbf{Set} analog to the construction of pushouts in (directed multi-)graphs.

The problem presented in Example 8.3.1 occurs if new edges are created having a super vertex as source or target, then the corresponding inherited edges in sub-vertices are not created in the flattened pushout. Obviously, this problem does not occur in cases where new edges are only created at leaf vertices of inheritance hierarchies. In a coupled transformation, this is ensured in cases where evolution rules are not using graphs with inheritance. Note that in Example 8.3.1, morphism $tm : TL \rightarrow TG$ is not subtype-reflecting and therefore TL does not specify any inheritance relationship. In a coupled transformation, this would not be allowed.

WHY ARE PUSHOUTS
IN \mathbf{IGraph} AND
 \mathbf{Graph} NOT
EQUIVALENT

A second problem occurs if vertices with a common super vertex are merged: constructing the pushout in category \mathbf{Graph} results in a pushout object with duplicated edges due to multiple inheritance. If the pushout is constructed in category \mathbf{IGraph} and flattened, the pushout object does not contain such duplications. To avoid both problems, we restrict pushouts:

Proposition 8.3.1 (Equivalence of pushouts in \mathbf{Graph} and \mathbf{IGraph}).

Given (see Figure 6.3 on page 78)

- an \mathcal{J} -graph morphism $tl : TL \rightarrow TI = (TL', \emptyset) \rightarrow (TI', \emptyset)$
- and a subtype-reflecting morphism $tm : TL \rightarrow TG = (TL', \emptyset) \rightarrow (TG', I_{TG})$ restricted by: if $\exists v_1, v_2 \in TL \implies \nexists v_3 \in TG$ with $tm(v_1) \in \text{clan}(v_3) \wedge tm(v_2) \in \text{clan}(v_3)$

EQUIVALENCE OF
PUSHOUTS IN
 \mathbf{Graph} AND
 \mathbf{IGraph}
(RESTRICTED CASE)

then a pushout of $TG \xleftarrow{tm} TL \xrightarrow{tl} TI$ in category \mathbf{IGraph} corresponds to its flattened pushout in category \mathbf{Graph} . This means that a pushout in category \mathbf{IGraph} is after flattening a pushout in category \mathbf{Graph} . Furthermore, this means that a pushout constructed in category \mathbf{IGraph} and flattened is isomorphic to the pushout in category \mathbf{Graph} of its flattened span.

Proof. Both \mathcal{J} -graph morphisms in Proposition 8.3.1 are already graph morphisms by assumption (see Definition 8.1.1).

1. Since morphism tm is subtype-reflecting, tm does not match any super vertex: if it would match a super vertex, then it would also need to match all corresponding sub-vertices i.e. the set of inheritance

relationships of TL would not be empty. This means “the rule application” i.e. the pushout does not create any additional edges in TU compared to TG that have a super vertex as source or target. Hence, also flattening the pushout graph TU does not create any additional implicit inherited edges compared to flattening graph TG .

2. In addition, the flattening of TU does not create less implicit inherited edges compared to graph TU' constructed by the pushout of the flattened span. This is ensured by the restriction that merging is restricted to leaf vertices that do not share any common super vertex.

□

Restricting the first pushout of the meta-model evolution rule is enough to transfer the construction of Procedure 6.4.1 to category **Graph**. The proof of Procedure 6.4.1 goes through: the left face of the coupled transformation is also a pullback in category **Graph**, as the construction of pullbacks in **IGraph** (see Fact 8.3.2) is identical to the construction of pullbacks in **Graph** for the case TL or G is a (directed multi-)graph (i.e. an \mathcal{J} -graph without inheritance relationships). The left bottom face is already a pushout in category **Graph** by assumption. Hence, the left cube (see Figure 6.3 on page 78) of the coupled transformation can be constructed in category **Graph** instead. Furthermore, it can be deduced that the middle face is a pullback in **Graph** that also corresponds to a pullback in **IGraph** (since also TI and U are graphs). This means the right cube of the coupled transformation can be constructed in category **IGraph**.

PULLBACKS IN
CATEGORY **IGraphs**
ALONG \mathbf{M}_{S-refl}
MORPHISMS

Fact 8.3.2 (Pullbacks in category **IGraphs** along \mathbf{M}_{S-refl} morphisms [54]).

Given a subtype-reflecting morphism $g2 : GI2 \rightarrow GI3$ and a general \mathcal{J} -graph morphism $g1 : GI1 \rightarrow GI2$, then the pullback (1) in category **IGraph** exists and can be constructed component-wise for the vertex and edge components, with $I0^*$ defined by $(v_0, v'_0) \in I0 \iff (f1_V(v_0), f1_V(v'_0)) \in I1^*$ and $(f2_V(v_0), f2_V(v'_0)) \in I2^*$. Moreover, $f1 : GI0 \rightarrow GI1$ becomes an subtype-reflecting morphism.

$$\begin{array}{ccc} GI0 & \xrightarrow{f1} & GI1 \\ f2 \downarrow & (1) & \downarrow g1 \\ GI2 & \xrightarrow{g2} & GI3 \end{array}$$

RELAX
CONDITION 2 OF
PROPOSITION 8.3.1

Reconsidering Proposition 8.3.1, we realize that in an implementation, the second condition can be relaxed so that match morphisms $tm : TL \rightarrow TG$ of rules only need to be subtype-reflecting. The condition that vertices to be merged do not share a super vertex ensures that inherited edges are considered correspondingly in **IGraph** and **Graph**. However, we also get equivalent pushouts if we assume that all inherited edges get merged, too (see Example 8.3.2). In an implementation, rules could be automatically completed to also merge such edges⁷. Similar rules can also be extended to be subtype-reflecting if necessary.

⁷Note, in practice such a completion can often be omitted.

Example 8.3.2 (Extending pushouts for merging vertices). Figure 8.3.2 shows pushouts merging two classes in category Graph respectively IGraph. While the left of Figure 8.3.2 shows pushouts in IGraph, its right shows pushouts of correspondingly flattened spans. Even if the two merged classes do not have a common superclass, the second condition in Proposition 8.3.1 is violated because their clans share a common class due to the inheritance relationship. In Figure 8.3.2 (a), the result is that the pushout objects (i.e. TU, \overline{TU}) in Graph and IGraph do not correspond. In Figure 8.3.2 (b), the spans have been changed so that *all* context edges are also matched by morphism $tm : TL \rightarrow TG$, such that the result is that the pushout objects correspond.

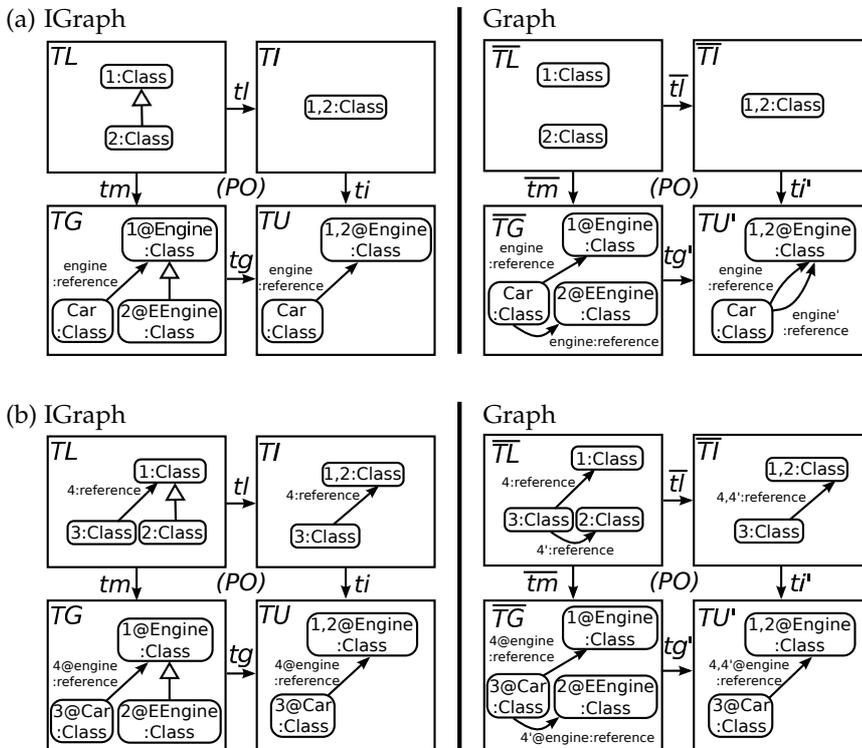


Figure 8.4: Pushouts in IGraph versus pushouts in Graph

8.4 Retyping Model Elements to Subtypes

In object-oriented modeling, classes may be refined and objects may be retyped to subclasses. Such operations should be additionally supported by coupled transformations in category IGraph.

The definition of a coupled transformation demands that all faces of the double cube in Figure 6.3 commute. If we allow subtyping, i.e. retype existing instances to subtypes of their original types, at least the left back face and the left front face of a coupled transformation are no longer commuting. Therefore, we want to relax the commuting condition for the left back and the left front face to weakly commutativity:

WEAKLY
COMMUTING
CLAN-MORPHISM

Definition 8.4.1 (Weakly commuting clan-morphism). Given two clan morphisms $f, g : G \rightarrow H$, f is called *finer* than g , written $f \leq g$, if $\forall x \in G$ $f(x) = g(x)$ or $f(x)$ is in the clan of $g(x)$. Furthermore, we call morphisms f and g *weakly commuting*, as they are commuting if we change the vertex mapping in f from a subtype to a corresponding supertype mapped in g .

Example 8.4.1 shows a coupled transformation rule with a weakly commuting left face.

Example 8.4.1 (Subtyping coupled transformation rule (in category IGraph)). Figure 8.5 shows a coupled transformation rule with a weakly commuting left face and a commuting right face. The (meta-model) evolution rule refines a class C1 by two subclasses C2 and C3 before class C1 is deleted. The migration rule retypes one instance of class C1 to its subtype C3. Element mappings are again denoted by dashed arrows.

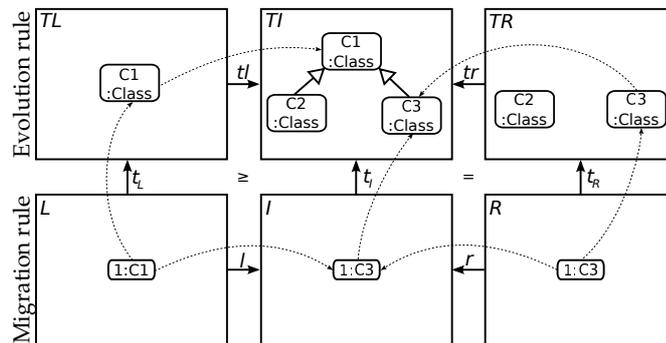


Figure 8.5: Coupled transformation rule with a weakly commuting left face

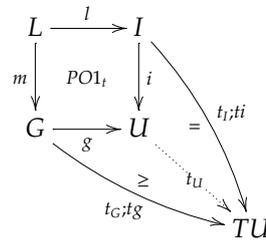
Retyping to a subtype affects only the left cube of a coupled transformation (see Figure 6.6). Therefore, we only need to reconsider the left cube construction. In particular, we want to construct the left cube so that *only* the back and front faces are weakly commuting. *The right face of the left cube shall still be commuting.* This ensures that a coupled transformation rule and its coupled transformation result retypes correspondingly.

In the following we show that Procedure 6.4.1 can be so adapted that it can be applied in category IGraph allowing retyping.

Proposition 8.4.1. [Procedure 6.4.1 allows subtyping in category \mathbf{IGraph}] By adapting Step 2 and Step 4 of Procedure 6.4.1, the procedure can also be applied in category \mathbf{IGraph} to construct coupled transformations with a weakly commuting left back and left front faces:

PROCEDURE 6.4.1
ALLOWS SUBTYPING
IN
CATEGORY \mathbf{IGraph}

- In Step 2, we allow to complete the left face of the coupled transformation rule to a weak commuting diagram (i.e. $t_L;tl \geq l;t_I$).
- In Step 4, the existence of a unique morphism $t_U : U \rightarrow TU$ with $t_I;ti = i;t_U$ and $t_G;tg \geq g;t_U$ can still be deduced by the pushout in the left bottom face of a coupled transformation, although the outer square of the diagram below is only weakly commuting.



That coupled transformations still have commuting middle faces has a desired side effect: it ensures that the right face of the left cube is still a pullback, which has been the precondition for constructing the right cube. The reason for this is that only properties of the left, top and bottom faces of the left cube have been used to show that its right face is pullback (see proof of Procedure 6.4.1). Hence, the right cube of a coupled transformation does not need to be considered in the proof of Proposition 8.4.1. This reduces the task to prove Step 4 of Procedure 6.4.1 i.e. that morphism $t_U : U \rightarrow TU$ exists with $t_I;ti = i;t_U$ and $t_G;tg \geq g;t_U$ also under the relaxed conditions.

Proof of Proposition 8.4.1. The proof primarily relies on the fact that $ti : TI \rightarrow TU$ is subtype-reflecting (Property 6), i.e. in particular an injective graph morphism (see Definition 8.1.1). It has to be shown that a morphism t_U exists: the left back face of a coupled transformation is weakly commuting by assumption. We assume $t'_I : I \rightarrow TI$ to be the corresponding morphism that makes the left back face commuting. Assuming a morphism t'_I in the pushout diagram of Proposition 8.4.1 implies the existence of a unique morphism $t'_U : U \rightarrow TU$. Because ti is an injective graph morphism and the fact that two vertices that are in an inheritance relationship in the domain of a morphism are also in an inheritance relationship in its co-domain (see Definition 3.2.9), we can adapt morphism t'_I and morphism t'_U correspondingly to morphism t_I , respectively t_U , so that $t_I;ti = i;t_U$ and $t_G;tg \geq g;t_U$ hold. \square

8.5 Model Migration Schemes

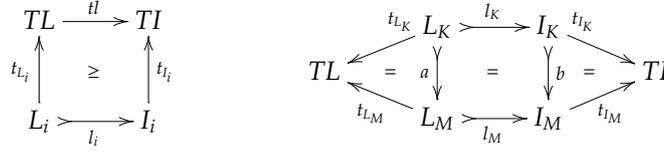
In the last section, it has been shown that coupled transformations can be used to retype to subtypes. In this section, it is shown that such coupled transformations can also be constructed by adapting Procedure 7.2.1. In particular, we want to consider two new features:

1. We intend to match multi-rules with more flexibility: a vertex in L typed by supertype of TL shall also match a vertex of a corresponding subtype, as this is the expected semantics of inheritance. A rule with more general types can also be used to transform graphs with more specific types.
2. Rules of a migration scheme shall allow retyping vertices to a subtype of their current type.

Examining Procedure 7.2.1, we realize that both new features only require to adapt Step 8, i.e. the construction of morphism $t_l : I \rightarrow TI$. Note that since models are formalized by (directed multi-) and only meta-models by \mathcal{J} -graphs, all other steps are well-defined. The *first new feature* implies that vertices matched to vertices typed by a subtype cannot rely on the types defined in right-hand sides of multi- and kernel rules. They have to be adapted correspondingly. The *second new feature* implies that retyping conflicts can occur. Multi-rule matches may overlap and multi-rules may try to retype a vertex to different subtypes. This conflict has to be avoided or resolved adequately. Furthermore, it has to be avoided that both new features interfere. This may cause unresolvable conflicts. It also has to be avoided that multi-rules which subtype match the corresponding vertices to vertices typed by a different subtype. In the following, the definition of migration schemes is adapted to allow subtyping. Afterward, Step 8 of Procedure 7.2.1 is replaced by a element-wise construction. That the adaption of Procedure 7.2.1 is well-defined is shown thereafter.

MIGRATION
SCHEME

Definition 8.5.1 (Migration scheme in category \mathbf{IGraph}). A *migration scheme* is a tuple $((K, M, KM) \triangleright tp, prio)$ consisting of an interaction scheme (K, M, KM) of non-deleting rules typed over an evolution rule tp and an priority function $prio : M \rightarrow \mathbb{N}$ for rules in M . Each rule $l_i \in M \cup K$, l_i is well-typed by the non-deleting part $tl : TL \rightarrow TR$ of rule tp , i.e. $tp: t_{L_i}; tl \geq l_i; t_{I_i}$ and *injective*. In addition, all rules in $M \cup K$ only have left and right-hand sides, which are (directed multi-)graphs (i.e an \mathcal{J} -graph without inheritance relationships). Furthermore, each kernel morphism $(a, b) \in KM$ between kernel rule $l_K : L_K \rightarrow I_K$ and multi-rule $l_M : L_M \rightarrow I_M$ is type compatible i.e. $a; t_{L_M} = t_{L_K}$ and $b; t_{I_M} = t_{I_K}$.



1. For each pair of rules l_1 and l_2 in M , we may have zero or finitely many triples of a kernel rule (l_k, k_1, k_2) with $l_k : L_k \rightarrow I_k$ in K and kernel morphisms $k_1 : l_k \rightarrow l_1$ and $k_2 : l_k \rightarrow l_2$ in KM .
2. Furthermore, for each subset of rules $R \subseteq M$ with (1) possibly overlapping matches (2) that retype a vertex typed by vertex s to a sub-vertex v_i of $ti(s)$ then either $v_i = v_1$ or there exists exactly one vertex $v_3 \in TI$, which is a sub-vertex of all v_i .

In [89], category \mathbf{G}^\top is presented which supports subtyping. Subtyping conflicts can be generally resolved in this category, as *all* types in type graphs have to form a lattice (i.e. there is always a unique minimal common subtype in conflicting cases). In \mathbf{IGraph} , we can avoid conflicts by restricting rules of migration schemes and the matching strategy.

Procedure 8.5.1 (Rule amalgamation for migration schemes in \mathbf{IGraph}). Given a migration scheme $((K, M, KM) \triangleright tp, prio)$ (see Definition 8.5.1), an evolution step $tt : TG \xrightarrow{tp, mm} TH$ and a graph G with typing $t_G : G \rightarrow TG$: the non-deleting part of the migration rule $l : L \rightarrow I$ for G can be constructed by the following amalgamation procedure (see also Figure 7.1 on page 104 and Figure 7.7 on page 110):

RULE
AMALGAMATION
FOR MIGRATION
SCHEMES IN
 \mathbf{IGraph}

1. Construct a pullback $G \xleftarrow{m} L \xrightarrow{t_L} TL$ of $G \xrightarrow{t_G} TG \xleftarrow{tm} TL$ (as before).
2. Match rules of M to L along a chosen matching strategy as before. Allow to match vertices typed by a supertype to corresponding subtypes if such vertices are not retyped to a subtype by the rule. The default strategy matches rules as before. If no match has been found, stop the procedure here with $l = id$ and $t_I = t_L$; tl constructing trivially a commuting diagram. Otherwise, continue with Step 3.
- 3.-7. See Procedure 7.2.1 on page 108.
8. Morphism $t_I : I \rightarrow TI$ is constructed element-wise, assume $t_{I_i} : I \rightarrow TI$ to be the typing morphisms of multi-rule copy i with $(0 \leq i \leq n)$:
 1. Map each vertex v in I to the most specific type of $V(v)$ (see below) if such a type exists. If such a type does not exist then map it to the most specific subtype of all types in $V(v)$.

$$V(v) = \{u \in TI_V \mid u = ti(t_L(w)) \text{ with } w \in \text{pre-image}(v) \text{ or } u = t_{I_i}(v)\}$$

2. Map each edge e in I to the only edge in $E(e)$.

$$E(e) = \{u \in TI_E \mid u = ti(t_L(w)) \text{ with } w \in \text{pre-image}(e) \text{ or } u = t_i(e)\}$$

PROCEDURE 8.5.1
IS WELL-DEFINED

Proposition 8.5.1 (Procedure 8.5.1 is well-defined). *In category IGraph: given a migration scheme $((K, M, KM) \triangleright tp, prio)$ (according to Definition 8.5.1) an evolution step $tt : TG \xrightarrow{tp, tm} TH$ and a morphism $t_G : G \rightarrow TG$, the left face of the coupled transformation rule for G can be constructed by Procedure 8.5.1 so that:*

1. *the coupled transformation rule is match-complete and*
2. *the left face of the coupled transformation rule is (weakly) commuting i.e. $t_L; tl \geq l; t_l$.*

Proof.

Only Step 8 has to be proven as all other steps are still valid (see Proof of Procedure 7.2.1).

1. Obviously, it is ensured that a unique vertex mapping exists so that the left face of a coupled transformation rule is weakly commuting (for vertices):
 - If an existing vertex $v \in I$ should not be retyped to a subtype, a unique type must exist as before. Vertex v can be typed by $l^{-1}(v); t_L; tl$. Note that $l : L \rightarrow I$ is an injective graph morphism.⁸
 - If an existing vertex should be retyped to a subtype, a unique most specific type must exist as required in Condition 2 of Definition 8.5.1.
 - A new vertex v can simply be typed by $t_i(v)$.
2. Analogously, it is ensured that each edge $e \in I$ can be uniquely typed by $l^{-1}(e); t_L; tl$. A new edge e is simply typed by $t_i(e)$.

That the construction ensures that morphism t_l is a valid clan-morphism is obvious, as vertices are only mapped to more specific types that “inherit” all edges from their supertypes.

Additionally, the construction trivially ensures that the left back face of the coupled transformation rule is (weakly) commuting. \square

Example 8.5.1 shows a migration scheme containing two multi rules that specify a conflicting subtyping. By the fact that both subtypes have a common subtype themselves, the conflict is solved.

⁸ Because IGraph is weak adhesive. Merging of model elements can be done in category Graph instead.

Example 8.5.1 (Subtyping in category IGraph). Figure 8.6 shows a meta-model evolution step with a corresponding model migration step formalized by \mathcal{J} -graphs. Graph TG formalizes a simple meta-model allowing the modeling of cars with different engines, in particular electrical ones. In graph TH , class Car is refined by three subclasses: $TCar$ (traditional car), $ECar$ (electrical car) and $HCar$ (hybrid car) (as in Example 8.1.1). Graph G contains three Car instances having electrical or traditional gas engines. In graph H , each car is retyped according to its type of engine, i.e., to a $TCar$, a $ECar$ or an $HCar$.

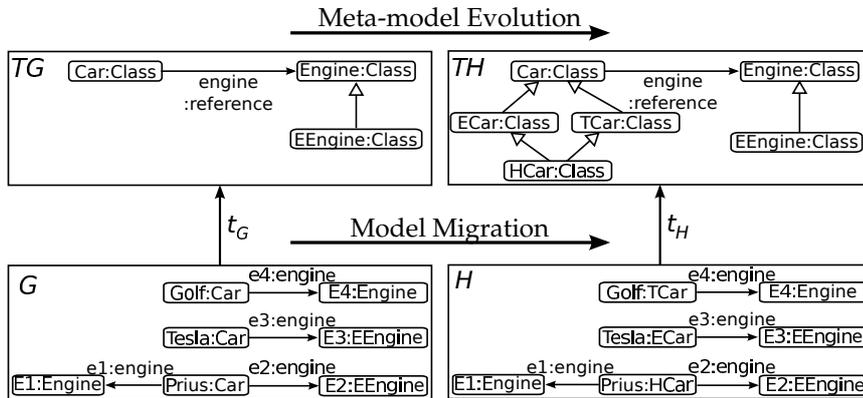


Figure 8.6: Meta-model evolution and model migration with subtyping

Figure 8.7 shows the meta-model evolution rule and the migration scheme used to build the corresponding coupled transformation.

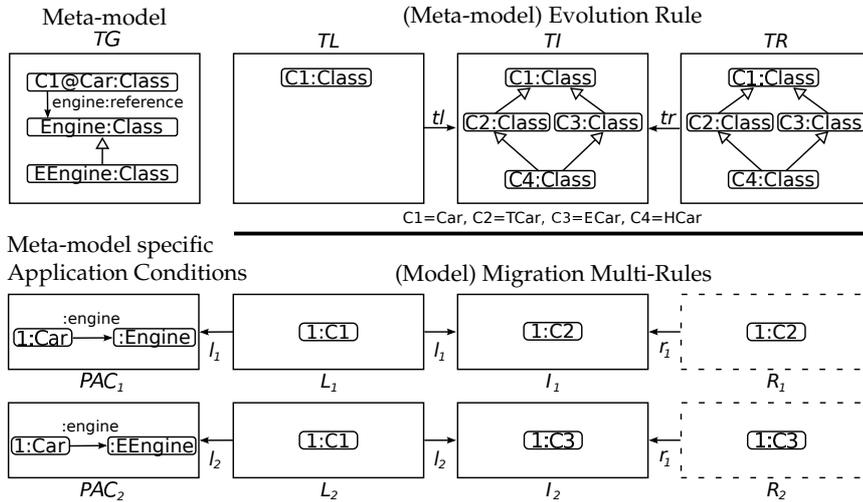


Figure 8.7: Migration scheme with subtyping

The migration scheme consists of only two multi-rules that retype to subtypes. Both rules are extended by custom positive application conditions (see Chapter 4) being directly typed over the meta-model TG in Figure 8.6. Note that subtyping multi-rules are only defined for traditional and electrical cars. If the matches of both rules overlap in an instance of type `Car` (as it is the case for `Prius:Car` in model G in Figure 8.6), the `Car` instance gets the common subtype, i.e. type `HCar`.

In Chapter 7, a construction for default migration schemes for (directed multi-)graphs is given (see Definition 7.3.2). A set of default multi-rules is deduced by retyping the meta-model evolution rule and its sub-rules. In category `lGraph`, however, inheritance relationships do not have instances, hence a simple retyping of rules is not possible. Therefore, we propose to use the empty migration scheme as a default migration scheme in cases when evolution rules use inheritance. In such cases, it is hard to propose a simple and useful solution. Hence, we can primarily reuse the construction for default migration schemes from category `Graph` in category `lGraph`. If rules merge vertices, we only keep those generated rules that merge leaving vertices (see Proposition 8.3.1 and Example 8.3.2).

8.6 Classification of Meta-model Changes Revisited

An often used classification for meta-model changes (e.g. in [22, 59, 97, 141]) proposes the distinction into non-breaking changes, breaking and resolvable changes and breaking and unresolvable changes [51] (see Chapter 2). Unfortunately, the classification is partly *context-dependent* and leaves room for interpretation. The consequence is that one and the same meta-model change may be classified differently by several researchers. In his dissertation [59], Herrmannsdörfer, e.g., reclassifies meta-model evolution changes in EMF according to an own model presentation. Hence, such classifications may be reasonable if specific modeling frameworks should be evaluated, but also restrict the validity of research results to tools. In this section, the existing classifications are revisited and formalized wrt. the developed theory of this thesis.

As illustrating example (see Example 8.6.1), we consider a sequence of meta-model evolution steps and discuss corresponding migration schemes.

Example 8.6.1 (Evolution of a statemachine meta-model⁹). This example builds on Example 7.2.1 and shows the evolution of a statemachine meta-model in several evolution steps using more realistic meta-models. Figure 8.8 shows an extended meta-model for Moore automata. Multiplicity annotations can be formalized e.g. by meta-attributes, associations that

⁹The example is based on an example that has been originally presented by Herrmannsdörfer et al. to illustrate the Cope [55, 57] tool.

are navigable in both directions by two opposite edges similar to EMF [34]. In the following, the evolution scenario performed on this meta-model is described in detail:

1. In the first step (see Figures 8.8 and 8.9), the modeling language is changed from Moore to Mealy automata. This means that we *move the attribute effect* from State to Transition. This evolution step has been shown in Example 7.2.1 before. To migrate models, effects (i.e. values of attribute `effect`) have to be moved from each state to all of its incoming transitions. The models can be migrated by the migration scheme presented in Example 7.2.1.

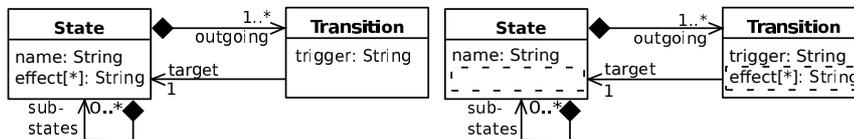


Figure 8.8: Original meta-model

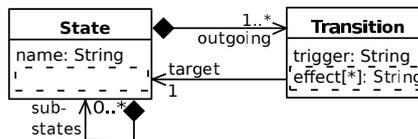


Figure 8.9: Move Attribute

2. In the second step, we *add opposite references* to outgoing and target edges (see Figures 8.9 and 8.10). This meta-model change may ease the usage of generated artifacts. In addition, after this evolution step, models can be migrated by the default migration scheme adding opposite links.

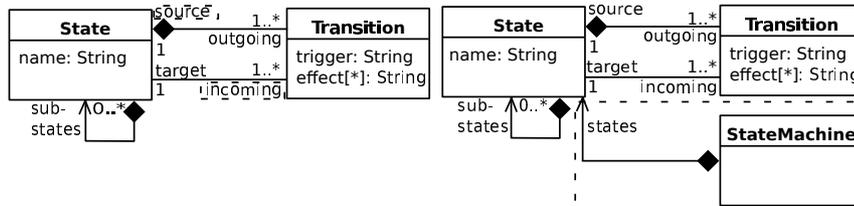


Figure 8.10: Add Opposite Ends

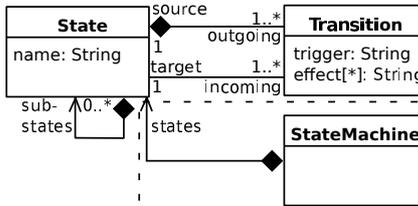


Figure 8.11: Introduce Container

3. In the third step, we *introduce a container class StateMachine* as it is required in EMF [34] meta-models (see Figure 8.10 and Figure 8.11). Such a container class allows a unique root object that may be helpful to serialize models in XML files. In EMF, it would be required that all states not already contained in another state are contained in the root container `StateMachine`. Models can be migrated by the

the construction of Procedure 7.2.1. A change where the empty migration scheme is sufficient is e.g. the “Pull up Attribute” refactoring in category IGraph. In this meta-model refactoring, attribute references are non-injectively mapped to a new attribute reference of a superclass as allowed in J-graph morphisms. Models can be migrated by retyping only. Figure 8.14 shows a sample coupled transformation rule pulling up an attribute and retyping one link correspondingly.

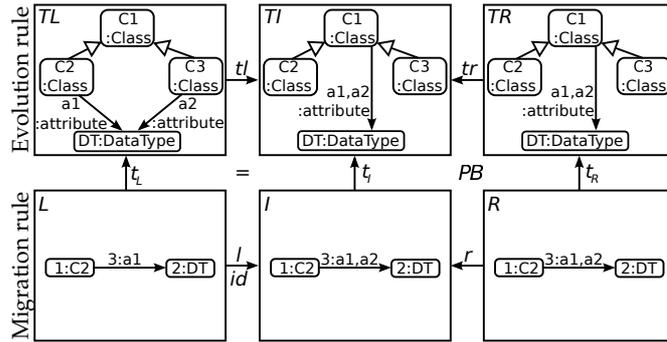


Figure 8.14: Coupled transformation rule “Pull up Attribute”

Based on the theory developed in this article, we can formally classify meta-model evolution steps exactly corresponding to Gruschko’s classification [51]. Similar to Brand et al. [141], we refined unresolvable changes: changes being resolved by manually defined or customized migration schemes are called *semi-automatically* resolvable. In addition, *fully automatically* resolvable and *automatically* resolvable changes are distinguished. We distinguish these two types of changes, as the second type is dependent on the heuristics used to define default migration schemes.

Definition 8.6.1 (Formal classification of meta-model evolution steps).

Given a meta-model evolution step $tt : TG \xrightarrow{tp, tm} TH$ at evolution rule $tp = TL \xrightarrow{tl} TI \xleftarrow{tr} TR$:

- Evolution step tt is called *non-breaking* if tl is injective and $tr = id$. Hence, instance graphs and their typing morphisms do not need to be migrated (if no additional constraints are added e.g. by DPF atomic constraints).
- Otherwise, evolution step tt is called *breaking*.
 - If models can be migrated by the *empty* migration scheme, the construction in Procedure 7.2.1 yields $l = id_L : L \rightarrow L$ as the

left part of the migration rule and tt is called *fully automatically* resolvable.

- If models can be migrated by the *default* migration scheme, tt is called *automatically* resolvable.
- If models can be migrated by a *customized or manually defined* migration scheme, tt is called *semi-automatically* resolvable.
- Otherwise, tt is called *unresolvable*.

Example 8.6.2 (Classification of evolution steps). Evolutions steps presented in Example 8.6.1 can now be classified according to Definition 8.6.1: Change 1 is semi-automatically resolvable, Change 2 is automatically resolvable, Change 3 is semi-automatically resolvable, Change 4 (a) can either be classified as non-breaking or semi-automatically resolvable depending on the desired migration, Change 4 (b) is semi-automatically resolvable and the “Pull Up Attribute” refactoring presented below is fully automatically resolvable.

Remark 8.6.1 (Unresolvable meta-model changes). While Definition 8.6.1 corresponds mainly to how other researchers classify unresolvable meta-model changes, we have to admit that, strictly speaking, many meta-model evolution steps can be automatically resolved to some extent. Newly required elements can be introduced with an automatically generated unique identifier, default name or further default values. Hence, an unresolvable i.e. *non-automatically* resolvable meta-model evolution step seems to be one that requires solving a constraint satisfaction problem without any resolution or without a unique resolution. This means that a modeler has to find a model-specific resolution.

In this chapter, coupled transformations and migration schemes have been considered in the context of category IGraph. Additional features such as (1) merging of elements on the model level, (2) subtyping as well as (3) matching of rules by supertypes, that are not supported by the core theory in weak adhesive categories, have been discussed and proven. Furthermore, the well-known classification of changes into unbreaking, breaking and (un)resolvable has been examined in the context of the introduced theory.

Towards Model Migration Ensuring Constraint Satisfaction

In Chapter 3, we discussed possibilities of equipping models by constraint annotations, e.g. by DPF atomic constraints. Trivially, such model constraints can be checked after model migration. However, ensuring that all well-formedness constraints are satisfied in models after model migration has been neglected so far. In this chapter, we present a first approach towards model migration that ensures constraint satisfaction after migration. Herein, rule schemes, called *model constraint resolution schemes*, are used to resolve constraint violations after models have been migrated by coupled transformations. In this first approach, we restrict ourselves to multiplicity constraints. The chapter is based on [137].

9.1 Introduction

Migrating models so that all modeling language constraints are ensured after model migration is rarely studied. Existing model migration approaches (see Chapter 2) use the following strategies to deal with constraint satisfaction:

- In manual specification approaches, it is up to the migration designer that model migrations ensure constraint satisfaction (e.g. by giving preconditions for migration cases). Only for simple evolution steps e.g. if meta-model constraints are removed or weakened,¹ can models be migrated safely. In other cases, models need to be verified after model migration and manually adapted in conflicting cases.

STATE-OF-THE-ART

¹For example a multiplicity constraint can be weakened by increasing its upper bound.

- In operator-based approaches (and matching approaches), the operator developer is responsible for models' satisfaction of modeling language constraints after migration. Therefore, meta-model evolution operators are usually restricted to cases where corresponding model migrations can be applied safely. Preconditions are defined by the operator developer and automatically checked before the evolution operator can be applied. If evolution operators are given by graph transformation rules, e.g. application conditions (see Chapter 4) can be used to specify such preconditions.²
- There are meta-model evolution steps that are generally considered unresolvable (e.g. an upper bound multiplicity constraint is decreased [22]). In such cases, models need to be migrated manually.

In the following, multiplicity constraint satisfaction is considered *over the whole model*. This means we solve constraint satisfaction problems (CSP) [46] and do not only consider the resolution of local constraint violations. Resolving a local constraint violation may imply that other constraints become violated and new violations need to be resolved.

Remark 9.1.1 (Local constraint violations). Note that local constraint violations can already (partly) be resolved by the presented theory. The left- and right-hand sides of graph transformations rules can be considered as *pre-* and *post-*conditions, respectively. Further application conditions may be added. This means that multi-rules can be specified, which ensure the satisfaction of lower bound multiplicity constraints by creating the required number of elements.

9.2 Resolution Procedure

After models are migrated by coupled transformations, all models are well-typed over the latest meta-model version. However, multiplicity constraints may still be violated. Such violations are resolved by applying a set of conflict-resolving graph transformation rules until global model consistency is established. This is possible [12, 18, 83] if the given meta-model is finitely satisfiable (i.e. if well-formed instance models do exist).

Figure 9.1 shows an activity diagram³ of the conflict resolution procedure introduced next. Conflicts are resolved by the following activities:

1. First, we check if a meta-model has finite model instances by a reasoning technique for UML class diagrams [12, 18]. This technique deduces a linear system of inequalities from all multiplicities.⁴ If a

²Note that such preconditions may also be defined by other approaches, e.g. OCL [105].

³Note that the activity diagram refines one activity of the diagram in Figure 1.3 on page 7.

⁴We restrict attention to binary associations here.

meta-model does not have finite model instances, we consider it as incorrect. In this case, it has to be (manually) adapted.

2. Afterward, meta-model specific graph transformation rules for constraint violation resolution are derived based on rule templates (defined in a model constraint resolution scheme).
3. Because derived resolution rules do not always adapt models meaningfully, custom resolution rules can be added to the rule set. In addition, priorities of generated rules may be changed.
4. First, violations are resolved by applying custom rules as often as possible.
5. Then, all upper bound multiplicity constraints are resolved in models by applying all upper bound resolution rules as often as possible.
6. Finally, all lower bound multiplicity constraints are resolved in models by applying all lower bound resolution rules as often as possible.

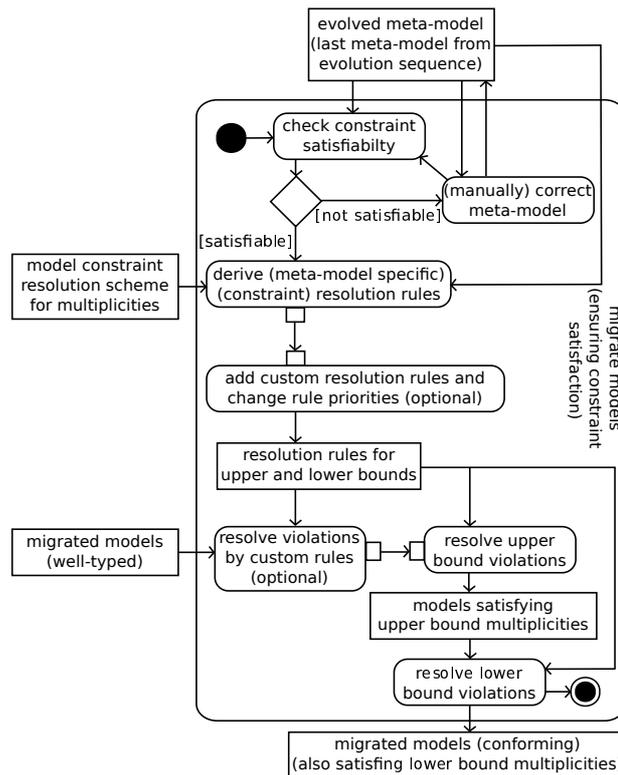
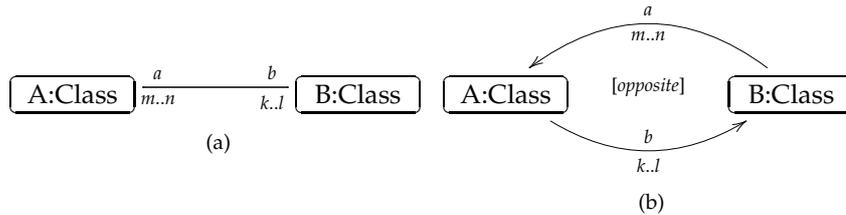


Figure 9.1: Resolving multiplicity violations

In subsequent examples, associations are used as compact notation for opposite references. This means the presentation of the association in Figure 9.2 (a) corresponds to the presentation in Figure 9.2 (b) using references. Multiplicity constraints and *[opposite]* can e.g. be annotated by meta-attributes or DPF atomic constraints (see Chapter 3).



with $0 \leq m \leq n \leq *$, $m \neq * \wedge n \neq 0$ and $0 \leq k \leq l \leq *$, $k \neq * \wedge l \neq 0$
 (* means finitely many)

Figure 9.2: Association as compact notation for opposite references

POSSIBLE
MULTIPLICITY
CHANGES

All possible multiplicity changes for reference *a* are listed in Table 9.1.⁵ We assume that changes to *m, n* respect their intervals (see Figure 9.2).

Table 9.1: Possible multiplicity changes (for reference *a*)

Change	Effect	Constraint
Increase lower bound ($x : Nat$)	$m + x$	stricter
Decrease lower bound ($x : Nat$)	$m - x$	weaker
Increase upper bound ($x : Nat$)	$n + x$	weaker
Decrease upper bound ($x : Nat$)	$n - x$	stricter

REFINEMENT OF
REFERENCES

In addition, we allow the *refinement of references*. MOF [110] offers the feature *subsets* for this purpose. This can e.g. be supported by formalizing references as vertices in category IGraph (see Chapter 3 and Figure 9.3). In Figure 9.3, reference b_0 has been refined by a reference b_1 . Figure 9.3 (a) shows the reference in its concrete syntax, while Figure 9.3 (b) shows the reference in its abstract syntax i.e. in category IGraph.⁶ In Figure 9.3 (a), the refinement/inheritance relationship is denoted by *subsets*. The corresponding super-reference is shown in curly brackets.

Dealing primarily with multiplicity changes here, we have to clarify how multiplicity constraints at super-references affect their sub-references:

⁵Changes for reference *b* are analog.

⁶Additional multiplicity constraints have been annotated as meta-attributes.

multiplicity references defined for super-references have to be satisfied by the set of all links typed by the super-reference or its sub-references. This means in Figure 9.3, each object of type A must have links of b_i types so that $k_0 \leq \sum_{i=0..1} \#(b_i) \leq l_0$ (“#” denotes the number as usual). In addition, multiplicities of sub-references need to be satisfied as usual. Note that also refinements of references can cause multiplicity changes, such as described in Table 9.1.

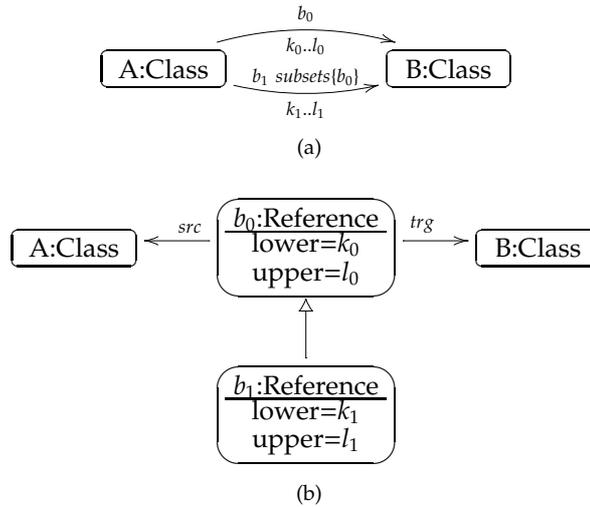


Figure 9.3: References in (a) concrete and (b) abstract syntax

Note that *reference vertex* b_1 in Figure 9.3 inherits edges *src* and *trg* from reference vertex b_0 . If the source or target of refined reference b_1 should be restricted to subclasses of class A respectively class B , additional *src* respectively *trg* edges have to be defined for b_1 . In models, we use the convention that *link vertices* that connect two objects use *all* available *src*, respectively *trg* edges.⁷ This ensures in graph transformation rules that links typed by sub-references can be matched by links typed by corresponding super-references.⁸

Furthermore, we allow classes and references to be *abstract*. This can be supported by a further meta-attribute or DPF atomic constraint. We give *abstract* the usual semantics, i.e. we define models that contain (direct) instances of abstract elements to be non-conforming.

ABSTRACT
ELEMENTS

Example 9.2.1 (Meta-model evolution with multiplicity adaptations). In the following, a concrete evolution scenario that focuses on multiplicity

⁷This can be easily ensured in a tool.

⁸I.e. including *src* and *trg* edges of the super-reference.

changes is given: we start with a simple activity meta-model (see Figure 9.4) containing two multiplicity constraints only to ensure that each transition has exactly one source and one target activity. Then, we refine this meta-model to produce a meta-model that ensures well-structured activity models to a certain extent. However, models conforming to the resulting meta-model can still be ill-structured with respect to additional constraints. Note that multiplicity constraints not shown are 0..*. Furthermore, if the lower bound of a multiplicity constraint is equal to the upper bound, only one number is denoted.

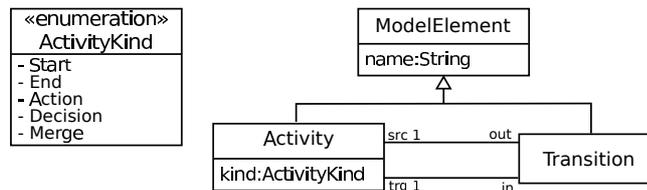


Figure 9.4: A meta-model for simple activity models

In Figure 9.5, the following evolution steps have been performed changing types and structure of the meta-model:

1. Class `Activity` has been marked as abstract i.e. all corresponding activities in models have to be retyped to a subclass of `Activity` or deleted.
2. Enumeration `ActivityKind` has been replaced by 5 subclasses of `Activity`.
3. References `src`, `trg`, `in` and `out` have also been marked as abstract, i.e. also all corresponding links have to be retyped or deleted.
4. References between `Activity` and `Transition` and vice versa are refined for all introduced subclasses of `Activity` using reference inheritance. For class `Start` and class `End`, unwanted references have been omitted. Note that this can be allowed here, as references are formalized by vertices. For example, `Start` inherits from `Activity`, however, there is no *concrete* reference in the meta-model that could be instantiated in a model to create a `trg`-link to a `Start` activity.

Furthermore, the following multiplicity changes have been performed:

1. All lower bound multiplicities of references targeting `Transition` have been increased.

2. All upper bound multiplicities of references targeting Transition have been decreased. Therefore Start activities are required to have exactly one outgoing transition and End activities exactly one incoming transition. Action activities are required to have exactly one incoming and one outgoing transition. Decision activities are required to have exactly one incoming and two outgoing transitions. Vice versa, Merge activities are required to have exactly two incoming transitions and one outgoing.
3. All multiplicities for references with source Transition targeting a subclass of Activity have not been set (i.e. are 0..*). Because the multiplicity constraints of the super-references src and trg also have to be satisfied, the constraint for transitions to connect exactly two activities has not been changed.

Hence, in the evolved meta-model stricter constraints have been introduced while no constraint has been weakened.

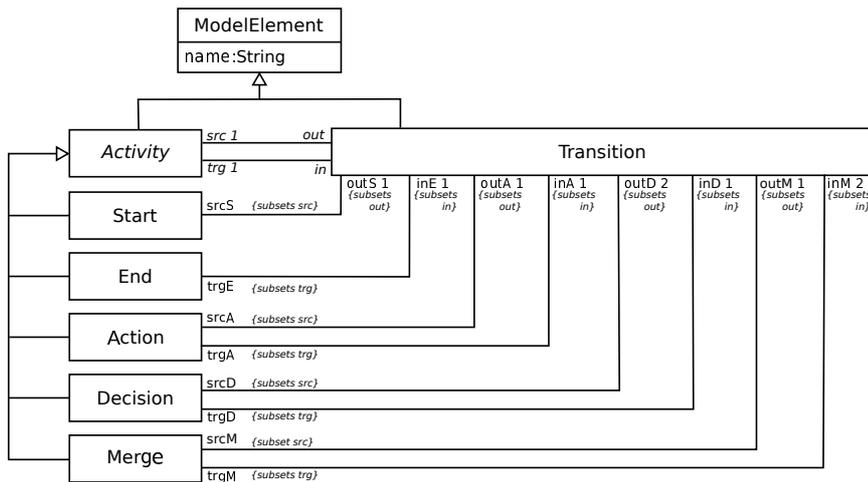


Figure 9.5: Evolved meta-model for activity models

9.3 Finitely Satisfiable Meta-models wrt. Multiplicities

Allowing meta-models with arbitrary multiplicities, it may happen that multiplicities are chosen such that no finite model can fulfill them. In the literature [12, 18], it is shown that this types of finite satisfiability

can be checked by solving a system of inequalities. Therefore, after meta-model evolution we first check that the resulting meta-model is still finitely satisfiable:

FINITELY
SATISFIABLE
META-MODELS

Assumption 9.3.1 (Finitely satisfiable meta-models (adapted fact from [18, 83])). *Given a meta-model MM, a system of inequalities I is built over C and R being variable sets for all classes and references in MM. Function $src : R \rightarrow C$ yields the class variable of the reference's source class. Function $clan^+ : E \rightarrow \mathcal{P}(E)$ yields the sum of all element variables (i.e. class or reference variables), which are in the clan of an element (including variables for sub-elements, excluding variables of abstract elements). Function $o : R \rightarrow R$ yields the variable of the opposite reference. Furthermore, function $lower : R \rightarrow \mathbb{N}$ yields the lower bound multiplicity constraint of a reference, respectively function $upper : R \rightarrow (\mathbb{N} \cup \{*\})$ its upper bound constraint. Then, we get the following inequalities in I for each reference variable r:*

1. $lower(r) \times clan^+(src(r)) \leq clan^+(r)$ if $lower(r) > 0$
2. $upper(r) \times clan^+(src(r)) \geq clan^+(r)$ if $upper(r) \neq *$
3. $clan^+(r) = clan^+(o(r))$

If the system of inequalities is solvable so that all $v \in C \cup R$ are assigned to numbers in \mathbb{N}^+ , then there exists a finite instance model containing elements of all concrete types that conforms to meta-model MM wrt. multiplicities. Meta-model MM, we call finitely satisfiable.

COMPLEXITY FOR
SOLVING SYSTEM OF
INEQUALITIES

The complexity to solve such a system of inequalities is EXPTIME-complete in general. However, the exponentiality depends on the maximum number of classes involved in the same generalization hierarchy, which is typically not very large (see also [18]).

Example 9.3.1. We now show the system of inequalities for the evolved meta-model in Example 9.2.1. As all defined upper and lower bound multiplicities are equal for each reference (or undefined i.e. 0..*), the system of inequalities can be reduced to a system of equations. For example:

$$1 \times Transition \leq srcS + srcA + srcD + srcM \text{ (src, Assumption 9.3.1 (1))}$$

$$1 \times Transition \geq srcS + srcA + srcD + srcM \text{ (src, Assumption 9.3.1 (2))}$$

can be reduced to: $Transition = srcS + srcA + srcD + srcM$.

The entire system of equations consists of:

$$Transition = srcS + srcA + srcD + srcM$$

$$Transition = trgE + trgA + trgD + trgM$$

9.4. Deriving Constraint Resolution Rules

$$\begin{array}{ll}
 \textit{Start} & = \textit{outS} & 2 \times \textit{Decision} & = \textit{outD} \\
 \textit{End} & = \textit{inE} & \textit{Decision} & = \textit{inD} \\
 \textit{Action} & = \textit{outA} & \textit{Merge} & = \textit{outM} \\
 \textit{Action} & = \textit{inA} & 2 \times \textit{Merge} & = \textit{inM}
 \end{array}$$

$$\begin{array}{l}
 \textit{srcS} + \textit{srcA} + \textit{srcD} + \textit{srcM} = \textit{outS} + \textit{outA} + \textit{outD} + \textit{outM} \\
 \textit{trgE} + \textit{trgA} + \textit{trgD} + \textit{trgM} = \textit{inE} + \textit{inA} + \textit{inD} + \textit{inM}
 \end{array}$$

$$\begin{array}{llll}
 \textit{srcS} = \textit{outS} & \textit{trgE} = \textit{inE} & \textit{srcA} = \textit{outA} & \textit{trgA} = \textit{inA} \\
 \textit{srcD} = \textit{outD} & \textit{trgD} = \textit{inD} & \textit{srcM} = \textit{outM} & \textit{trgM} = \textit{inM}
 \end{array}$$

The system of equations has solutions with all variables in \mathbb{N}^+ e.g.:

- $\textit{srcD} = \textit{outD} = \textit{trgM} = \textit{inM} = 2$
- $\textit{Action} = \textit{outA} = \textit{inA} = \textit{srcA} = \textit{trgA} = 3$
- $\textit{Transition} = 7$
- all other variables = 1

Hence, the meta-model is finitely satisfiable. Figure 9.6 shows a conforming model satisfying all multiplicity constraints using the numbers of elements above.

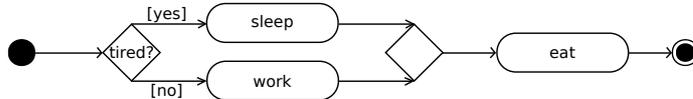


Figure 9.6: Conforming activity model

9.4 Deriving Constraint Resolution Rules

In the next step, we derive a set of graph transformation rules using rule templates that have been defined for resolving multiplicity constraint violations. To resolve further constraint violations, additional resolution schemes may be identified in the future. For example, we do not consider containment relationships here.

The constraint resolution scheme for multiplicity constraints consists of three types of templates and prioritize objects over links (i.e. only links may be deleted):

1. Templates for rules that delete links to satisfy upper bounds.
2. Templates for rules that create links to satisfy lower bounds.

TYPES OF
RESOLUTION RULES

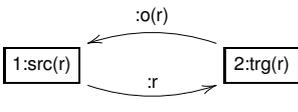
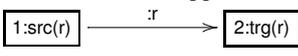
3. Templates for rules that create links with corresponding target objects (also to satisfy lower bounds). Rules of this type may induce that further lower bound multiplicity constraints need to be satisfied after their application.

In the following, all templates are sketched by only showing the left-hand sides (LHSs) and right-hand sides (RHSs) of rules in concrete syntax, as well as positive and negative application conditions. Intermediate graphs of rules can be computed as union graphs of LHS and RHS (as elements are neither merged nor split). In the rule templates, types of elements are specified by functions analog to the functions used in Assumption 9.3.1. However, functions that yield variables for element numbers in Assumption 9.3.1 yield types in this section. In the previous section, e.g. function $src : R \rightarrow C$ yields a variable for the source class of a reference. In this section, it yields the source class. In addition, the following functions are used in templates:

1. Function $trg : R \rightarrow C$ yields the target class of a reference.
2. Function $super : R \rightarrow R$ yields a super reference of a reference.⁹

Rules of Type 1 (see Table 9.2) are derived for all references with upper bound constraints ($\neq *$) and their sub-references. Herein, for references with opposite references, we use template r_1 , and for all other references template r_2 . Note that opposite references are paired (by a constraint) and hence always have to be created or deleted together. Rules created by such templates delete random links until the upper bound constraints of all references are satisfied. To not delete more links than required, positive application conditions are generated (see Table 9.3).

Table 9.2: Templates for rules of Type 1

<i>LHS</i>	<i>RHS</i>
r_1 : Delete link together with opposite (if \exists opposite reference) 	
r_2 : Delete link (if \nexists opposite reference) 	

⁹ For simplicity reasons, function $super : R \rightarrow R$ returns only one super-reference. Rules and application conditions are generated for all corresponding super-references, implicitly iterating over the set of all super-references.

9.4. Deriving Constraint Resolution Rules

The positive application conditions generated by the templates in Table 9.3 are used by all rules of Type 1. Note that links do not have targets in the PACs. Nevertheless, at the level of the abstract syntax, the PACs specify valid graphs, as links are vertices. Expressions such as “ $\times upper(r) + 1$ ” below links evaluate to the number of links to be generated. Furthermore, note that application conditions are generally only generated if expressions do not evaluate to “0”, respectively “*”. A rule of Type 1 can be applied *if*:

- the PAC generated by template $T - PAC_1$ applies or
- *one* PAC generated by template $T - PAC_2$ applies (*in addition* to the PAC generated by template $T - PAC_3$ *if this PAC exists*).

Table 9.3: PAC templates for rules of Type 1 (for both rules)

$T - PAC_1$	$T - PAC_2$
$\boxed{1:\text{src}(r)} \xrightarrow[\times upper(r)+1]{:r} \rightarrow$	$\boxed{1:\text{src}(r)} \xrightarrow[\times upper(\text{super}(r))+1]{:\text{super}(r)} \rightarrow$
$T - PAC_3$ (for $T - PAC_2$)	
$\boxed{1:\text{src}(r)} \xrightarrow[\times lower(r)+1]{:r} \rightarrow$	

PACs generated by Template $T - PAC_1$ ensure that rules are applied until the upper bound multiplicity constraint of corresponding references are satisfied. Template $T - PAC_2$ is used for generating an additional PAC for each upper bound multiplicity constraint of a super-reference. Such PACs analogously¹⁰ ensure that also links are deleted that violate inherited upper bound constraints. Hence, for each reference more than one PAC may be created since there may be more than one super-reference. Rules generated by template $T - PAC_3$ ensure that more references are deleted as allowed. In case an inherited upper bound constraint cannot be satisfied by applying a rule, there exists a different rule ensuring that the inherited upper bound constraint also will be satisfied in the end.

Example 9.4.1 (Sample rules of Type 1). The example builds on Example 9.2.1 and shows in Figure 9.7 all sample rules of Type 1 that are derived for the two references presented as association `srcD-outD` in Figure 9.5.

¹⁰Note that template $T - PAC_1$ could also be considered as special case of template $T - PAC_2$ if `super` would return a type including a reference’s direct type.

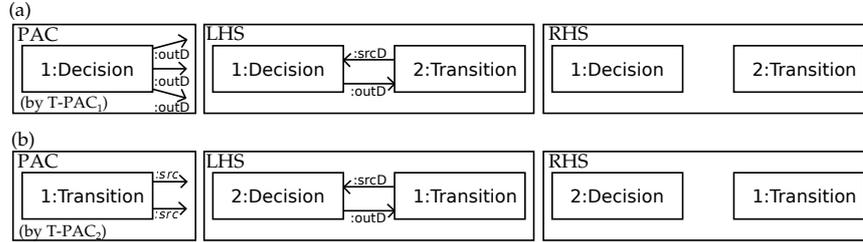


Figure 9.7: Sample rules of Type 1

Figure 9.7 (a) shows the rule that has been derived for reference `outD`. Figure 9.7 (b) shows the rule that has been derived for reference `srcD`. As both references are opposites, the “same” rules have been derived from Template r_1 in Table 9.2. Hence, the derived rules could theoretically be merged into one rule with two PACs. The PAC of the rule in Figure 9.7 (a) is generated by Template $T - PAC_1$. Because super-reference `out` does not have any multiplicity constraint (other than $0..*$), no further PACs are generated. The PAC of the rule in Figure 9.7 (b) is generated by Template $T - PAC_2$. The PAC is generated because the super-reference `src` has an upper bound constraint. As the reference `srcD` does not have any multiplicity constraint, no additional PAC is generated. Note that the used templates are denoted inside the figure for application conditions.

Rules of Type 2 (see Table 9.4) are derived for all references with lower bound constraints (> 0) and their sub-references. The rule templates are analogously defined to the rule templates of Type 1. Indeed, only the labels for LHS and RHS need to be swapped. Rules of Type 2 ensure that lower bound multiplicity constraints are satisfied in case a model already contains enough objects for creating necessary links. That rules do not create more links as necessary, two templates for generating NACs are used (see Table 9.5, $T - NAC_1$ and $T - NAC_2$). That rules do not create more links as allowed, two additional templates for generating NACs have been defined (see Table 9.5, $T - NAC_3$ and $T - NAC_4$). Further templates for NACs are required for rules generated by template r_3 , as multiplicity constraints of opposite references need to be considered (see Table 9.6). A rule of Type 2 can be applied *as long as* not:

- all NACs generated by Templates $T - NAC_1$ and $T - NAC_2$ apply or
- one NAC generated by Templates $T - NAC_3$, $T - NAC_4$, $T - NAC_5$ or $T - NAC_6$ applies.

Table 9.4: Templates for rules of Type 2

<i>LHS</i>	<i>RHS</i>
r_3 : Create link together with opposite (if \exists opposite reference) <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 2px;">1:src(r)</div> <div style="border: 1px solid black; padding: 2px;">2:trg(r)</div> </div>	<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 2px;">1:src(r)</div> <div style="border: 1px solid black; padding: 2px;">2:trg(r)</div> </div> <div style="text-align: center; margin-top: 5px;"> $\overset{:o(r)}{\curvearrowright}$ $\underset{:r}{\curvearrowleft}$ </div>
r_4 : Create link (if \nexists opposite reference) <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 2px;">1:src(r)</div> <div style="border: 1px solid black; padding: 2px;">2:trg(r)</div> </div>	<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 2px;">1:src(r)</div> <div style="border: 1px solid black; padding: 2px;">2:trg(r)</div> </div> <div style="text-align: center; margin-top: 5px;"> $\xrightarrow{:r}$ </div>

The negative application conditions generated by templates in Table 9.5 are used by all rules of Type 2 (and also Type 3). Note that NACs of all types are not always generated. For example, a reference may not have a lower bound but a super-reference has. In this case, the NAC for the super-reference is important. In addition, note that it is always ensured that a NAC considering a lower bound constraint exists.¹¹ NACs generated by templates $T - NAC_3$ and $T - NAC_4$ ensure that no upper bound constraints are violated by the rule application. Rules generated by templates $T - NAC_3$ are e.g. necessary if a reference has a lesser upper bound constraint than the lower bound constraint of a super-reference.

Table 9.5: NAC templates for rules of Type 2 and 3

$T - NAC_1$	$T - NAC_2$
<div style="display: flex; align-items: center; justify-content: center;"> <div style="border: 1px solid black; padding: 2px;">1:src(r)</div> <div style="margin: 0 10px;"> $\xrightarrow{:r}$ $\times lower(r)$ </div> </div>	<div style="display: flex; align-items: center; justify-content: center;"> <div style="border: 1px solid black; padding: 2px;">1:src(r)</div> <div style="margin: 0 10px;"> $\xrightarrow{:super(r)}$ $\times lower(super(r))$ </div> </div>
$T - NAC_3$	$T - NAC_4$
<div style="display: flex; align-items: center; justify-content: center;"> <div style="border: 1px solid black; padding: 2px;">1:src(r)</div> <div style="margin: 0 10px;"> $\xrightarrow{:r}$ $\times upper(r)$ </div> </div>	<div style="display: flex; align-items: center; justify-content: center;"> <div style="border: 1px solid black; padding: 2px;">1:src(r)</div> <div style="margin: 0 10px;"> $\xrightarrow{:super(r)}$ $\times upper(super(r))$ </div> </div>

The NACs generated by templates of Table 9.6 are only needed for rules generated by Template r_3 . They ensure that generated opposite links do not violate their upper bound multiplicity constraints.

¹¹Rules are only generated if there is a lower bound constraint.

Table 9.6: Templates for additional NACs generated for r_3 -Rules

$T - NAC_5$	$T - NAC_6$
$\boxed{2:\text{trg}(r)} \xrightarrow[\times \text{upper}(o(r))]{:o(r)}$	$\boxed{2:\text{trg}(r)} \xrightarrow[\times \text{upper}(\text{super}(o(r)))]{:\text{super}(o(r))}$

Example 9.4.2 (Sample rules of Type 2). The example builds on Example 9.2.1 and shows in Figure 9.8 all sample rules of Type 2 that are derived for the two references presented as association `srcD-outD` in Figure 9.5.

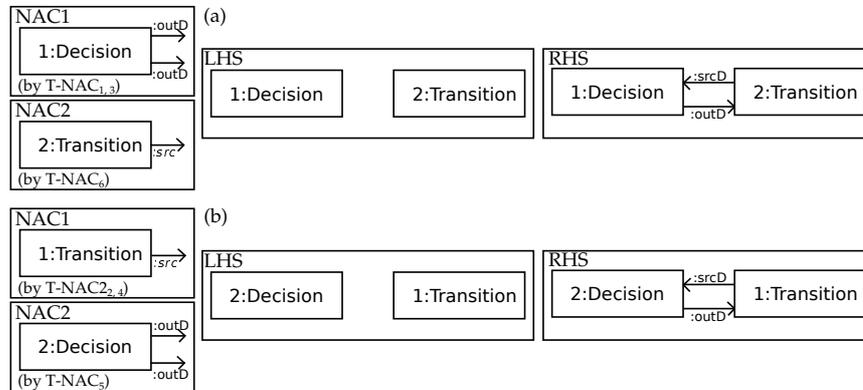


Figure 9.8: Sample rules of Type 2

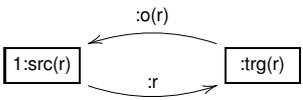
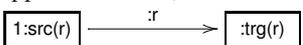
Figure 9.8 (a) shows the rule that has been derived for reference `outD`. Figure 9.8 (b) shows the rule that has been derived for reference `srcD`. Because both references are opposites, the same rules have been derived again; here by Template r_3 in Table 9.4. In this case, the same NACs also have been derived. However, the NACs derived for the rule in Figure 9.8 (a) result from different templates as in Figure 9.8 (b).

After applying rules of Type 1, we are not allowed to violate upper bound constraints again. With rules of Type 2, we try to satisfy lower bound constraints. As it is not always possible to satisfy all lower bound constraints by only adding links, additional rules of Type 3 are derived (see Table 9.7). Rules generated by templates of Type 3 create links together with target objects. Because also the target object of a link is new, such rules cannot induce new upper bound multiplicity constraint violations for created opposite links. Therefore, it is sufficient for rules of Type 3 to

9.4. Deriving Constraint Resolution Rules

generate NACs only by templates of Table 9.5. However, rules of Type 3 may introduce new objects that require resolving additional lower bound violations. Therefore, we prioritize rules of Type 2 and those rules of Type 3 that do not introduce such new constraint violations.

Table 9.7: Templates for rules of Type 3

<i>LHS</i>	<i>RHS</i>
r_5 : Create link, target object and opposite link (if \exists opposite reference) 	
r_6 : Create link and target object (if \nexists opposite reference) 	

Example 9.4.3 (Sample rules of Type 3). The example builds on Example 9.2.1 and shows in Figure 9.9 all sample rules of Type 3 that are derived for the two references presented as association *srcD-outD* in Figure 9.5.



Figure 9.9: Sample rules of Type 3

Figure 9.8 (a) shows the rule that has been derived for reference *outD*. Figure 9.8 (b) shows the rule that has been derived for reference *srcD*. Both rules differ because each rule creates an object of a different type. The same NACs as in Example 9.4.2 are generated for the rules in Figure 9.8, without those NACs previously generated by Templates $T - NAC_5$ and $T - NAC_6$.

9.5 Resolving Multiplicity Constraint Violations

Multiplicity constraints are resolved after models have been migrated to well-typed models (see Figure 1.3). Graph transformation rules are generated for this purpose specifically for the final meta-model version, as discussed in Section 9.4. Well-typed models are adapted to conforming models thereafter. To resolve multiplicity constraint violations, graph transformation rules are applied in three phases. We assume that the *abstract* constraint has been resolved before by either retyping instances of abstract elements or deleting them together with their context links.

Procedure 9.5.1 (Conflict resolution for multiplicity violations).

CONFLICT
RESOLUTION FOR
MULTIPLICITY
VIOLATIONS

1. First apply custom resolution rules as often as possible. Herein, matches are found randomly (as usual). Custom resolution rules are user defined graph transformation rules typed over the final meta-model version. That this phase terminates can be ensured by well-known results from graph transformations [36]. We propose that each custom rule should resolve an upper-bound multiplicity constraint violation without introducing a new upper-bound violation. In this case, this phase will trivially terminate, as finite models can contain only a finite number of violations.
2. In the second phase, rules of Type 1 are applied as often as possible. Because we consider only finite models, this phase trivially terminates as well, as each rule application only deletes. After this phase, all upper bound multiplicity constraints are satisfied because the generated rules delete as many links as necessary.
3. In the third, phase we apply all rules of Type 2 and Type 3 as often as possible until all lower bound constraint violations are also resolved. Note that rules of Type 2 and Type 3 are generated such that they cannot violate any upper bound constraint. To find a solution more quickly, we prioritize such rules in this phase that do not introduce new lower bound constraint violations. These are all rules of Type 2 and those rules of Type 3 that create objects only that do not violate lower bounds again. Inside this priority group again we prioritize rules of Type 3, as such rules introduce “terminal objects” and do not connect random elements. However, we allow inside this group customizing priorities. This is possible because such rules do not introduce new violations that have to be solved. That the phase terminates follows by a result given in [134], which presents an algorithm for instance-generating graph grammars that respect multiplicity constraints.

Note that rules are generally matched also non-injectively in Phase 2 and Phase 3, as sometimes multiplicity constraints may only be resolved by deleting or creating loop links.¹² Application conditions however are always matched injectively so that we can rely on that the number of specified links in PACs and NACs do exist (see Definition 4.7.1). In addition, we prioritize generally rules that delete or create links typed by *more specific references* in the inheritance hierarchy. This ensures that links typed by super-references are not deleted before all upper bound constraint violations of links typed by more specific references are resolved. Hence, more links are not deleted than necessary. Furthermore, this prioritization of rules avoids that the creation of links typed by super-references induce conflicts when links typed by sub-references need to be created.

Example 9.5.1 (Applying (conflict) resolution rules). The example builds again on Example 9.2.1 and shows how a sample model is adapted by (a) generated resolution rules (see Figure 9.10) (b) generated and custom resolution rules (see Figure 9.12). Model correction is performed by the stepwise application of resolution rules (see Procedure 9.5.1).

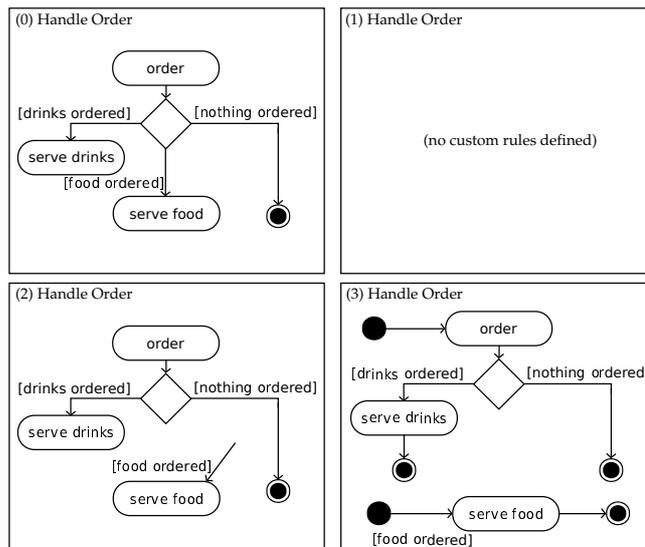


Figure 9.10: Sample constraint resolution

The sample model in Figure 9.10 (0) has already been migrated so that it is typed by the meta-model in Figure 9.5. Furthermore, we assume that models generally do not contain any abstract elements at this step. Abstract

¹²I.e. links that have the same object as source and target.

elements have been retyped, or deleted by the help of a trivial constraint resolution scheme that has been omitted here. However, models are not yet conforming to the meta-model, as several multiplicity constraints are not satisfied. The sample model describes how orders are placed in a restaurant. A waiter typically asks waiting customers what they would like to drink. After he/she has served the drinks, customers typically order food. However, sometimes the waiter arrives at a table and the customers have not decided or do not want to order anything more.

Figure 9.10 (1) corresponds to Phase 1. As no custom rules have been specified, this Figure is empty. Figure 9.10 (2) shows the model after Phase 2. One upper bound constraint is violated, therefore one outgoing transition has to be disconnected. Note that transitions are vertices and therefore the model is still a valid graph in its abstract syntax. The rule system chooses one random (outgoing) transition. Figure 9.10 (3) shows the model after Phase 3. Initial and end activity nodes have been introduced. Note that it would have been possible to connect Transition [food ordered] also with activity serve drinks. However, the default rule priority prioritizes the creation of new terminal objects.

Figure 9.11 shows a custom rule a language engineer specified to take the semantics of decision nodes into account. The rule replaces one decision node by two so that multiplicity constraint violations for outgoing transitions are properly solved. A step-wise application of this rule also allows resolving such multiplicity constraint violations for more than three links.

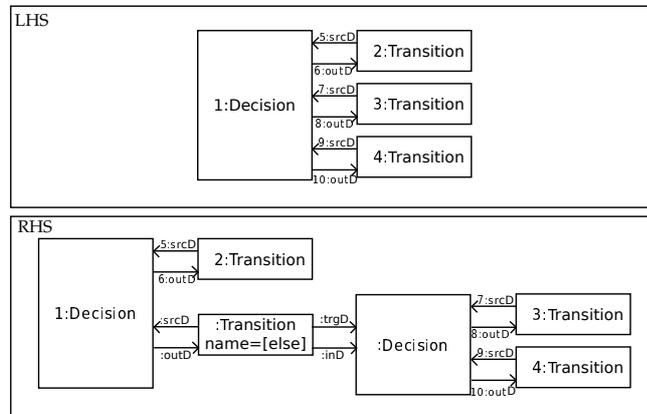


Figure 9.11: Sample custom rule for decision nodes

Figure 9.12 shows all conflict resolution phases again for the simple sample activity model introduced before using also the specified custom

9.5. Resolving Multiplicity Constraint Violations

resolution rule of Figure 9.11. Figure 9.12 (1) corresponds to Phase 1. The multiplicity constraint violation of the decision node is resolved adequately. Because no further upper bound constraints are violated, the model is not changed in Phase 2 (see Figure 9.12 (2)). In Phase 3, new terminal objects (i.e. start and end activities) are introduced as before, resolving all lower bound constraints without introducing new upper bound violations.

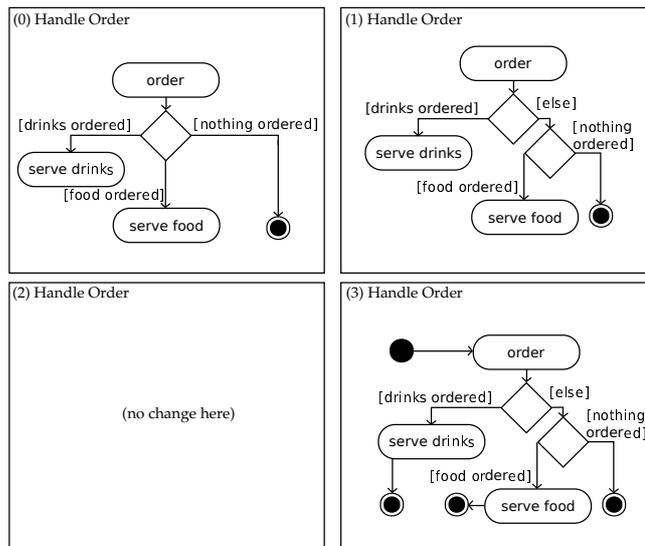


Figure 9.12: Sample constraint resolution with customization

In this chapter, an approach has been presented to resolve multiplicity constraint violations. The assumption has been that other violations such as for constraint *abstract* have been resolved before (e.g. by retyping or deleting). Furthermore, the presented approach considers the *opposite* constraint when multiplicity constraints are resolved, i.e. links are always deleted or created together with their opposites. The proposed approach uses a heuristic to suggest solutions. However, the constraint resolution can be adapted by a language engineer by defining custom rules. It is up to future work to also consider resolution schemes for other constraints such as containment relationships that are often used in modeling. Furthermore, it remains to examine which constraints can be resolved independent from each other and which constraints need to be resolved together. For example, it is obvious that *abstract* constraint violations can be resolved before multiplicity constraint violations are resolved.

Migrating UML Activity Models from Version 1.4 to 2.2

In this chapter, we examine and illustrate the proposed co-evolution approach on an example that has been considered earlier as a challenging “real world” problem. Herein, the evolution scenario under study originates from the Transformations Tool Contest 2010 [119]. We choose the scenario for two reasons: (1) many researchers agreed that the studied case is challenging and relevant for practice, and (2) nine different solutions have been proposed that can be compared to our solution. This chapter serves as proof-of-concept.

10.1 About the Transformations Tool Contest 2010

The Transformations Tool Contest (TTC) is a yearly contest for users and developers of transformation tools since 2007. In 2010, model migration was in the focus of the contest. The aim has been to compare different approaches and to motivate future model migration research. The model migration problem under study has been the migration of UML activity models from version 1.4 to version 2.2. Because UML activity models are widely used (e.g. to model workflows in Grid systems [153]), the migration problem has been considered as practically relevant. In addition, the migration problem has been considered as non-trivial, as the migration of activity models between versions 1.4 and 2.2 of the UML specification involved changes to the underlying semantics. In UML 1, activities were defined as a special case of state machines, while they are defined on top of a variant of colored Petri nets [62] in UML 2. Furthermore, it was pointed out that the chosen migration case was also not typical for

CORE TASK

model migration, as the pruned meta-model was more or less completely changed. Hence, the case study was considered as challenging.

PARTICIPANTS

Participants attended the workshop with the following tools: (1) Epsilon Flock [118], (2) COPE [57], (3) GrGen.NET [50], (4) Fujaba [44], (5) MOLA [99], (6) PETE [114], (7) ATL [6] and Java, (8) GReTL [61] and (9) UML-RSDS [140]. While (1) and (2) are special-purpose tools for model migration, (3), (4) and (8) are general graph transformation tools. The remaining tools are for model transformation. The contest participants had to submit a virtual machine containing their solution in addition to an accompanying document describing their solution before the workshop. These virtual machines were made available to the participants before the tool contest and hence could be analyzed by the other participants. During the workshop, solutions have been graded by the workshop participants after a presentation of each solution provider. Only one model had to be migrated in the core task.

EVALUATION
CRITERIA

Evaluated was in the TTC 2010 (*perceived*) *correctness, conciseness, understandability, appropriateness, (tool) maturity and support for the extension of the core task*. We use the case-study to evaluate: (1) Can all required migrations be expressed by the proposed approach? (2) Can we specify the required migrations in a concise and understandable manner? If the approach is appropriate, it is subjective and can hardly be evaluated in our opinion. Furthermore, we think that if an approach satisfies (1) and (2) and supports correct migration to some extent, it can also be considered as appropriate. Tool maturity we do not evaluate, as first its implementation has not been finished, and second it is not important here because we want to evaluate an approach and not a tool. “Perceived” correctness is also a criteria we think is not appropriate to evaluate an approach. Instead, the proposed approach supports statically typed rules and ensures that migration specifications are viable.

Additionally, there have been three (voluntary) extensions:

VOLUNTARY TASKS

1. The first extension resulted from a discussion on the tool contest’s online forum concerning an alternative migration of object flow states. The discussion revealed an ambiguity in the UML 2.2 specification, indicating that the migration semantics for the `ObjectFlowState` UML 1.4 concept are not clear from the UML 2.2 specification. This extension had been considered by all tools except (5) and (9).
2. The second extension considered the adaption of the concrete syntax “encoding,” which has not been in the focus of the thesis and will not be considered here. In addition, a solution has only been provided by one workshop participant who implemented the solution in Java.¹

¹ The OMG also does not provide any formal notation for the concrete syntax and only proposes to store it using XMI [119].

3. The third extension was more technical in nature, as models were provided in an older version of the XMI file format. This extension has been considered by tools (1) and (4), but is also not interesting for the theory developed in this thesis.

In the next section, we discuss the core task and the first extension in detail.

10.2 The Migration Task

In the tool contest, one model had to be migrated. A sample model is shown in Example 10.2.1. To illustrate our proposed solution, the model of the example is migrated later on.²

Example 10.2.1 (Activity model). Figure 10.1 shows a sample UML activity model. A customer makes an order by email. Afterward, the ordered items are prepared for the customer in the shop.

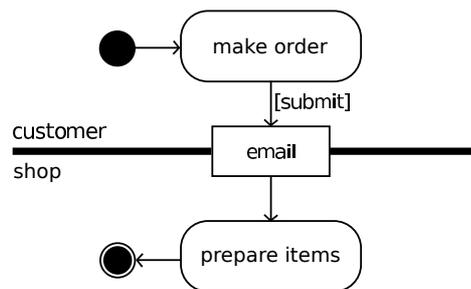


Figure 10.1: Workflow “Order items”

The sample model is presented in its concrete syntax and can be considered as an instance model of the UML 1.4 meta-model shown in Figure 10.2. However, because the concrete syntax for activity models did not change from UML 1.4 to UML 2.2, it can also be considered as an instance model of the UML 2.2 meta-model in Figure 10.3 [119]. Nevertheless, various adaptations of the model have to be done on the abstract syntax level to migrate it from UML 1.4 to UML 2.2.

² A smaller sample model as in the tool contest had been chosen, as the model will also be shown in its abstract syntax in a subsequent example. However, the model migration is as challenging as in the TTC and the proposed solution is applicable to any activity model.

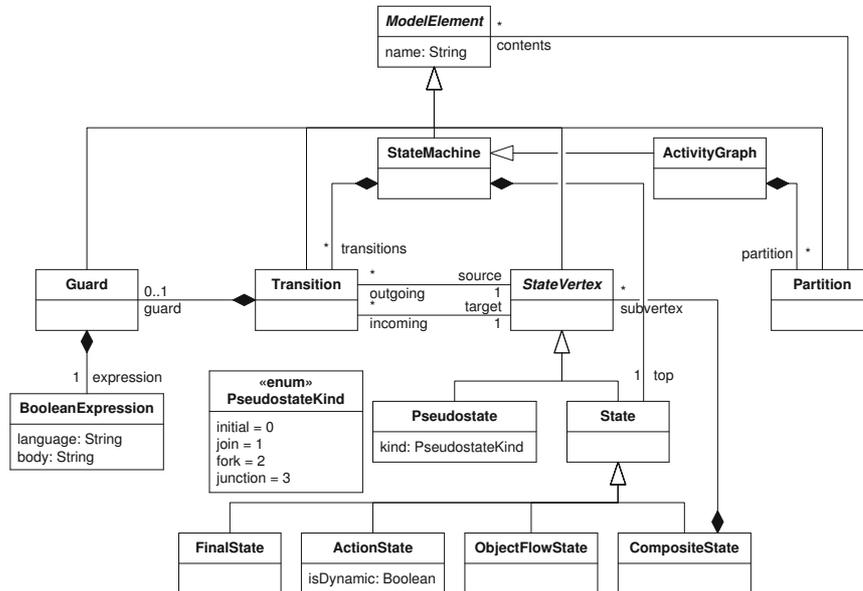


Figure 10.2: Activity meta-model in UML 1.4 (from [119])

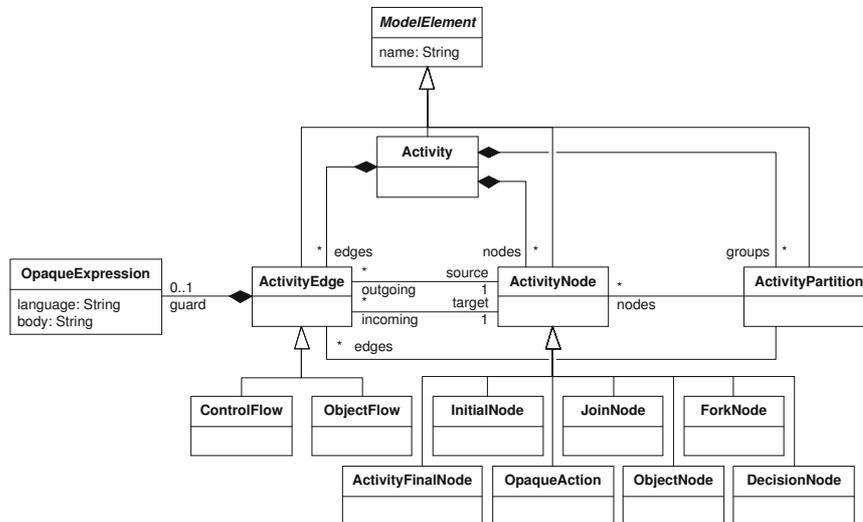


Figure 10.3: Activity meta-model in UML 2.2 (from [119])

To migrate activity models from UML 1.4 to UML 2.2, the following meta-model changes had to be addressed (from [119]):

«

- ActivityGraphs are now represented as **Activities**. The top [i.e. composition top;subvertex] and transition references are now represented using the nodes and edges references.
- Partitions are now represented as **ActivityPartitions**. The contents reference is now represented using the nodes and edges references.
- ActionStates are now represented as **OpaqueActions**.
- Pseudostates are now represented as a subtype of **ActivityNode**, such as **InitialNode** or **ForkNode**.
- Transitions are now represented as **ObjectFlows** or **ControlFlows**.
- Guards are now represented as **OpaqueExpressions**.”

DESCRIPTION OF THE CORE TASK

»

Extension 1 requires migrating **ObjectFlowStates** differently: Figure 10.4 (a) shows an **ObjectFlowState** structure in UML 1.4 semantics. The equivalent **ObjectNode** structure in UML 2.2 semantics according to the core task is shown in Figure 10.4 (b). The equivalent **ObjectNode** structure in UML 2.2 semantics according to Extension 1 is shown in Figure 10.4 (c).

DESCRIPTION OF EXTENSION 1

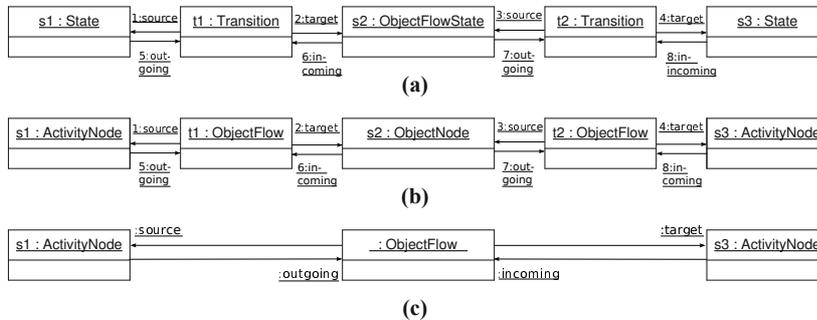


Figure 10.4: Alternative migration of object flow states (from [119])

Note that the original Figure 10.4 contained several errors (i.e. wrong type names, wrong identifier and missing references) and has therefore been corrected. In (c), identifiers have been left out if the elements are new and not identical to the elements in (a).

10.3 The DPF Text Modeling Framework

In the following, we consider all models, in particular *both* UML activity meta-models given in the DPF Text presentation. In the contest, models were given as EMF models, and had to be converted into tool specific formats in various approaches. For the theoretical approach developed in this thesis, the model format is of minor importance. However, having a concrete modeling format in mind helps to better understand the solution. DPF Text is a modeling framework that has been developed as part of the PhD project. It has already been introduced in Chapter 5 and is inspired by categories `IGraph`, `DPF` and `SymbGraphD`.³ Because it has been developed as an auxiliary tool to support the research on model migration, it shows how we think model migration could ideally be. A graphical notation using the textual notation is future work and may incorporate the graphical DPF tool [81] developed by our research group. While the basic concepts have been shown in Chapter 5, the main features of DPF Text are explained next:

- Every model element is a (typed) vertex or edge.
- Every model element has a unique identifier. The unique identifier consists of a set of integer values. If e.g. two types are merged, we assume the new identifier of the merged type to contain both integer sets. In case a type is split, we assume the new identifier to contain the old integer set and a new integer each. Using such identifiers gives us the possibility to effectively trace merges and splits of elements between (meta-)model versions. Data types we consider as vertices with a special identifier (special identifier we prefix with “S”, see Figure 10.5). Values we consider as vertices where the identifier is the value itself.
- Every model element has a name, except value identifiers (as they cannot be instantiated anymore). Names, however, provide labels for identifiers only. Mappings do not depend on names. Even models refer to types by names *and* unique identifiers, the names have been added only for the human reader. If a type name is changed in a meta-model, this does not have any effect on the model. It can still be opened and even shows the correct type, as type names are always read from the meta-model file.
- The tool allows creating arbitrary deep meta-model hierarchies. Herein, we always allow adding attributes on each level. Attributes can only be instantiated on the level below.

³Neglecting attributes and constraints, DPF Text implements category `IGraph` beside that every element has additionally a name.

- Arbitrary DPF atomic constraints can be added to the model as shown in Chapter 5. The constraint semantics are defined by OCL templates.

The meta-meta-model used to model the meta-models is shown in Figure 10.5 below. Identifiers are denoted as postfixes (“@identifier”). Note that the identifier of elements and types belong to different number sets.

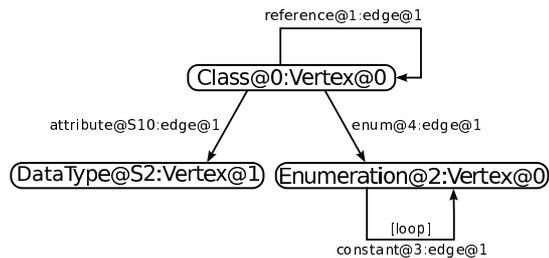


Figure 10.5: Used meta-meta-model

The meta-meta-model supports the concepts *classes*, *references*, *attributes*, *enumerations* and *inheritance*. Enumerations are modeled as vertices and each enumeration constant as loop edge. An object having a special enumeration type refers accordingly to an instance of the enumeration vertex having a loop instance for the type. Inheritance is modeled by special edges that can be used to specify clans of vertices as defined in category `IGraph` (see Chapter 3). Therefore, the meta-meta-model does not include an inheritance edge type. Constraints can be added by DPF atomic constraints (see Chapter 3). They are used for loops, containments, abstract classes, opposite references and multiplicities.

Listing 10.1 shows the meta-meta-model from Figure 10.5 as a textual presentation.

Listing 10.1: Figure 10.5 in DPF Text

```

1 Specification:(DPF ,TTCSig)<5> {
2   Graph {
3     Class@0:Vertex@0{
4       enum@4:edge@1->Enumeration@2:Vertex@0,
5       reference@1:edge@1->Class@0:Vertex@0
6     },
7     Enumeration@2:Vertex@0{
8       constant@3:edge@1->Enumeration@2:Vertex@0
9     }
10  }
11  Constraints {
12    loop@6(){Enumeration@2:Vertex@0 - constant@3:edge@1->Enumeration@2:Vertex@0}
13  }
14 }

```

Since attribution of vertices is built in the tool with some special support to deal with data types, the attribute type edge and data type vertex are implicit and do not appear in Listing 10.1.

Listing 10.2 shows the activity UML 1.4 meta-model in DPF Text and Listing 10.3 shows the activity UML 2.2 meta-model in DPF Text.

Listing 10.2: Figure 10.2 in DPF Text

```

1 Specification: (ClassModel ,TCSig) <36> {
2   Graph {
3     ActionState@13:Class@0 extends State@9:Class@0{
4       isDynamic@18:*->Boolean
5     },
6     ActivityGraph@3:Class@0 extends StateMachine@4:Class@0{
7       partition@29:reference@1->Partition@5:Class@0
8     },
9     BooleanExpression@15:Class@0{
10      body@16:*->String,
11      language@17:*->String
12    },
13    CompositeState@11:Class@0 extends State@9:Class@0{
14      subvertex@30:reference@1->StateVertex@6:Class@0
15    },
16    FinalState@14:Class@0 extends State@9:Class@0,
17    Guard@8:Class@0 extends ModelElement@1:Class@0{
18      expression@24:reference@1->BooleanExpression@15:Class@0
19    },
20    ModelElement@1:Class@0{
21      name@2:*->String
22    },
23    ObjectFlowState@12:Class@0 extends State@9:Class@0,
24    Partition@5:Class@0 extends ModelElement@1:Class@0{
25      contents@28:reference@1->ModelElement@1:Class@0
26    },
27    Pseudostate@10:Class@0 extends StateVertex@6:Class@0{
28      kind@31:enum@4->PseudostateKind@19:Enumeration@2
29    },
30    PseudostateKind@19:Enumeration@2{
31      fork@21:constant@3->PseudostateKind@19:Enumeration@2,
32      initial@23:constant@3->PseudostateKind@19:Enumeration@2,
33      join@22:constant@3->PseudostateKind@19:Enumeration@2,
34      junction@20:constant@3->PseudostateKind@19:Enumeration@2
35    },
36    State@9:Class@0 extends StateVertex@6:Class@0,
37    StateMachine@4:Class@0 extends ModelElement@1:Class@0{
38      top@27:reference@1->State@9:Class@0,
39      transitions@26:reference@1->Transition@7:Class@0
40    },
41    StateVertex@6:Class@0 extends ModelElement@1:Class@0{
42      incoming@34:reference@1->Transition@7:Class@0,
43      outgoing@35:reference@1->Transition@7:Class@0
44    },
45    Transition@7:Class@0 extends ModelElement@1:Class@0{
46      guard@25:reference@1->Guard@8:Class@0,
47      source@33:reference@1->StateVertex@6:Class@0,
48      target@32:reference@1->StateVertex@6:Class@0
49    }
50  }
51  Constraints {
52    /*ommitted to save space*/
53  }
54 }

```

Listing 10.3: Figure 10.3 in DPF Text

```

1 Specification:(ClassModel ,TTCSig)<45> {
2   Graph {
3     Activity@3,4:Class@0 extends ModelElement@1:Class@0{
4       edges@26:reference@1->ActivityEdge@7,8:Class@0,
5       groups@29:reference@1->ActivityPartition@5:Class@0,
6       nodes@44:reference@1->ActivityNode@6,9,10,11:Class@0
7     },
8     ActivityEdge@7,8:Class@0 extends ModelElement@1:Class@0{
9       guard@24:reference@1->OpaqueExpression@15:Class@0,
10      source@33:reference@1->ActivityNode@6,9,10,11:Class@0,
11      target@32:reference@1->ActivityNode@6,9,10,11:Class@0
12    },
13    ActivityFinalNode@14:Class@0 extends ActivityNode@6,9,10,11:Class@0,
14    ActivityNode@6,9,10,11:Class@0 extends ModelElement@1:Class@0{
15      incoming@34:reference@1->ActivityEdge@7,8:Class@0,
16      outgoing@35:reference@1->ActivityEdge@7,8:Class@0
17    },
18    ActivityPartition@5:Class@0 extends ModelElement@1:Class@0{
19      edges@41:reference@1->ActivityEdge@7,8:Class@0,
20      nodes@40:reference@1->ActivityNode@6,9,10,11:Class@0
21    },
22    ControlFlow@42:Class@0 extends ActivityEdge@7,8:Class@0,
23    DecisionNode@36:Class@0 extends ActivityNode@6,9,10,11:Class@0,
24    ForkNode@39:Class@0 extends ActivityNode@6,9,10,11:Class@0,
25    InitialNode@38:Class@0 extends ActivityNode@6,9,10,11:Class@0,
26    JoinNode@37:Class@0 extends ActivityNode@6,9,10,11:Class@0,
27    ModelElement@1:Class@0{
28      name@2:*->String
29    },
30    ObjectFlow@43:Class@0 extends ActivityEdge@7,8:Class@0,
31    ObjectNode@12:Class@0 extends ActivityNode@6,9,10,11:Class@0,
32    OpaqueAction@13:Class@0 extends ActivityNode@6,9,10,11:Class@0,
33    OpaqueExpression@15:Class@0{
34      body@16:*->String,
35      language@17:*->String
36    }
37  }
38  Constraints {
39    /*omitted to save space*/
40  }
41 }

```

The DPF Text presentation of the model in Example 10.2.1 is straightforward and has been omitted here. Instead, the model is shown in its abstract syntax in Example 10.4.1.

10.4 Model Migration by Coupled Transformations

We tackle the migration challenge in three steps:

1. Given the two meta-model versions, we need to find an evolution sequence. In the first step, we define coupled operators that can potentially be reused in future meta-model evolutions. Afterward, we detect meta-model evolution steps analog to Chapter 5 and evolve the UML 1.4 activity meta-model accordingly. Type names do not need to be considered, as such changes are non-breaking.

SOLUTION OUTLINE

2. Not all meta-model changes are likely to be reused in the future. Some meta-model changes are too specific such that it does not make sense to specify reusable operations. Therefore, we define a model migration scheme that is directly typed over the meta-model cospan “UML 1.4’(evolved by Step 1) → common meta-model ← UML 2.2”. The corresponding evolution rule (not required) can be considered as the trivial one: the rule is identical to the meta-model cospan. Match morphisms are all identity morphisms.
3. Finally, models can be migrated along the evolution sequence of Step 1 with the last step defined by Step 2. Migrated models are well-typed by the UML 2.2 activity model.

Last, but not least, multiplicity constraints could be handled analog to Chapter 9. However, this is not required here because multiplicity constraint did not become stricter.

We identify four meta-model evolution operators we can potentially reuse in the future:

REUSABLE OPERATIONS

1. “Inline Superclass” (and retype objects accordingly)
2. “Inline Class” (and merge and retype objects accordingly)
3. “Replace Enumeration Kind by Subclass”
4. “Partition Reference”

INLINE SUPERCLASS

The meta-model evolution rule “Inline Superclass” is shown in Figure 10.6. It simply merges a class and its superclass. The inheritance edge defines a clan of vertices (see Definition 3.2.7). It is therefore neither mapped nor folded.

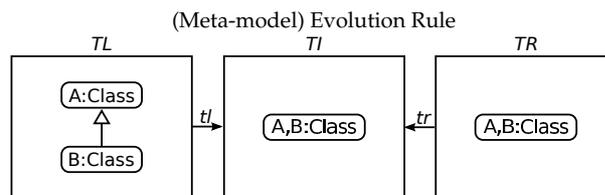


Figure 10.6: Evolution rule: “Inline Superclass”

The required migration for this change simply retypes that which can be handled by the empty migration scheme (which is also the default one). Therefore, we classify this change as *fully automatically resolvable*. Note in

10.4. Model Migration by Coupled Transformations

the following, element types in migration rules are shown by identifiers only. To make the rules more readable, letters are used for type identifiers instead of numbers.

The model migration scheme “Inline Class” (including its meta-model evolution rule) is shown in Figure 10.7. The meta-model evolution rule merges two classes and deletes the connecting reference. The required migration scheme is the default one. Hence, the change is *automatically resolvable*.

INLINE CLASS

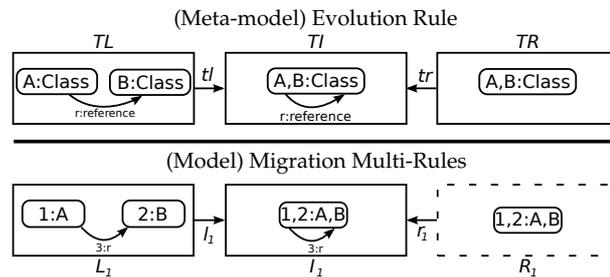


Figure 10.7: Migration scheme: “Inline Class”

The migration scheme “Replace Enumeration Kind by Subclass” is shown in Figure 10.8 below. The migration scheme has been defined manually. Hence, the meta-model change is *semi-automatically resolvable*.

REPLACE
ENUMERATION
KIND BY SUBCLASS

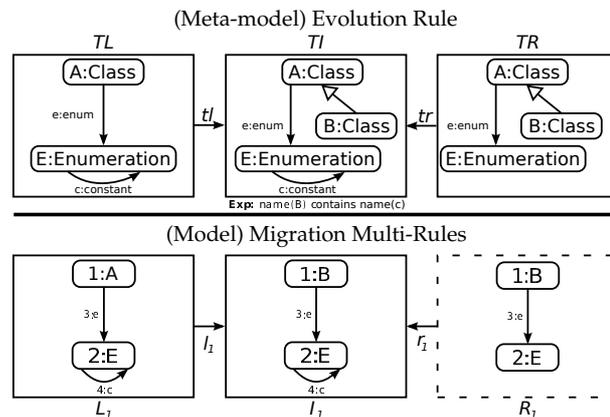


Figure 10.8: Migration scheme: “Replace Enumeration Kind by Subclass”

To infer the correct type names, an expression is used. This can be considered as a form of attribute condition. Unfortunately, we need to retype objects having enumeration constant junction to `DecisionNode`. There are many possibilities for solving this problem. We choose the simplest one and manually rename enumeration constant junction to `decision` in the UML 1.4 activity meta-model.

Alternatively to the migration scheme in Figure 10.8, a meta-model evolution rule could also be defined that replaces the complete enumeration by corresponding subclasses in one step. Note that by constraints, it is ensured that each object of type `Pseudostate` has exactly one type.

PARTITION
REFERENCE

The migration scheme “Partition Reference” is shown in Figure 10.9. The migration scheme has been manually defined. Therefore, the meta-model change is *semi-automatically resolvable*. Links typed by a reference targeting a superclass are replaced by links typed by a reference targeting a corresponding subclass. The meta-model evolution rule is defined as an amalgamated graph transformation rule and is applicable for an arbitrary number of references. Note that the migration multi-rule copies (in Procedure 7.2.1) are typed correspondingly to meta-model evolution multi-rule copies used to amalgamate the evolution rule.

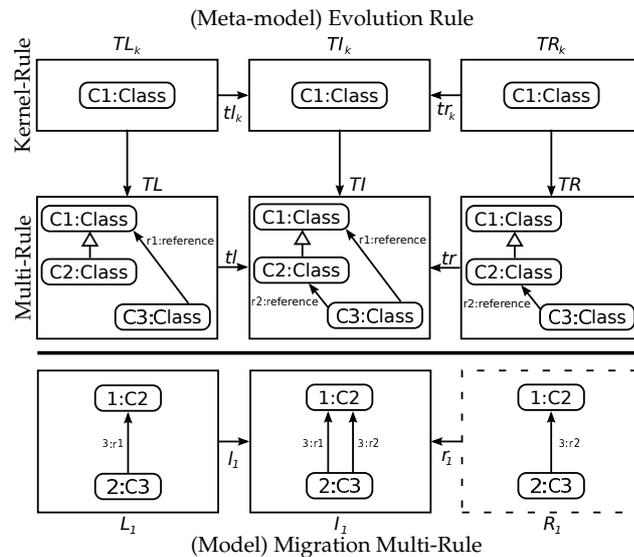


Figure 10.9: Migration scheme: “Partition Reference”

Meta-model changes are detected as described in Chapter 5, ten matches of evolution rules are found:

10.4. Model Migration by Coupled Transformations

- Four triple matches of Rule “Inline Superclass” are found since (1) StateMachine and ActivityGraph, (2) StateVertex and Pseudostate, (3) StateVertex and State as well as (4) State and CompositeState have been merged.
- One triple match of Rule “Inline Class” is found, as Guard and Transition have been merged. Note that merged vertices are leaf vertices, i.e. do not have subclasses, in the UML 1.4 meta-model.
- Four triple matches of the Rule “Replace Enumeration Kind by Subclass” are found, one for each enumeration constant in PseudostateKind.
- One amalgamated triple match of the Rule “Partition Reference” is found replacing links of type contents by links of type edges⁴ respectively nodes.

DETECTED
META-MODEL
CHANGES

We obtain an arbitrary order of evolution steps because all changes are independent.

Remark 10.4.1 (Subtype-reflecting evolution matches). To build coupled transformations, it is on the formal level required to have subtype-reflecting evolution matches in category IGraph. In particular, subtype reflective matches of evolution rules ensure that the pullback construction in Step 1 in Procedure 8.5.1 can be applied. Subtype-reflecting matches guarantee that the pullback in Step 1 results in a left-hand-side of a migration rule that also contains all possible instances of matched subtypes. Those may need to be considered during migration. In a tool, such instances can easily be added to the left-hand-side of the migration rule even if the match of the evolution rule has not been subtype-reflecting. On the formal level, this can be achieved by extending the match $tm : TL \rightarrow TG$ and the detected evolution rule correspondingly before applying Procedure 8.5.1, i.e. before migration (see Figure 10.10). This in particular means that in a tool it would be unnecessary to specify individual rules to obtain subtype-reflecting matches. It is even beneficial during change detection to not consider possible inheritance hierarchies, as additional changes in the hierarchy may prevent finding a triple match (see Definition 5.2.1).

Figure 10.10 shows a simple evolution rule deleting a reference. The reference’s source vertex is matched to a vertex in the meta-model TG that has subclasses. Therefore, match tm and the evolution rule is extended correspondingly. The inheritance hierarchy below the matched superclass is denoted by a triangle. Hence, the triangle represents the tail of an arbitrary inheritance hierarchy with class Car in the top. A tool could easily extend the rule to make the match subtype-reflecting.

⁴The edges type with source Partition and target ActivityEdge.

We continue by defining a model migration scheme containing migration rules directly typed by the current meta-model cospan such as in manual specification approaches. The rules of this migration scheme are rules that we do not expect to be reusable. Indeed, they are not reusable due to this direct typing. In this case, the graph transformation in the top face of the coupled transformation can be considered as the trivial one, i.e. the meta-model cospan and the rule are identical and all match morphisms are identity morphisms (see Chapter 6). Trivially, we can transfer in this case all constructions to category `Graph` and do not need to obey any restriction that was special for category `IGraph` (see Chapter 8).

By help of this scheme, we handle the remaining model adoptions. Renaming of types⁵ and deletions⁶ due to type deletions are not required to be considered because they are handled automatically. In addition, the adaptations of constraint annotations do not need to be considered, as finally the evolved meta-model is used.

For the *core* task, a migration scheme (see Figure 10.12) containing three multi-rules is required:

- One migration rule to replace the composition of `top` and `subvertex` links by `nodes`⁷ links because there has been a containment change in UML 2.2 (see Figure 10.12 (first rule)).
- One migration rule to retype `Transition` vertices (i.e. `ActivityEdge` vertices in UML 2.2 syntax) to the new subtype `ObjectFlow` if corresponding `source` or `target` links are connected to vertices of type `ObjectFlowState` (see Figure 10.12 (second rule)).
- Another rule to retype `Transition` vertices to the new subtype `ControlFlow` in all other cases (see Figure 10.12 (third rule)). Because it is obvious that the second and the third rule cannot be matched so that both matches overlap (ensured by the application conditions), it is not required that `ObjectFlow` and `ControlFlow` have a common subtype.

Note that in the following, we show types by names (without identifier) in migration multi-rules only to make them more readable.

CUSTOM
MIGRATION RULES

⁵ `FinalState`, `ActionState`, `ObjectFlowState`, `partition`, `transitions` and `expression`

⁶ `PseudostateKind`, `kind`, `isDynamic`, `top` and `subvertex`

⁷ Type `nodes` has source `Activity` and target `ActivityNode`.

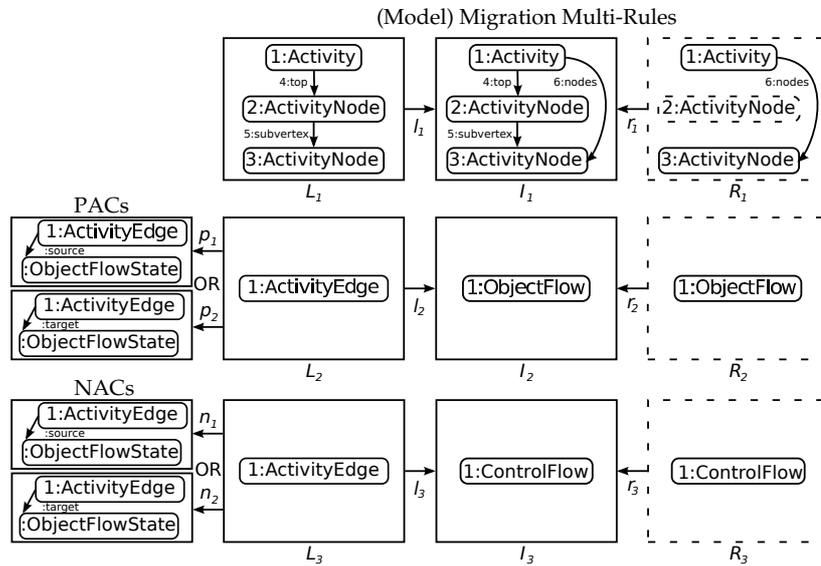


Figure 10.12: Migration scheme typed by meta-model cospan

Finally, we can migrate all models in a batch along the evolution sequence.

Example 10.4.1 (Model migration). Example 10.4.1 shows the migration of the sample model of Example 10.2.1 (also shown below) for the core task.

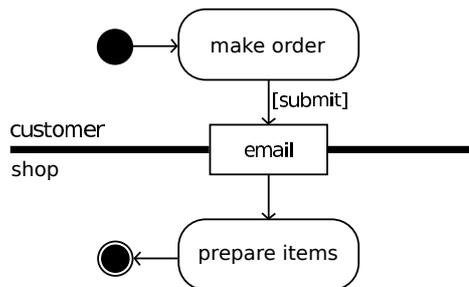


Figure 10.13: Workflow “Order items”

Figure 10.14 shows the model of Figure 10.13 in abstract syntax i.e. as object diagram typed by the UML 1.4 activity meta-model (see Figure 10.2).

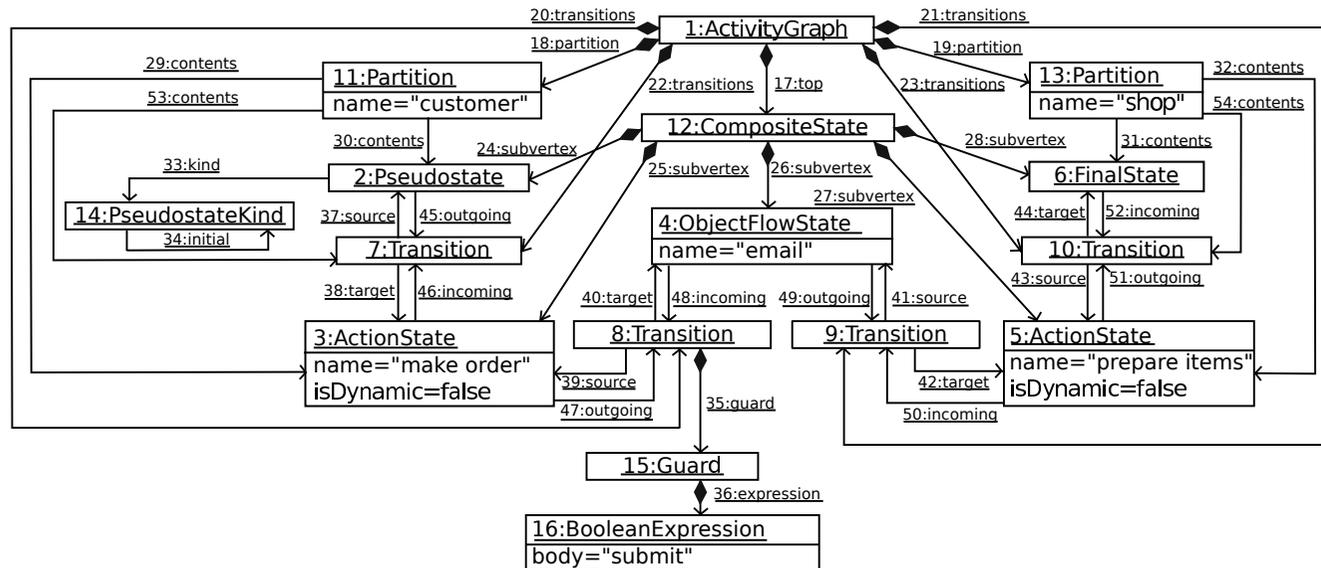


Figure 10.14: Activity model in UML 1.4

Figure 10.15 shows the partly migrated model after the deduced model migrations steps have been applied for all detected meta-model evolution changes (i.e. the model is well-typed by the meta-model in Figure 10.11). Identifiers of new elements start with number 80.

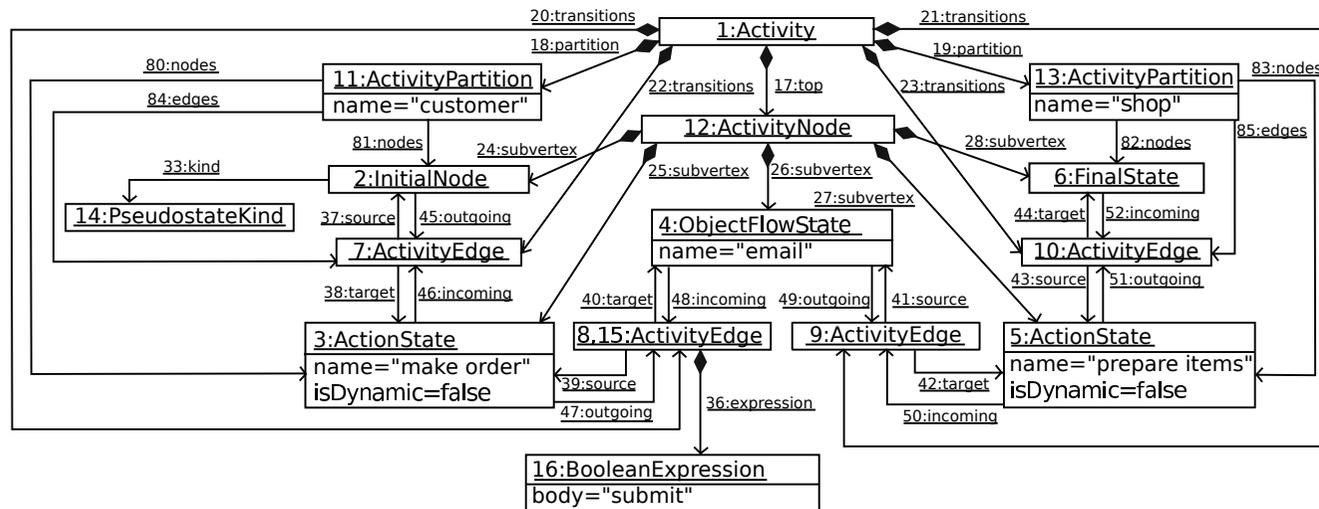


Figure 10.15: Activity model in UML 1.4 (partly evolved)

Figure 10.16 shows the migrated model after the migration scheme that is individual for the migration of activity meta-models also has been applied.

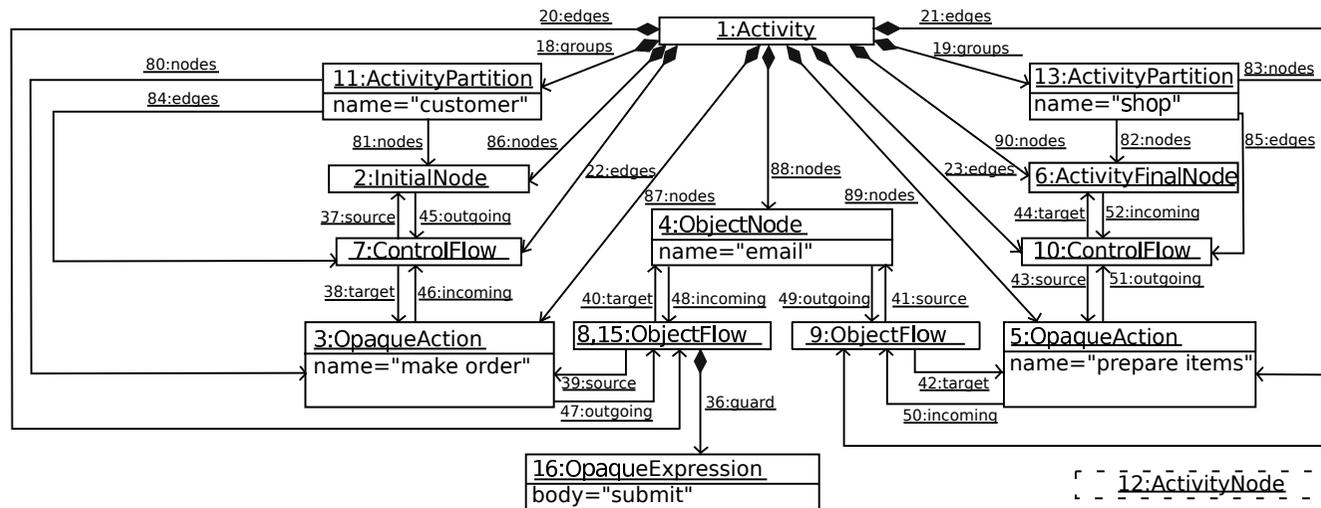


Figure 10.16: Activity model in UML 2.2

One vertex **12:ActivityNode** (**12:CompositeState**) remains that could be deleted after migration. Because type **ActivityNode** is abstract this can e.g. be automatically done by a corresponding *model constraint resolution schemes* for constraint *abstract*.

Each UML 1.4 activity model is migrated by eleven amalgamated graph transformations. Some of those rules are doing nothing in the case of the model in Figure 10.14, as e.g. the model does not contain any other pseudostates than with type *initial*. Nevertheless, showing all remaining eight graph transformation rules has been omitted for space restrictions. Instead, all deduced changes have been listed in Listing 10.4 by six different types of simple changes:

1. Change *Retype* changes the typing of an element. Herein, *retype* describes a type change where the identifier of the type is changed due to a merge operation.
2. Change *Subtype* changes a type of an element to a subtype of its current type.
3. Change **Pseudo*-Retype* describes a retyping where only the type name has been changed (i.e. a non-breaking meta-model change). The change is called *pseudo* retyping, as the identifier of the type is not changed i.e. DPF text shows the adapted type names automatically without migration.
4. Change *Merge* merges elements and also retypes here because corresponding types are merged too.
5. Change *Delete* deletes an element.
6. Change *Replace* adds a new element and deletes an old one.

Note that elements and in particular types in Listing 10.4 are annotated with identifiers. Such identifier can be compared to the identifier used in Listing 10.2 and Listing 10.3 as well as in Figure 10.14, Figure 10.15 and Figure 10.16. The listing also notes which evolution rules and which matches trigger the adaptations. Herein, the text after *Match* should be understood as a comment, i.e. the real match may contain additional elements.

Listing 10.4: Deduced Migration for the model of Figure 10.14

```

1 ===== START =====
2 Evolution-Rule 1: "Inline Superclass"
3 Match:  "ActivityGraph and StateMachine"      (to Activity type)
4 -----
5 Retype  1:ActivityGraph@3 to 1:Activity@3,4
6 -----
7 Match:  "StateVertex and Pseudostate"        (to ActivityNode type)
8 -----
9 Retype  2:Pseudostate@10 to 2:ActivityNode@6,10
10 -----
11 Match:  "ActivityNode and State"             (to ActivityNode type)
12 -----
13 Retype  2:ActivityNode@6,10 to 2:ActivityNode@6,9,10
14 -----

```

10.4. Model Migration by Coupled Transformations

```

15 Match: "ActivityNode and CompositeState" (to ActivityNode type)
16 -----
17 Retype 2:ActivityNode@6,9,10 to 2:ActivityNode@6,9,10,11
18 Retype 12:CompositeState@11 to 2:ActivityNode@6,9,10,11
19 =====
20 Evolution-Rule 2: "Inline Class"
21 Match: "Guard and Transition" (to ActivityEdge type)
22 -----
23 Merge: 8:Transition@7 + 15:Guard@8 to 8,15:ActivityEdge@7,8
24 Delete: 35:guard@24
25 Retype 7:Transition@7 to 7:ActivityEdge@7,8
26 Retype 9:Transition@7 to 9:ActivityEdge@7,8
27 Retype 10:Transition@7 to 10:ActivityEdge@7,8
28 =====
29 Evolution-Rule 3: "Replace Enumeration Kind by Subclass"
30 Match: "ActivityNode and PseudostateKind initial"
31 -----
32 Subtype 2:ActivityNode@6,10 to 2:InitialNode@38
33 Delete 34:initial@23
34 =====
35 Evolution-Rule 3: "Partition Reference"
36 Match: "Partition and partition edge to ModelElement"
37 -----
38 Replace 29:contents@28 by 80:nodes@40
39 Replace 30:contents@28 by 81:nodes@40
40 Replace 31:contents@28 by 82:nodes@40
41 Replace 32:contents@28 by 83:nodes@40
42 Replace 53:contents@28 by 84:edges@41
43 Replace 54:contents@28 by 85:edges@41
44 *Pseudo*-Retype: 11:Partition@5 to 11:ActivityPartition@5
45 *Pseudo*-Retype: 13:Partition@5 to 13:ActivityPartition@5
46 =====
47 Evolution-Rule: Id-Rule (Manual-Specification)
48 -----Multi-Rule 1-----
49 Replace 24:subvertex@30 by 86:nodes@44
50 Replace 25:subvertex@30 by 87:nodes@44
51 Replace 26:subvertex@30 by 88:nodes@44
52 Replace 27:subvertex@30 by 89:nodes@44
53 Replace 28:subvertex@30 by 90:nodes@44
54 Delete 17:top@27
55 -----Multi-Rule 2-----
56 Subtype 8,15:ActivityEdge@7,8 to 8,15:ObjectFlow@43
57 Subtype 9:ActivityEdge@7,8 to 9:ObjectFlow@43
58 -----Multi-Rule 3-----
59 Subtype 7:ActivityEdge@7,8 to 7:ControlFlow@42
60 Subtype 10:ActivityEdge@7,8 to 10:ControlFlow@42
61 -----
62 ----- Without Rule (by construction) -----
63 -----
64 Delete 14:PseudostateKind@19
65 Delete 33:kind@31
66 Delete 55:isDynamic@18
67 Delete 56:isDynamic@18
68 *Pseudo*-Retype 3:ActionState@13 to 3:OpaqueAction@13
69 *Pseudo*-Retype 4:ObjectFlowState@12 to 4:ObjectNode@12
70 *Pseudo*-Retype 5:ActionState@13 to 5:OpaqueAction@13
71 *Pseudo*-Retype 6:FinalState@14 to 6:ActivityFinalNode@14
72 *Pseudo*-Retype 16:BooleanExpression@15 to 16:OpaqueExpression@15
73 *Pseudo*-Retype 18:partition@29 to 18:groups@29
74 *Pseudo*-Retype 19:partition@29 to 19:groups@29
75 *Pseudo*-Retype 20:transitions@26 to 20:edges@26
76 *Pseudo*-Retype 21:transitions@26 to 21:edges@26
77 *Pseudo*-Retype 22:transitions@26 to 22:edges@26
78 *Pseudo*-Retype 23:transitions@26 to 23:edges@26
79 *Pseudo*-Retype 36:expression@24 to 36:guard@24
80 ===== STOP =====

```

SOLUTION
EXTENSION 1

It remains to discuss which changes need to be done to address *Extension 1*. Extension 1 requires a different migration of *ObjectFlows*. To migrate models accordingly, we only need to replace the second rule of the migration scheme in Figure 10.12 by the rule shown in Figure 10.17. The migration rule corresponds to a large extent to the pattern shown in Figure 10.4. To make it easy to compare both figures, we formatted the rule in the concrete syntax used in Figure 10.4. There are two differences to Figure 10.4 (a+c): first, the left-hand side of the rule is typed by the evolved meta-model shown in Figure 10.11. Second, the rule also creates an edge typed by containment edge edges. Note that there is only one root container.

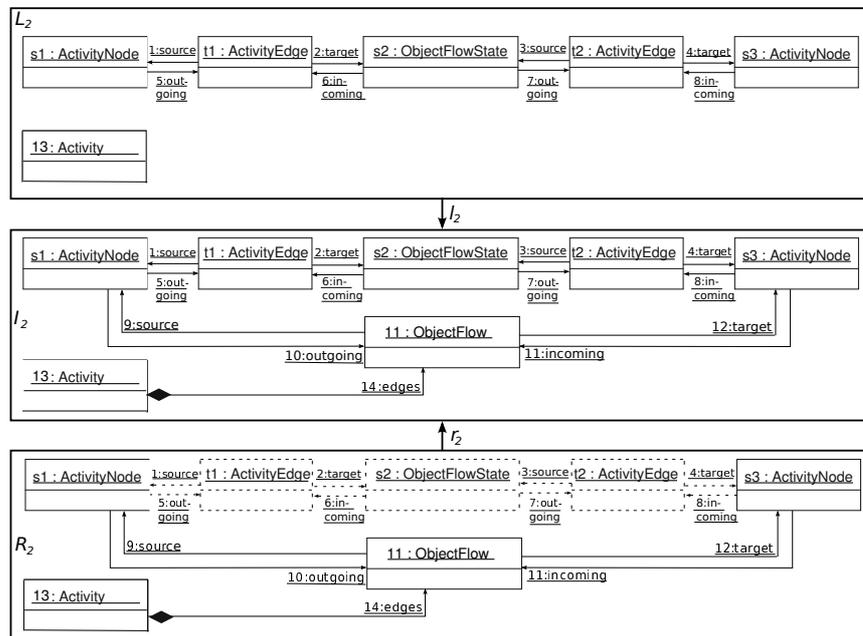


Figure 10.17: Migration rule for *ObjectFlows*

Unfortunately, the migration also requires deleting elements of types that are not deleted (elements with dashed border). To cope with this situation, there are different possibilities:

- We can extend the amalgamation procedure to also delete the dashed elements. Therefore, we either need to extend the rule to also delete necessary containment edges or apply the rule with the cospan SqPO approach (which deletes in unknown context). The second possibility has the advantage that the migration of models is always viable

but has the disadvantage that unexpected elements may be deleted. Note that `ObjectFlowStates` may be the source or target of several `ActivityEdges`. If the first possibility is used the gluing condition needs to be checked for each model. If the condition is not satisfied the amalgamated rule needs to be adapted.

- We delete the undesired patterns by applying a “usual” graph transformation rule as often as possible in a post-processing step.

Remark 10.4.2 (Syntax versus semantics). Algebraic graph transformations can be used to manipulate graphs based on graph patterns. However, sometimes such patterns are hard to define statically. At the PAMT 2014 workshop⁸, a participant asked us how we would address the challenge of translating integer values specifying token numbers in a Petri net by an equivalent structure, where each token is presented by a vertex. This challenge requires giving data values a semantics that is hard to describe by pure graph patterns. Therefore, we propose to use dynamic rules for such cases. Such rules can be graph transformation rules where the interface graph is generated by a template after a match of the left hand side is found. Figure 10.18 shows a migration scheme (including a meta-model evolution rule) that can be used as a coupled operator in the Petri net example. A variable v is translated into an equivalent number of elements. The number bound to variable v is not deleted because in the formalization every number uniquely exists in the universe. However, unused numbers are not explicitly stored by DPF Text.

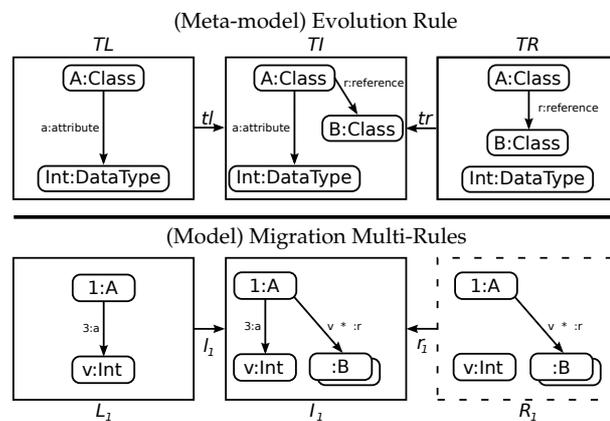


Figure 10.18: Migration scheme employing template

⁸<http://rosacreative.nl/PAMT/>

10.5 On the Results of the Transformation Tool Contest

WINNING CONTEST SOLUTIONS	<p>In the TTC 2010, the case provider's tools Flock was ranked first in the overall score list, while COPE, the second tool explicitly made for model migration, was ranked second. The tools in places 3 and 4 were the general purpose graph transformation tools GrGen.net and Fujaba. We will compare the proposed solution to the two winning solutions in the following.</p> <p>Flock, a manual specification approach, scored best because the workshop participants evaluated high that Flock's migration programs are concise and understandable. Furthermore, Flock scored best for appropriateness and extensions. However, Flock scored very low (place 7) on perceived correctness because of the perceived difficulty in identifying when a Flock migration strategy is complete. We experienced the same difficulties using the tool. In addition, as rules are not statically type checked, mistakes can cause additional information loses. In addition, Flock's features to re-type vertices by edges (used to implement Extension 1) we experienced as confusing.</p>
EPSILON FLOCK	<p>COPE, an operator-based approach, was ranked second. COPE also scored high in perceived correctness because many co-evolutions steps could be handled by its predefined operation library that has been considered as correct. COPE was in particular considered less concise than Flock, as user-defined operations in COPE need to be programmed in a general-purpose programming language (Groovy). We experienced COPE as a nice tool to work with if the predefined operations are sufficient. However, its likely that there are situations where operations need to be defined manually, because a required operation is missing or the preconditions for an operation are not satisfied.</p> <p>According to [119], support for retyping and in-place transformations had a strong positive effect on conciseness. That a tool's special purpose is model migration had a strong positive impact on conciseness, understandability and appropriateness. Other criteria did not have strong impacts.</p>
COPE	<p>The proposed solution here combines strength of both winning approaches. We used reusable operators first (as in COPE) and afterward special purpose migration rules (as in Flock). The proposed solution supports retyping as well as in-place transformations. Furthermore, instead of applying reusable operators manually, we detected the meta-model changes. Only one migration scheme containing three rules (core tasks) had to be specified manually that is specific for the migration of UML activity models. In total, we specified less rules than the winning solutions, i.e. the proposed solution is concise. We also think the solution is very understandable, as the solution is based on migration pattern that can be visualized.</p>
PROPOSED SOLUTION	

As pointed out in [119], the correctness criteria in the TTC 2010 has been very weak even if weighted as most important in the tool contest. Therefore, we propose using stricter correctness criteria in future tool contests. For example, we propose asking the following questions: (1) Are migration rules statically typed? (2) Is it ensured that the migration specifications are viable? (3) Is it still true if migration rules are specified manually or have been customized? (4) What type of constraints can be ensured and how?

CORRECTNESS
CRITERIA

The proposed approach is based on statically typed rules. It is proven that migration specifications are always viable. Furthermore, we support the well-defined customization of model migration specifications. In addition, we can ensure the satisfaction of multiplicity constraints as shown in Chapter 9.

In this chapter, it has been demonstrated that the proposed model co-evolution approach is well suited to the challenge of model migration. All required migration steps could be specified in a concise and understandable manner. Furthermore, the approach supports a stricter form of correctness for model migration specifications as used in the TTC 2010.

Related Work

In the following, we discuss related work in two different areas: first, we briefly review the achievements in database schema evolution. Then we go over considering different aspects of meta-model co-evolution in MDE.

11.1 Schema Evolution

Database schema evolution has been studied for relational as well as for object-oriented databases. In particular, schema co-evolution in object-oriented databases has similarities to meta-model co-evolution in MDE. While it is far beyond the scope of a thesis to describe all types of related work on schema evolution, we focus on recent and frequently cited works in the following.

Schema evolution for relational databases has been studied e.g. in the PRISM project by Curino et al. [25, 26, 27]. According to the authors, their tool PRISM is the most advanced system supporting relational schema evolution in practice today [25]. The PRISM workbench supports schema modification operators (SMOs) that capture atomic operations to evolve schemes. In [25], 11 SMOs are supported, which span from simple operations such as create/drop/rename table to more complex ones such as merge or partition table. From SMO sequences, PRISM generates SQL migration scripts. Furthermore, it supports automatic SQL query rewriting and the generation of database views mimicking the previous schema version. In [26], a new version of the tool is presented that in addition supports 6 new operators to manipulate integrity constraints namely key, foreign key and value constraints. In the new version, query rewriting is semi-automatically supported.

FORMAL ISSUES IN
SCHEMA
EVOLUTION

Formal issues of schema mapping, mapping composition, invertibility and query rewriting for relational databases has been considered by a number of authors, e.g. Nash [100], Bernstein [13], Fagin [40] and Deutsch [31]. Because our approach is independent of a specific modeling approach, it is up to future work to find a suitable adhesive category of relational algebras and to compare our theoretical results for that category with formal issues mentioned above.

SCHEMA
EVOLUTION FOR
OBJECT-ORIENTED
DATABASES

Schema evolution for object-oriented databases has also been studied for many years. A survey on this topic can be found in [85]. For the ORION database, for example, Banerjee et al. [9] define a set of 5 invariants and 12 change primitives [115] that ensure these invariants. Similar approaches exist for the GemStone [115] and O_2 [41] databases. Claypool et al. present in [23] a framework for defining reusable change operations based on templates that employ primitive operations as well as database queries.

SCHEMA
EVOLUTION
BASED-ON
CATEGORY THEORY

Formal work on schema evolution and data migration in an object-oriented setting have been presented, for example, by König et al. [71]. They employ category theory [11] to manipulate data models and migrate data according to fixed migration functors. In their work, they focus on information-preserving operations with fixed migration strategies. In contrast, we allow more flexibility in the specification of model migration but do not consider information-preservation.

Although schema evolution and meta-model evolution are related, there are also a number of differences between both challenges e.g. importance of technological challenges, depending artifacts, number of instances to be migrated, complexity of migration pattern, coupling and used constraints (see Chapter 2). Therefore, we consider related work on meta-model evolution and model migration next.

11.2 Meta-model Evolution

Meta-model evolution with model migration has been extensively studied. In particular, model migration has been in the focus of research, while meta-model changes have been mainly classified:

NON-BREAKING,
BREAKING AND
(UN)RESOLVABLE
META-MODEL
CHANGES

An introduction to the state-of-the-art of model co-evolution and especially a well-known classifications of co-evolutions has been presented in Chapter 2. Meta-model changes can be classified into *non-breaking*, *breaking and resolvable*, and *breaking and unresolvable* changes [51]. In Chapter 8, this classification is considered in the context of our work. In addition, we refine *breaking and resolvable* changes into *fully-automatically* and *automatically* resolvable changes. Furthermore, we define *semi-automatically* changes that need a manually or customarily defined migration specification. Van den Brand et al. [141] have a similar informal definition. Additional classifications are given in the following articles:

In [146], Wachsmuth presents an elaborated classification of meta-model relations according to preservation properties of concepts and meta-model instances. In this formal consideration, a meta-model is characterized by a set of concepts and a set of possible meta-model instances. A meta-model change is defined as e.g. *semantic-preserving* if its set of possible instances does not change. While Wachsmuth's classification is defined on a high abstraction level, our work is based on concrete modeling concepts formalized by a suitable category. It is up to future work to put the theory developed here into the context of this classification.

In [22], Ciccetti et al. distinguish *additive*, *subtractive* and *adaptive* meta-model changes and use these classes to describe the effect of a change to the set of valid instance models of a meta-model. While *additive* meta-model changes increase the total number of valid instance models, *subtractive* changes decrease the number of valid instance models, and *adaptive* changes describe changes such as *renaming* and *moving* of meta-model elements. Because our evolution rules clearly specify elements to be deleted ($tl(TL) \setminus tr(TR)$) and added ($TI \setminus tl(TL)$), additive as well as subtractive changes can be identified. Adaptive changes are simulated by combining deletion and addition.

Furthermore, meta-model changes can be presented either *structurally* or *operationally* [97]. While a structural presentation marks changed meta-model parts only, an *operational* presentation provides a possible sequence of meta-model evolution steps. Meta-model operations can be either *primitive* i.e. simply create, delete and update operations, or more *complex* ones describing for example meta-model refactorings [144] by composing simple operations. In our work, we consider evolution scenarios in an operational way by identifying a possible sequence of evolution steps manually or automatically (see Chapter 5).

Hermannsdorfer et al. [57] divide meta-model changes into meta-model only changes and coupled ones. While meta-model only changes can be considered as *non-breaking* changes, coupled changes are further split into three different groups according to their reusability: (1) *meta-model independent changes* require migration strategies that can be reused in various meta-models, (2) *meta-model specific* changes require migration strategies that can be applied to all instances of the meta-model, while (3) *model-specific changes* require user interactions during migration. We provide a heuristics for the construction of default migration schemes supporting the specification of meta-model independent migrations. Manually defined migration schemes or customized ones may be meta-model independent or dependent.

In the following, we move the focus from meta-model changes to the migration of their instance models. There are also a few works where researchers consider the migration of other dependent artifacts as well. These are model transformations [147], OCL constraints [53], and meta-

META-MODEL
CHANGES
CLASSIFIED BY
LANGUAGE
PROPERTIES

ADDITIVE,
SUBSTRUCTIVE AND
ADAPTIVE
META-MODEL
CHANGES

STRUCTURAL AND
OPERATIONAL
META-MODEL CHANGE
PRESENTATIONS

PRIMITIVE AND
COMPLEX
OPERATIONAL
CHANGES

META-MODEL
INDEPENDENT,
META-MODEL
SPECIFIC AND
MODEL-SPECIFIC
CHANGES

model hierarchies with more than two levels [5]. However, such work is out of the scope of this thesis. A taxonomy for the evolution of modeling languages discussing different scenarios including the evolution of meta-models and the migration of models and model transformations, can be found in [97]. We think our approach is promising to also serve as a solid basis to formally consider the migration of dependent modeling artifacts other than models.

In the following, we are asking several questions on model migration: Which formalizations of co-evolutions do exist and which properties are shown for model migrations? How far can model migration specifications be automatically deduced from given meta-model evolutions? Is it possible to reuse specifications for co-evolutions? Is it possible to customize deduced model migration specifications? What types of transformation approaches are used?

11.3 Correctness Properties of Model Migrations

While database schema evolution has been underpinned with formalization, approaches to meta-model evolution are usually not formally founded. Besides ours, we are aware of two approaches that consider the formal foundation of model migration due to meta-model evolution:

In [71], König et al. present a formal framework to data migration based on categorical constructions. It can also be used as formal basis for model migration. In contrast to our framework, their framework does not support reusable evolution operations yet and uses a fixed construction for migration, while we allow implementation of different migration strategies. Therefore, they have a correctness property supporting refactorings. König et al. consider transformation as correct if they do not loose data i.e. preserve paths in the instance graphs. Model migrations allow two operations: folding and unfolding of vertices. Vertices connected by edges can be merged (folded). Vertices can be split along loop edges¹ (unfolded) resulting only in connected vertices. Edges connecting vertices are never deleted. Hence, paths are preserved.

In [127], Sprinkle et al. propose a manual specification approach that has been implemented in the Model Change Language (MCL) [84]. MCL is presented as semi-formal approach. Model migration rules for unchanged types are not required to be defined explicitly, elements of such types are automatically copied. MCL rules are formalized by (span) DPO graph transformation rules consisting of injective morphisms only where the left rule morphism is the identity. Hence, the creation of model elements is only supported. Theorems are presented concerning termination and confluence of MCL transformations. MCL transformations always terminate

¹A loop edge of a graph G is an edge e with $src^G(e) = trg^G(e)$.

but are not always confluent. However, confluence is decidable in MCL. In our framework, we deduce each migration step from a meta-model evolution step, hence our migrations trivially terminate. In addition, the procedure to amalgamate coupled transformations in Chapter 7 yields confluent migration results².

Additionally, Krause et al. [73] consider well-typed model migration, however, only on the level of a prototypical implementation in the EMF model transformation tool Henshin [3]. In particular, they introduce a library to encode transformations of meta-models and instance models in the same rule. In our framework, rules are similarly coupled. In addition, they use the concept of rule amalgamation as we do. In their approach, meta-models and instance models are transferred into a presentation as single graphs including typing relationships. After a coupled transformation, meta-models and models are split. Because this is not practical, Krause et al. suggest splitting already evolution and migration rules before execution. However, this would require match-complete migration transformations as we suggest. Furthermore, they propose to derive customizable default migration rules from meta-model evolution rules as we support. In general, Krause et al. focus on the implementation while we focus on the theory. To some extent, the implementation in [73] can be considered as a first implementation of our framework covering only core concepts.

In [119], the authors evaluate the correctness of different migration approaches based on selected questions answered by the participants of the Transformation Tool Contest 2010. For each presented solution, the participants had to speculate about its correctness wrt. other test cases than the submitted ones. Unsurprisingly, none of the approaches scored high on this correctness property. In contrast, we formulate properties that can be used to define a basic form of correctness, such as match-completeness and executable migration definitions. Such properties allow an evaluation of correctness in a more precise manner.

Related to correctness of model migrations, a few researchers [97, 127] also distinguish *syntactical* and *semantical* model migrations. In [127], semantics is distinguished in static and dynamic semantics. *Static semantics* is given by well-formedness rules, while *dynamic semantics* is given by a mapping into a semantic domain. A syntactical migration is defined as migration transformation that produces syntactically valid instance models. Models of this type are at least well-typed. A semantical migration defines a model transformation that preserves also the meaning of the migrated instances given by the semantic mapping. In MDE, semantical model migrations are rarely studied. One obvious reason is that semantical mappings are usually not formally defined and given by code templates. In

SYNTACTICAL AND
SEMANTICAL
MODEL
MIGRATIONS

²As long as the default matching strategy is used.

our work, we ensure well-typed model migration and in addition we developed an approach to ensure the satisfaction of multiplicity constraints. This means our work focuses on syntactical model migration but we also consider aspects of static semantics.

11.4 Reuse of Migration Knowledge

Reuse of migration knowledge is a key idea of operator-based approaches. Usually, a set of coupled evolution-migration operators is supported, similarly to database schema evolution (see e.g. [25, 57]). However, researchers in MDE realized that a fixed set of reusable coupled evolution-migration operations is not enough for model co-evolution [57]. Therefore, current approaches allow to extend model migration specifications by manually written code using a general purpose programming language or a transformation language.

Reusable co-evolution operators for meta-model evolution have been first proposed by Wachsmuth [146], who tried to combine ideas from object-oriented refactorings with grammar adaptation. He also presents a prototypical implementation using QVT Relation [106].

An extensive catalog of reusable evolution operators being structured by different categories such as structural primitive operators, operators dealing with inheritance, and delegation, can be found in [58].

COPE/Edapt [57] is a meta-model evolution tool for EMF that allows the coupled evolution of meta-models and models by operators. Coupled operations are implemented according to a textual specification in Groovy/Java. The tool provides a rich library of coupled operators that can be applied if the required preconditions are satisfied. If an evolution operation is missing or the migration is not the desired one, the migration operation has to be implemented as Groovy/Java program. In this case, there is not any support to ensure well-defined migration results.

Vermolen et al. present in [143] a textual domain-specific transformation languages allowing the coupled evolution of WebDSL [145] models with data migration. Implementing the coupled operators has been supported by a generic approach based on textual grammars and XPath expressions, which may also be used in other domains. In contrast to this approach, we consider the co-evolution of meta-models and models based on graphs. Furthermore, in their approach, the migration transformations cannot be directly applied and have to be mapped to a suitable transformation language such as SQL for the case of WebDSL [145].

In [22, 45], complex meta-model evolution operations are detected for which migration operations have been defined. Specified migration operations are reusable. A tool EMFMigrate [147], picking up ideas from [22], is currently under development. EMFMigrate aims at also migrating other depending artifacts than models such as transformations written

in the Atlas Transformation Language (ATL) [6]. Migration rules can be assembled to reusable libraries as well as customized by overriding or refinement. However, to the best of our knowledge, there is no support to check that migration rules are defined consistently to their corresponding meta-model changes.

Rose et al. present their tool Epsilon Flock in [118]. It offers a manual specification approach for model migrations. Flock supports several modeling frameworks. In contrast to other approaches, textual migration scripts have to be written in Epsilon Flock. Reuse of migration knowledge is only supported by locally defined migration functions. In contrast to our formal framework, Epsilon Flock rules are not type checked. If a migration script is not valid according to the typing, elements may be forgotten without warning.

In this thesis, we propose coupled transformations that in particular fit into an operator-based approach. New coupled operators can be specified on the high abstraction level of algebraic graph transformations. Hence, we support the reuse of migration knowledge.

11.5 Deduction of Model Migration Specifications

Meta-model differencing and the automatic detecting of meta-model evolution steps have been considered by for example Ciccetti et al. in [22], and is also in the focus of the Atlas Matching Language (AML) [45, 117]. While AML [45] builds on matching heuristics to detect complex change operations, the approach presented in [22] focuses on the decomposition of difference models isolating dependent changes.

In [144], Sander et al. consider the detection of evolution steps. Starting from a structural change presentation, valid meta-model evolution traces are identified as sequences of primitive operations (based on operator preconditions and dependencies). Afterward, the approach is extended to detect complex operations from this sequences.

Similar approaches have also been developed e.g. to semantically lift edit scripts to higher levels. In [65, 66], Kelter et al. present an approach based on algebraic graph transformations. However, the approach presented by Kelter et al. differs from the approach presented in Chapter 5. In [65, 66], corresponding meta-model elements are related element-wise similar to Triple Graph Grammars [123]. Analyzed are the deltas between two model versions: all possible rule matches are detected first, then application sequences are calculated based-on dependency analyses.

Considering current modeling frameworks however, the detection of meta-model evolution steps is still a topic of ongoing research [65, 66, 118]. This means fully satisfying solutions are not yet available.

When pre-defined evolution operators cannot be used, model migrations are specified manually in most approaches. The automatic deduction

of migration specifications typically support the preservation and deletion of model elements:

In Epsilon Flock [118], rules for unchanged or slightly changed meta-model elements do not need to be defined similarly to MCL. Such model elements are automatically copied to a new model conforming to the new meta-model if they pass a conformance test.

In Chapter 5, a new approach for the stepwise detecting of evolution operations has been presented. The presented approach hereby naturally extends to the framework of coupled transformations presented thereafter. In contrast to other approaches, the approach can also be used if elements have been merged or split.

In addition to the detection of evolution steps, we also support the generation of default migration schemes (see Chapter 7). Default migration schemes migrate models according to a general heuristics also considering new element types. This has not been supported by others.

11.6 Customization of Model Migration Specifications

Customization of model migration specifications is supported in COPE/Edapt [57]. However, custom model migration operations have to be programmed in Groovy/Java and added to the tool.

In [22, 45], migration scripts are generated from recognized evolution operations using the textual model transformation language ATL. Customization of model migration scripts can be done on the level of such ATL scripts.

To summarize, for all informal approaches, there are no criteria to ensure that migration scripts remain well-defined after customization. Instead, our approach supports the generation of well-defined migration specification from meta-model evolutions, in form of default migration schemes. Moreover, they may be customized such that stay well-defined.

11.7 Employed Model Transformation Approaches

Model transformation approaches either transform models *in-place* by changing them or *out-place* i.e. by creating newly changed copies (see Chapter 2). For model migration, in-place and out-place approaches have been used. Especially in [119], several model and graph transformations tools have been applied to a migration case. Seven out of nine tools used out-place transformations. However, in-place transformations have an advantage: identity rules to copy unchanged elements are not required. Therefore, a variety of researchers argue for in-place transformations [56, 93, 98, 149]. In-place transformations, however, require that the migration transformation cover all model elements that need to be migrated to ensure well-typed migration results. Our framework abstracts

11.7. Employed Model Transformation Approaches

away from in-place and out-place transformations and can support both types of transformation approaches.

Each meta-model change usually requires many adaptations in each of its instances. These adaptations can be applied either simultaneously in *one step* or *incrementally* in several steps. In most cases, adaptations are applied incrementally, while we and [71, 73] apply each migration step in one transaction.

INCREMENTAL AND
SIMULTANEOUS
ADAPTATIONS

Conclusion and Future Work

Chapter 12 first provides a summary of this thesis. Then it concludes with an outlook on the current and future work.

12.1 Summary

The evolution of meta-models does not always lead to meaningful model migrations. In this thesis, we clarify the conditions for well-defined co-evolutions of meta-models and models by addressing different challenges on a formal level. An overview of addressed challenges and developed solutions is presented in Figure 12.1.

Contributions can be summarized as follows:

- First, we present a new approach for detecting evolution sequences based on algebraic graph transformations [36] that fit naturally into the theory of coupled transformations developed thereafter.
- Then, we formalize coupled transformations of meta-models and models as coupled algebraic graph transformations of type and instance graphs. This allows us to show that definitions and constructions presented in this thesis are well-defined based on results known from category theory [11] and algebraic graph transformations [36]. Herein, the presented theory is general enough that all types of current existing approaches supporting model migration can be formalized, i.e., manual specification, operator-based and matching approaches [118].

META-MODEL STEP
DETECTION

COUPLED
TRANSFORMATIONS

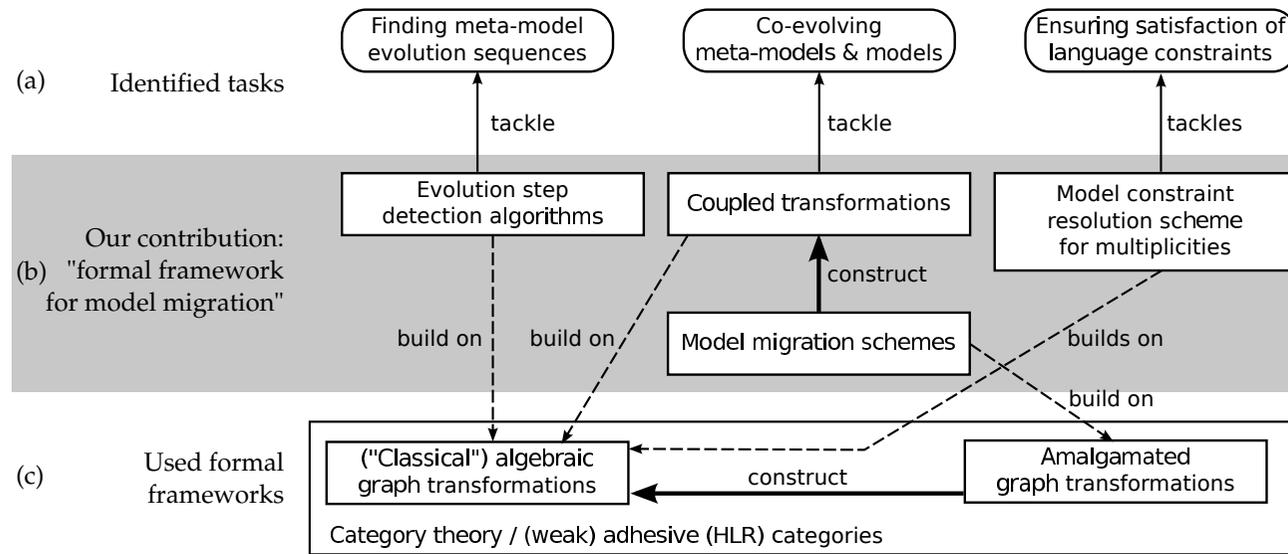


Figure 12.1: Overview of thesis results

-
- Furthermore, we define migration schemes by adapting amalgamated graph transformation and discuss how model migrations can be specified based on migration patterns. Migration schemes herein complete evolution steps given by graph transformations canonical to coupled transformations. By Theorem 6.4.1 and Theorem 7.2.1, we show that this is ensured. We also show how migration schemes may be customized while still ensuring completeness of the migration process and well-typedness of migration results. In addition, we introduce default migration schemes that are automatically deduced from meta-model evolution rules by a heuristic.

MIGRATION
SCHEMES
 - While we explain our approach for (directed multi-)graphs first, its application to more elaborated graph structures supporting type inheritance and multiplicity constraints is discussed thereafter. Because our work is based on (weak) adhesive High-Level-Replacement (HLR) categories [36, 77], a variety of high-level structures are covered. Furthermore, the presented approach is discussed in the context of non-breaking changes, breaking and resolvable changes [51], probably the most used classification of meta-model changes in the literature.

HIGH-LEVEL
STRUCTURES
 - In addition, we present a new approach to deal with multiplicity constraints after models have been migrated to well-typed models by coupled transformations. The approach is based on an approach originally developed for generating finite instance models [134] and ensures that all multiplicity constraint violations are corrected.

MULTIPLICITY
CONSTRAINTS
 - Finally, we evaluate our approach on a case study that has been contributed by the Transformations Tool Contest 2010 [119].

CASE STUDY FROM
TOOL CONTEST

While meta-model evolution induces many challenges, we have covered a subset of such challenges on a formal level. We think that the developed theory has a promising potential to improve the current state-of-the-art in model migration in MDE.

12.2 Outlook

Open points that should be addressed in the future concern primarily tool support and the migration of other artifacts than models:

- In addition to models, other artifacts such as model transformations, visual model presentations and editors need to be adapted when meta-models are evolved. It would be an interesting research line to extend our work to consider also such artifacts (see Chapter 2). In particular, it would be beneficial to have one framework that puts such research under a common umbrella to develop technologies that smoothly work together.

OTHER ARTIFACTS

- TOOL SUPPORT
 - Up to now the proposed approach has already been (partly) implemented in Scala. To evaluate the contributed work in practice it would be beneficial to finish this work. While the prototypical implementation is working with a modeling framework that has been implemented for this purpose based on categories presented in this thesis, it would be interesting to transfer this work to existing modeling frameworks such as EMF. Therefore, in particular, additional tasks have to be considered such as missing element identifiers and intrinsic modeling language constraints implemented in such frameworks. We neglected such problems so far to be able to build theory.
- META-MODEL EVOLUTION STEP DETECTION
 - The algorithm to detect sequences of meta-model evolution steps presented in Chapter 5 needs to be extended to be applicable if meta-model elements are only transient. To put it in a nutshell, as in many other approaches, it is not possible yet to detect operation steps that create model elements that are deleted by a following step. In addition, we assume that it does not matter which evolution step is detected first. Otherwise, backtracking may be helpful if this is not the case and our approach using rule priorities is not sufficient.
- AMALGAMATED GRAPH TRANSFORMATIONS
 - Amalgamated graph transformations [17, 133, 135] are useful not only for model migration but also for meta-model evolution, for example, to express evolution steps such as “Pull up Attribute” for arbitrary numbers of subclasses. While such amalgamated graph transformations also fit naturally into our framework, as they construct “usual” graph transformations, not all details have been formulated wrt. model migration schemes. However, we assume that such an extension is straight-forward.
- OTHER CONSTRAINTS
 - Furthermore, future work is needed to extend our model co-evolution approach wrt. meta-model constraints other than multiplicities. While it has been discussed how constraints such as *abstract* can be handled by a proper retyping of elements, in particular *containment* constraints should be considered. In addition, Assumption 9.3.1 has to be shown since finite satisfiability has been studied in [18, 83] without reference refinement.
- MORE CASE STUDIES
 - Moreover, further co-evolution case studies are needed to evaluate the heuristic used to derive default migration schemes as well as their customization facilities.

Appendices

Proofs of Auxiliary Propositions

Proofs in this thesis base on properties of (weak) adhesive (HLR) categories that have been listed in Chapter 4. These properties are well-known facts. However, in two cases we need to generalize existing results. The proofs for these generalizations are presented in this appendix.

A.1 Generalizing the Special Pullback-Pushout Property

Property 9 in Table 4.2 is taken from [78] and generalized to adhesive HLR categories with arbitrary pullbacks in Proposition A.1.1. Property 11 in Table 4.2 can be deduced as corollary from the proof of Proposition A.1.1 below. In the following, we explicitly show the proof of Property 9 to argue that adhesive HLR categories with arbitrary pullbacks are sufficient. In contrast to adhesive categories, adhesive HLR categories ensure the existence of pullbacks along \mathbf{M} -morphisms only. The proof of this proposition, however, demands the existence of arbitrary pullbacks. While some steps in the proof in [78] are only sketched, we show the fully elaborated proof here. First, we recall Property 9 from [78] generalized to adhesive HLR categories:

Proposition A.1.1 (Special pullback-pushout property).

Suppose the commutative diagram to the right in an adhesive HLR category with general pullbacks has \mathbf{M} -morphisms m, n, l : Suppose that square (1) is a pushout and square (1+2) is a pullback, then square (2) is a pullback.

$$\begin{array}{ccccc}
 A & \xrightarrow{f} & C & \xrightarrow{p} & E \\
 \downarrow m & & \downarrow n & & \downarrow l \\
 B & \xrightarrow{g} & D & \xrightarrow{q} & F
 \end{array}$$

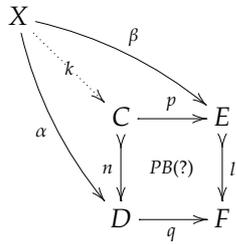
(1) (2)

SPECIAL
PULLBACK-PUSHOUT
PROPERTY

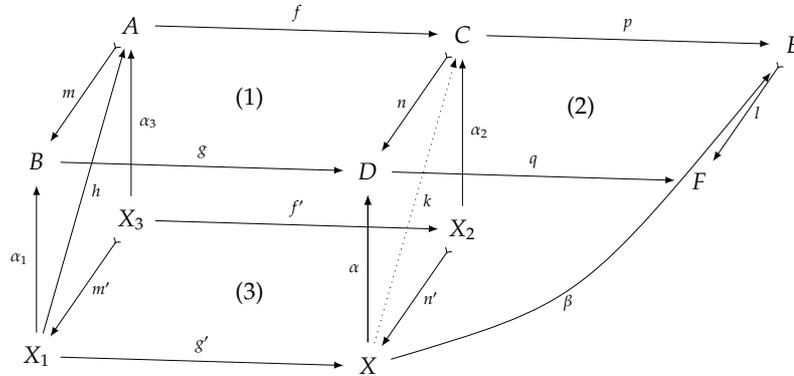
In contrast to the usual pullback decomposition saying that, if (1+2) and (2) are pullbacks, then (1) is also a pullback, we can decompose pullback

(1+2) by pullback (1) and can deduce that (2) is a pullback given that morphism $m : A \rightarrow B$ and $l : E \rightarrow F$ are \mathbf{M} -morphisms (see Definition A.1.1) and pullback (1) is also a pushout¹. While Property 9 applies to all adhesive (HLR) categories with arbitrary pullbacks, it does not generally apply to weak adhesive HLR categories.

Proof of Proposition A.1.1. We have to show that (2) in the right diagram of Proposition A.1.1 is a pullback. Suppose that we have an object X and morphisms $\alpha : X \rightarrow D$ and $\beta : X \rightarrow E$ such that $\alpha; q = \beta; l$ as in the diagram below. We shall show that there exists a morphism $k : X \rightarrow C$ such that $k; n = \alpha$ and $k; p = \beta$. Note that it is sufficient to show that such a k exists because uniqueness follows by n given as monomorphism.



In the following, we construct the diagram below. The upper faces with (1) pushout and (1+2) pullback are already given. Furthermore, we assume an object X with morphisms α and β as given in the diagram above.



1. Construct the front face of the cube (on the left of the figure above) by taking the pullback $B \xleftarrow{\alpha_1} X_1 \xrightarrow{g'} X$ of co-span $B \xrightarrow{g} D \xleftarrow{\alpha} X$. Note that this step requires pullbacks along arbitrary morphisms, as g does not need to be a \mathbf{M} -morphism.

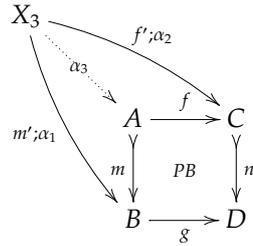
¹Note that it follows by Property 6 that morphism $n : C \rightarrow D$ is a \mathbf{M} -morphisms. By Property 4, we have that (1) is also a pullback since it is a pushout.

A.1. Generalizing the Special Pullback-Pushout Property

2. Construct the right face of the cube by taking a pullback $X \xleftarrow{n'} X_2 \xrightarrow{\alpha_3} C$ of co-span $X \xrightarrow{\alpha} D \xleftarrow{n} C$. Note that morphism n' is a **M**-morphism, as **M**-morphisms are stable under pullback and n is a **M**-morphism (Property 5).
3. Construct the bottom face of the cube by taking the pullback $X_1 \xleftarrow{m'} X_3 \xrightarrow{f'} X_2$ of co-span $X_1 \xrightarrow{g'} X \xleftarrow{n'} X_2$. Note that morphism m' is a **M**-morphism, as **M**-morphisms are stable under pullback and n' is a **M**-morphism (Property 5).
4. Because (1) is a pushout along a **M**-morphism, it is also a pullback (Property 4).

$$\begin{array}{l}
 m';\alpha_1;g = m';g';\alpha \\
 = f';n';\alpha \\
 = f';\alpha_2;n
 \end{array} \left| \begin{array}{l}
 \text{front face commutes (see Step 1)} \\
 \text{bottom face commutes (see Step 3)} \\
 \text{right face commutes (see Step 2)}
 \end{array} \right.$$

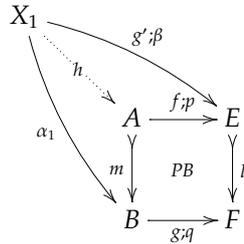
Hence, the outer square in the diagram below commutes and we can deduce the existence of a unique mediating morphism $\alpha_3 : X_3 \rightarrow A$ with $m';\alpha_1 = \alpha_3;m$ and $f';\alpha_2 = \alpha_3;f$.



5. Diagram (1+2) is a pullback by assumption, therefore $f;p;l = m;g;q$.

$$\begin{array}{l}
 \alpha_1;g;q = g';\alpha;q \\
 = g';\beta;l
 \end{array} \left| \begin{array}{l}
 \text{front face pullback} \\
 \alpha;q = \beta;l \text{ was assumption}
 \end{array} \right.$$

Hence, the outer square in the diagram below commutes and we can deduce the existence of a unique mediating morphism $h : X_1 \rightarrow A$ such that $h;m = \alpha_1$ and $g';\beta = h;f;p$.



6. The bottom face of the cube is also a pushout, due to the Van Kampen property.

a) *The back face is a pullback:* the bottom (3) and front face (4) of the cube can be composed to pullback (3+4). Because the cube commutes, (3+4) can be decomposed by the top face pullback (1). Hence, the back face (5) is a pullback.

$$\begin{array}{ccc}
 B & \xrightarrow{g} & D \\
 \alpha_1 \uparrow & (4) & \uparrow \alpha \\
 X_1 & \xrightarrow{g'} & X \\
 m' \uparrow & (3) & \uparrow n' \\
 X_3 & \xrightarrow{f'} & X_2
 \end{array}
 \qquad
 \begin{array}{ccc}
 B & \xrightarrow{g} & D \\
 m \uparrow & (1) & \uparrow n \\
 A & \xrightarrow{f} & C \\
 \alpha_3 \uparrow & (5) & \uparrow \alpha_2 \\
 X_3 & \xrightarrow{f'} & X_2
 \end{array}$$

b) *The left face is a pullback:* the bottom (3) and right faces (6) of the cube can be composed to pullback (3+6). Because the cube commutes, (3+6) can be decomposed by the top face pullback (1). Hence, the left face (7) is a pullback.

$$\begin{array}{ccc}
 X_3 & \xrightarrow{f'} & X_2 & \xrightarrow{\alpha_2} & C \\
 m' \downarrow & (3) & \downarrow n' & (6) & \downarrow n \\
 X_1 & \xrightarrow{g'} & X & \xrightarrow{\alpha} & D
 \end{array}
 \qquad
 \begin{array}{ccc}
 X_3 & \xrightarrow{\alpha_3} & A & \xrightarrow{f} & C \\
 m' \downarrow & (7) & \downarrow m & (1) & \downarrow n \\
 X_1 & \xrightarrow{\alpha_1} & B & \xrightarrow{g} & D
 \end{array}$$

c) *Now we can apply the Van Kampen property:* the cube commutes, the left and the back faces are pullbacks and the top face is a pushout along a **M**-morphism by assumption. Hence, the bottom face is also a pushout.

7. Because the bottom face (3) is a pushout, we get the existence of morphism k .

First, we need a helper equation:

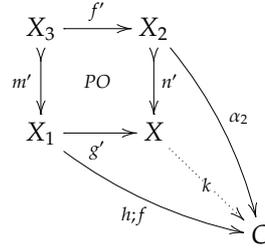
$$\begin{array}{l}
 \alpha_3; m = m'; \alpha_1 \\
 = m'; h; m \\
 \alpha_3 = m'; h
 \end{array}
 \left| \begin{array}{l}
 \text{left top face PB (1); (see Step 4)} \\
 \alpha_1 = h; m \text{ top faces PB (1+2);} \\
 \text{(see Step 5)} \\
 m \text{ is mono, (*)}
 \end{array} \right.$$

Now, we can use the bottom pushout.

$$\begin{array}{l}
 f'; \alpha_2 = \alpha_3; f \\
 = m'; h; f
 \end{array}
 \left| \begin{array}{l}
 \text{back face commutes (Step 4)} \\
 \text{helper equation (*) above}
 \end{array} \right.$$

This means that the outer square commutes.

A.1. Generalizing the Special Pullback-Pushout Property



By this pushout, we can deduce the existence of a unique morphism $k : X \rightarrow C$ with $g';k = h;f$ and $n';k = \alpha_2$.

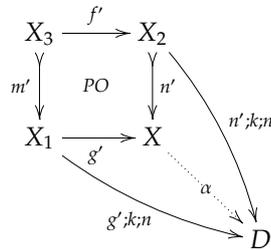
8. It remains to show that k satisfies the necessary properties.

a) We have to show that $k;n = \alpha$.

$$\begin{array}{lcl}
 g';k;n & = & h;f;n \\
 & = & h;m;g \\
 & = & \alpha_1;g \\
 & = & g';\alpha
 \end{array}
 \left| \begin{array}{l}
 \text{bottom PO (see Step 7)} \\
 \text{diagram (1) commutes by assumption} \\
 \alpha_1 = h;m \text{ top faces PB (1+2);} \\
 \text{(see Step 5)} \\
 \text{front face commutes (Step 1)}
 \end{array} \right.$$

In addition we have:

$$\begin{array}{lcl}
 n';k;n & = & \alpha_2;n \\
 & = & n';\alpha
 \end{array}
 \left| \begin{array}{l}
 \text{bottom PO (see Step 7)} \\
 \text{right face commutes (see Step 2)}
 \end{array} \right.$$



By the pushout in the bottom face and g' and n' being jointly epi, we have in fact $k;n = \alpha$.

b) Furthermore, we have to show that $k;p = \beta$.

Again, we need a helper equation:

$$\begin{array}{lcl}
 \alpha_2;p;l & = & \alpha_2;n;q \\
 & = & n';\alpha;q \\
 & = & n';\beta;l
 \end{array}
 \left| \begin{array}{l}
 \text{right top face commutes (assumption)} \\
 \text{right face commutes (see Step 2)} \\
 \text{assumption of diagram (2)}
 \end{array} \right.$$

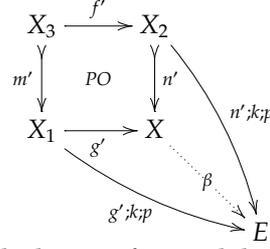
$$\alpha_2;p = n';\beta \quad \left| \quad l \text{ is mono, (**)} \right.$$

Similar to 8.(a) we get:

$$\begin{array}{l} g';k;p \\ = \\ = g';\beta \end{array} \left| \begin{array}{l} \text{bottom PO (see Step 7)} \\ \text{top faces PB (1+2); (see Step 5)} \end{array} \right.$$

And in addition we have:

$$\begin{array}{l} n';k;p \\ = \\ = n';\beta \end{array} \left| \begin{array}{l} \text{bottom PO (see Step 7)} \\ \text{helper equation (**) above} \end{array} \right.$$



By the pushout in the bottom face and the uniqueness of mediating morphisms, we get $k;p = \beta$.

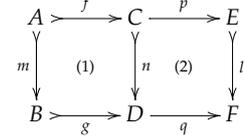
□

All steps in the presented proof are valid for adhesive HLR categories with pullbacks. Except Step 6(c), all steps are also valid in weak adhesive categories. In Step 6(c), we cannot apply the Van Kampen property because the pushout in the top face of the cube is required to be a pushout along two \mathbf{M} -morphisms. By assuming morphism f to be a \mathbf{M} -morphism as well, we get the following corollary for weak adhesive categories:

WEAK SPECIAL
PULLBACK-PUSHOUT

Corollary A.1.1 (Weak special pullback-pushout property).

Let the commutative diagram to the right be given in a weak adhesive HLR category with \mathbf{M} -morphisms m, n, l, f and g : If square (1) is a pushout and square (1+2) is a pullback, then square (2) is a pullback.



Note that g is a \mathbf{M} -morphism, as f is a \mathbf{M} -morphism (Property 6). Therefore, the existence of pullbacks along \mathbf{M} -morphisms is sufficient in Step 1 of the proof and in the corollary.

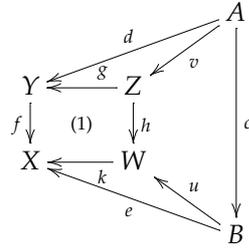
A.2 On the Stability of Final Pullback Complements

In the thesis, we also make use of Final Pullback Complements (FPBCs), which are essential for a specific type of graph transformation approach, called sesqui pushout approach [24]. In particular, we need a new theorem, Property 12 in Table 4.2, which is presented and proven in this section. While we consider pushouts and pullbacks as well known, we define Final Pullback Complements next:

A.2. On the Stability of Final Pullback Complements

Definition A.2.1 (Final pullback complement). A final pullback complement of $Z \xrightarrow{g} Y \xrightarrow{f} X$ is defined by a composition of morphisms $Z \xrightarrow{h} W \xrightarrow{k} X$ (see figure on the right) where:

1. square (1) is a pullback and
2. for each pullback $d; f = c; e$ and for each $v : A \rightarrow Z$ with $v; g = d$, there exists a unique $u : B \rightarrow W$ such that $u; k = e$ and $v; h = c; u$.

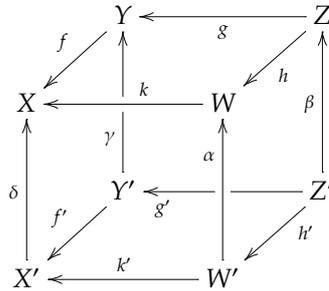


FINAL PULLBACK COMPLEMENT

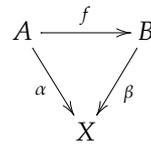
The following theorem has been proven in special categories by Loewe in [88]. In contrast to that, we provide an elementary proof that is true in any category with pullbacks. The proof has been found in collaboration with Harald König from FHDW Hannover and has not been published before.

Theorem A.2.1 (Final-Pullback-Complements are stable under pullbacks). In any category \mathbf{C} with pullbacks, Final-Pullback-Complements (FPBCs) are stable under pullback, i.e. if $Z \xrightarrow{h} W \xrightarrow{k} X$ in the top face in the figure below is a FPBC and all side faces are pullbacks, then $Z' \xrightarrow{h'} W' \xrightarrow{k'} X'$ in the bottom square is a FPBC too.

FINAL-PULLBACK-COMPLEMENTS ARE STABLE UNDER PULLBACKS



In the following, we denote by \mathbf{C}/X the slice category whose objects are morphisms $\alpha : A \rightarrow X$ with codomain X and whose morphisms $f : \alpha \rightarrow \beta$ are morphisms $f : A \rightarrow B$ in \mathbf{C} such that $f; \beta = \alpha$. Furthermore, we denote by $Hom_{\mathbf{D}}(A, B)$ the set of morphisms from A to B in some category \mathbf{D} .



In the proof below, we use well known facts about the pullback functor [11]:

A. PROOFS OF AUXILIARY PROPOSITIONS

Fact A.2.1 (Pullback functor). *Let $\delta : X' \rightarrow X$ be any morphism of category \mathbf{C} with pullbacks. Constructing the pullback (more precisely: a fixed choice of it) of a morphism $k : W \rightarrow X$ along δ yields a mapping*

$$\delta^* : \mathbf{C}/X \rightarrow \mathbf{C}/X' \tag{A.1}$$

that extends to a functor, called the pullback functor, by mapping morphisms to unique mediators between pullbacks.

$$\begin{array}{ccc} X & \xleftarrow{k} & W \\ \delta \uparrow & \text{PB} & \uparrow \alpha \\ X' & \xleftarrow{\delta^*(k)} & W' \end{array}$$

Additionally, there is the post-composing functor

$$\delta_* : \mathbf{C}/X' \rightarrow \mathbf{C}/X \text{ with } \delta_*(b) = b; \delta \tag{A.2}$$

for all objects $b : B \rightarrow X'$ of \mathbf{C}/X' the object $b; \delta$ of \mathbf{C}/X .

It is a well-known fact that $\delta_* \dashv \delta^*$, i.e. δ_* is left-adjoint to δ^* , which means that there is a family

$$(\eta_b : b \rightarrow \delta^*(\delta_*(b)))_{b \in \text{Ob } \mathbf{C}/X'} \tag{A.3}$$

of morphisms indexed over the objects of \mathbf{C}/X' (the unit of the adjunction), such that in the diagram below

$$\begin{array}{ccc} X & \xleftarrow{\delta_*(b)} & B \\ \delta \uparrow & & \parallel \\ X' & \xleftarrow{b} & B \\ & \searrow \eta_b & \downarrow \pi_2 \\ & & X' \times_X B \\ & \swarrow \delta^*(\delta_*(b)) & \nearrow \end{array}$$

in which the outer square is the pullback of $X' \xrightarrow{\delta} X \xleftarrow{\delta_*(b)} B$ and $b; \delta = \delta_*(b)$, the following statements hold:

1. We have:

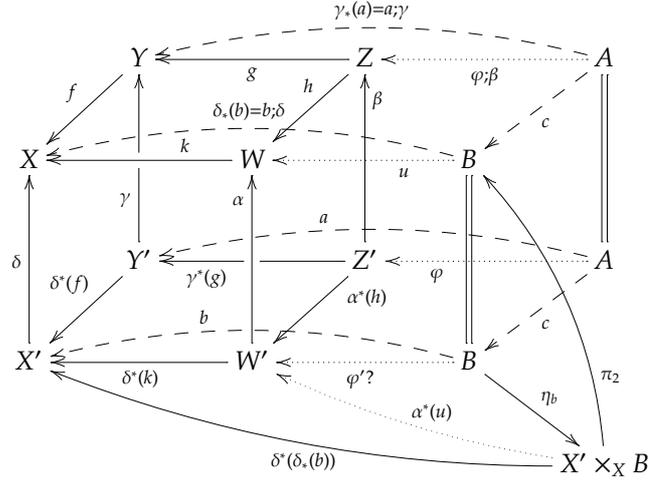
$$\eta_b; \pi_2 = id_B. \tag{A.4}$$

2. Moreover, we have for each fixed $b \in \mathbf{C}/X'$ and $k \in \mathbf{C}/X$ the assignments $u \mapsto \eta_b; \delta^*(u)$ for all $u \in \text{Hom}_{\mathbf{C}/X}(\delta_*(b), k)$ defining a bijective

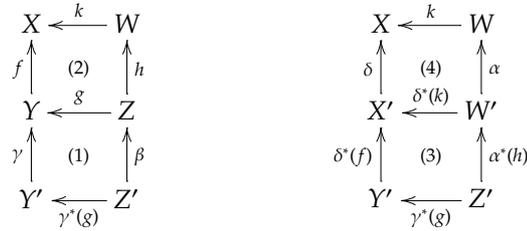
$$i_{b,k} : \text{Hom}_{\mathbf{C}/X}(\delta_*(b), k) \rightarrow \text{Hom}_{\mathbf{C}/X'}(b, \delta^*(k)) \tag{A.5}$$

A.2. On the Stability of Final Pullback Complements

Proof of Theorem A.2.1. Suppose the commuting double cube below with $Z \xrightarrow{h} W \xrightarrow{k} X$ is a FPBC (left top face) and all side faces of the left cube pullbacks. In the following, it shall be shown that $Z' \xrightarrow{\alpha^*(h)} W' \xrightarrow{\delta^*(k)} X'$ is also a FPBC.



1. First we show that the left bottom face is a pullback: by composing the left back face (1) and the left top face (2) of the cube to a pullback (1+2), and by decomposing (1+2) by the left front face (4), we can deduce the left bottom face (3) is a pullback.



2. Existence of mediating morphism: let $Y' \xleftarrow{a} A \xrightarrow{c} B$ in the bottom of the double cube above (the dashed lines) be a pullback of $B \xrightarrow{b} X' \xleftarrow{\delta^*(f)} Y'$. We have to show that for each $\varphi : A \rightarrow Z'$ with $\varphi; \gamma^*(g) = a$, there is a unique

$$\varphi' : B \rightarrow W'$$

such that:

$$\varphi'; \delta^*(k) = b \text{ and } c; \varphi' = \varphi; \alpha^*(h) \tag{A.6}$$

A. PROOFS OF AUXILIARY PROPOSITIONS

For this we consider the composition of this pullback with the left face of the cube (this establishes the pullback shown as the top dashed rectangle in the double cube above).

We obtain

$$\begin{array}{l} \varphi; \beta; g \\ = \\ = a; \gamma \end{array} \quad \left| \begin{array}{l} \text{left back face commutes} \\ \text{by assumption that } \varphi : a \rightarrow \gamma^*(g) \\ \text{i.e. } a = \varphi; \gamma^*(g) \end{array} \right.$$

such that by the FPBC property of the top square there is a *unique* u with:

$$\delta_*(b) = u; k \text{ and } \varphi; \beta; h = c; u \quad (\text{A.7})$$

Thus $u \in \text{Hom}_{\mathbf{C}/X}(\delta_*(b), k)$ and we can choose $\varphi' := i_{b,k}(u) \in \text{Hom}_{\mathbf{C}/X'}(b, \delta^*(k))$, i.e.

$$\varphi'; \delta^*(k) = b \text{ and } \varphi' = \eta_b; \delta^*(u) \quad (\text{A.8})$$

by (A.5). It remains to show validity of the second equation in (A.6). For this we construct the pullback of $X' \xrightarrow{\delta} X \xleftarrow{\delta_*(b)} B$ yielding the span $X' \times_X B \xleftarrow{\delta^*(\delta_*(b))} X' \xrightarrow{\pi_2} B$ (see cube above and diagram below).

$$\begin{array}{ccc} X & \xleftarrow{\delta_*(b)} & B \\ \delta \uparrow & \text{PB} & \uparrow \pi_2 \\ X' & \xleftarrow{\delta^*(\delta_*(b))} & X' \times_X B \end{array}$$

We have:

$$\begin{array}{l} \varphi'; \alpha \\ = \\ = \eta_b; \pi_2; u \\ = u \end{array} \quad \left| \begin{array}{l} \text{by (A.8)} \\ \text{by (A.1)} \\ \text{by (A.4)} \end{array} \right.$$

and thus:

$$\begin{array}{l} c; u \\ = \\ = \varphi; \beta; h \\ = \varphi; \alpha^*(h); \alpha \end{array} \quad \left| \begin{array}{l} \text{(above)} \\ \text{by (A.7)} \\ \text{by (right face of left cube commutes)} \end{array} \right.$$

A.2. On the Stability of Final Pullback Complements

Moreover, we have:

$$\begin{array}{lcl}
 c; \varphi'; \delta^*(k) = & c; b & \left| \begin{array}{l} \text{by (A.8)} \\ \text{dashed bottom PB commutes} \end{array} \right. \\
 = & a; \delta^*(f) & \\
 = & \varphi; \gamma^*(g); \delta^*(f) & \left| \begin{array}{l} \text{by assumption that } \varphi : a \rightarrow \gamma^*(g) \\ \text{i.e. } a = \varphi; \gamma^*(g) \end{array} \right. \\
 = & \varphi; \alpha^*(h); \delta^*(k) & \left| \begin{array}{l} \text{left bottom face commutes} \end{array} \right.
 \end{array}$$

Because, in the front face pullback α and $\delta^*(k)$ are jointly monic, the last two equations, i.e.

$$\begin{array}{lcl}
 c; \varphi'; \alpha & = & \varphi; \alpha^*(h); \alpha \quad \text{and} \\
 c; \varphi'; \delta^*(k) & = & \varphi; \alpha^*(h); \delta^*(k)
 \end{array}$$

yield $c; \varphi' = \varphi; \alpha^*(h)$ i.e. the second equation of (A.6) as required.

3. *Uniqueness of mediating morphism:* in order to show uniqueness of φ' , let any other $\varphi'' \in \text{Hom}_{\mathbf{C}_{/X'}}(b, \delta^*(k))$ with $c; \varphi'' = \varphi; \alpha^*(h)$ be given.

Then, for $u'' := i_{b,k}^{-1}(\varphi'')$ one can show in the same way as above that $\varphi''; \alpha = u''$. This yields $c; u'' = \varphi; \alpha^*(h); \alpha = c; u$.

Thus (by uniqueness of u with these properties) $u = u''$. Because $i_{b,k}$ is bijective, $\varphi' = \varphi''$.

□

Case Study: Adhesiveness

The theory developed in this thesis relies heavily on properties that are only valid in adhesive-like categories. Therefore, a good question is which categories can be expected to be adhesive. In Chapter 3, the Diagram Predicate Framework has been illustrated by Example 3.3.3 as a flexible method to equip directed multi-graphs with constraints. This chapter is based on technical report [152]. First, it is shown that even category `Graph` is adhesive, category `Spec`, the category of DPF specifications, is not. Then DPF is generalized so that we get an adhesive category `GSpec`. To provide the reader with a good intuition, this generalization is presented analogue to the step from simple to multi-graphs. Herein, simple graphs are graphs that do not allow more than one edge between a source and target vertex. It is well-known that also the category of such simple graphs is not adhesive [63].

B.1 Categories of Simple Directed Graphs

The category `Set` of sets and total mappings is a well-know adhesive category. The category of (simple) directed graphs `SGraph` extends the category of sets by a second component called “edges,” that consists of a set of ordered pairs of vertices (*source, target*). In the following, the definition of directed graphs are recalled. Note, that we call these graphs simple graphs in this chapter to distinguish them from multi-graphs.

Definition B.1.1 (Simple (directed) graph). A simple (directed) graph $G = (G_V, G_E)$ consists of a set G_V of vertices (or nodes) and a set $G_E \subseteq G_V \times G_V$ of edges (or arrows) where each edge is an ordered pair of vertices $(x, y) \in G_V \times G_V$. The source, respectively target of the edge is denoted by

SIMPLE (DIRECTED)
GRAPH

projection to the first ($src^G(x, y) = x$), respectively second ($trg^G(x, y) = y$) component of this pair.

GRAPH MORPHISM
IN SIMPLE
(DIRECTED) GRAPHS

Definition B.1.2 (Graph morphism between simple (directed) graphs). A graph morphism $\phi : G \rightarrow H$ is a mapping $\phi_V : G_V \rightarrow H_V$ from the vertex set of G to the vertex set of H such that each edge $(x, y) \in G_E$ entails an edge $(\phi_V \times \phi_V)(x, y) := (\phi_V(x), \phi_V(y)) \in H_E$.

$$\begin{array}{ccc} G_V \times G_V & \xrightarrow{\phi_V \times \phi_V} & H_V \times H_V \\ \uparrow & & \uparrow \\ G_E & \xrightarrow{\phi_E} & H_E \end{array}$$

Remark B.1.1 (Mapping of edges). The requirement $(\phi_V \times \phi_V)(G_E) \subseteq H_E$ means equivalently that the mapping $\phi_V \times \phi_V$ restricts to a mapping from G_E to H_E . We will denote this mapping by $\phi_E : G_E \rightarrow H_E$.

CATEGORY
SGraph

Definition B.1.3 (Category of simple (directed) graphs). The category **SGraph** has all simple (directed) graphs G as objects and all graph morphisms $\phi : G \rightarrow H$ as morphisms between graphs G and H . The associativity of composition of mappings ensures that the composition of two graph morphisms is a graph morphism as well and that the composition of graph morphisms is associative. Moreover, the identity mappings $id^{G_V} : G_V \rightarrow G_V$ define identity graph morphisms $id^G : G \rightarrow G$ and ensure that identity graph morphisms are left and right neutral with respect to composition.

MONOMORPHISM
IN SIMPLE
(DIRECTED) GRAPHS

Fact B.1.1 (Monomorphism in simple (directed) graphs). *The monomorphisms in **Set** are exactly the injective mappings. In **SGraph**, monomorphisms are the morphisms where the vertex mapping is injective. Note, that also in this case the induced mapping of edges is injective.*

B.1.1 Pushouts/Pullbacks of Simple Directed Graphs

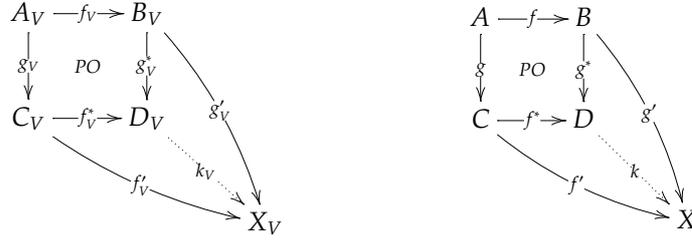
Let us consider the general construction of pushouts and pullbacks in **SGraph** relying on the construction of pushouts and pullbacks in **Set**. This restricts the task primarily to the construction of edges for simple graphs.

PUSHOUT FOR
SIMPLE (DIRECTED)
GRAPHS

Proposition B.1.1 (Pushout for simple (directed) graphs). *A pushout $B \xrightarrow{g} D \xleftarrow{f} C$ of a span $B \xleftarrow{f} A \xrightarrow{g} C$ of graph morphisms is obtained by constructing first a pushout $B_V \xrightarrow{g_V^*} D_V \xleftarrow{f_V^*} C_V$ in **Set** of the underlying span $B_V \xleftarrow{f_V} A_V \xrightarrow{g_V} C_V$ of mappings between sets of vertices and by defining the set of edges D_E as follows:*

$$D_E := (g_V^* \times g_V^*)(B_E) \cup (f_V^* \times f_V^*)(C_E) = \{(g_V^*(x), g_V^*(y)) \mid (x, y) \in B_E\} \cup \{(f_V^*(x), f_V^*(y)) \mid (x, y) \in C_E\} \subseteq D_V \times D_V$$

B.1. Categories of Simple Directed Graphs



Proof.

Homomorphism property: we have $(g_V^* \times g_V^*)(B_E) \subseteq D_E$ and $(f_V^* \times f_V^*)(C_E) \subseteq D_E$, by construction, thus the mappings g_V^* and f_V^* constitute graph morphisms $g^* : B \rightarrow D$ and $f^* : C \rightarrow D$, respectively.

Universal property:

1. There exists for all graph morphisms $g' : B \rightarrow X$ and $f' : C \rightarrow X$ with $g; f' = f; g'$, i. e. $g_V; f'_V = f_V; g'_V$, a unique mapping $k_V : D_V \rightarrow X_V$ with $f_V^*; k_V = f'_V$ and $g_V^*; k_V = g'_V$.
2. Because f^* and f' are graph morphisms, we have

$$\begin{aligned}
 (k_V \times k_V)((f_V^* \times f_V^*)(C_E)) &= (f_V^* \times f_V^*); (k_V \times k_V)(C_E) \\
 &= (f_V^*; k_V \times f_V^*; k_V)(C_E) \\
 &= (f_V^* \times f'_V)(C_E) \subseteq X_E
 \end{aligned}$$

and, analogously, we have $(k_V \times k_V)((g_V^* \times g_V^*)(B_E)) \subseteq X_E$. Due to the construction of D_E , this ensures $(k_V \times k_V)(D_E) \subseteq X_E$, thus k_V establishes a graph morphism $k : D \rightarrow X$.

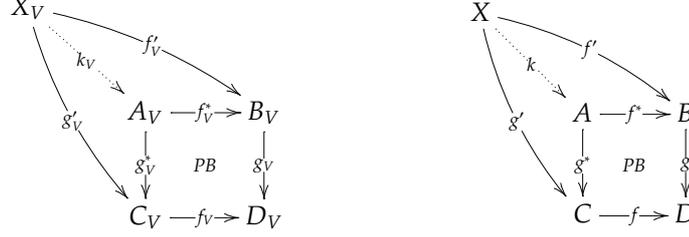
□

Besides pushouts, pullbacks in SGraph are also based on the corresponding construction in Set .

Proposition B.1.2 (Pullback for simple (directed) graphs). *A pullback $B \xleftarrow{f^*} A \xrightarrow{g^*} C$ of a cospan $B \xrightarrow{g} D \xleftarrow{f} C$ is obtained by constructing first a pullback $B_V \xleftarrow{f_V^*} A_V \xrightarrow{g_V^*} C_V$ in Set of the underlying cospan $B_V \xrightarrow{g_V} D_V \xleftarrow{f_V} C_V$ of mappings between sets of vertices and by defining the set of edges A_E as follows:*

$$\begin{aligned}
 A_E := \{e \in A_V \times A_V \mid (f_V^* \times f_V^*)(e) \in B_E, (g_V^* \times g_V^*)(e) \in C_E, \\
 g_E((f_V^* \times f_V^*)(e)) = f_E((g_V^* \times g_V^*)(e))\}
 \end{aligned}$$

PULLBACK FOR
SIMPLE (DIRECTED)
GRAPHS



Proof.

Homomorphism property: by construction we have $(f'_V \times f_V)(A_E) \subseteq B_E$ and $(g'_V \times g_V)(C_E) \subseteq C_E$. Thus, the mappings g'_V and f'_V constitute graph morphisms $f^* : A \rightarrow B$ and $g^* : A \rightarrow C$, respectively.

Universal property:

1. There exists for all graph morphisms $f' : E \rightarrow B$ and $g' : E \rightarrow C$ with $f'; g = g'; f$, i. e. $f'_V; g_V = g'_V; f_V$, a unique mapping $k_V : X_V \rightarrow A_V$ with $k_V; f'_V = f_V$ and $k_V; g'_V = g_V$.

2. Because f^* and f' are graph morphisms, we have

$$\begin{aligned} (f^* \times f_V)((k_V \times k_V)(X_E)) &\subseteq B_E &= (k_V \times k_V); (f^* \times f_V)(X_E) \\ & &= (k_V; f^* \times k_V; f_V)(X_E) \\ & &= (f'_V \times f'_V)(X_E) \end{aligned}$$

and, analogously, we have $(g^* \times g_V)((k_V \times k_V)(X_E)) \subseteq C_E$. Because we have $k_V; f'_V; g_V = k_V; g'_V; f_V$, by assumption, the construction of A_E ensures, in such a way, $(k_V \times k_V)(X_E) \subseteq A_E$ thus k_V establishes a graph morphism $k : X \rightarrow A$.

□

B.1.2 Van Kampen Property for Simple Graphs

It is a well known fact that the category of directed multi-graph \mathbf{Graph} is adhesive [36]. It inherits adhesiveness from the category \mathbf{Set} . However, what about the category \mathbf{SGraph} of simple graphs? Let us have a look at the cube in Figure B.1. All morphisms are given by the identity mapping on the set of vertices and edges are mapped accordingly. Note, that these implicit mappings are unique, as no multiple edges are allowed. The top and the bottom face of the cube in Figure B.1 are pushouts along monomorphism m respectively m' , both back faces are pullbacks, hence both front faces of the cube have to be pullbacks to fulfill the VK property. However, while the left front face is trivially a valid pullback, the right

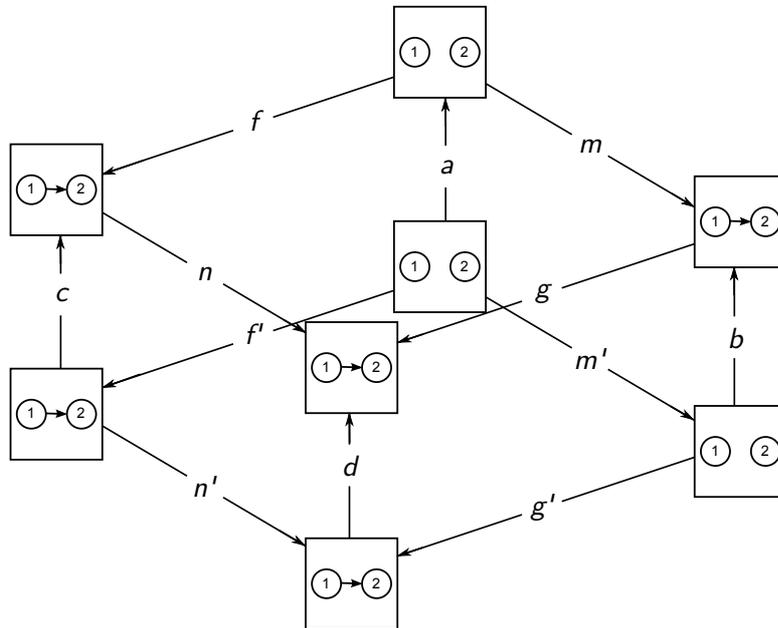


Figure B.1: Counter example VK property in SGraph

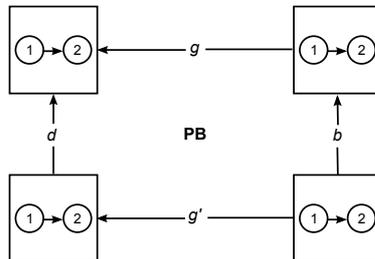


Figure B.2: Corrected pullback in SGraph

face is not. This means the category \mathbf{SGraph} is not adhesive. The valid pullback according to the definition above is shown in Figure B.2.

What goes “wrong” in the category \mathbf{SGraph} ? Pullbacks are constructed by means of pullbacks, inverse images, intersection and equalizers in \mathbf{Set} , i.e. by limit constructions, and should not, therefore, cause any problems. Pushouts in \mathbf{SGraph} are based for vertices on pushouts in \mathbf{Set} which is neither a problem. The construction of the set of edges, however, is based on the union of sets, which is not a colimit construction in \mathbf{Set} . This

union construction in pushouts causes the problem, as edges collapse in the pushout object, i.e. they cannot be traced back. If there is an edge in the pushout, we cannot find out if this edge originates from left or from right or from both sides. To have such a tracing back facility, we have to move from simple to multi-graphs.

B.2 Category of (Directed Multi-)Graphs

The basic definitions for multi-graphs have already been recalled in Chapter 3. Based on general results about functor categories, it is shown in [36] that pushouts, pullbacks, epimorphisms, and monomorphisms in \mathbf{Graph} are exactly given by componentwise pushouts, pullbacks, epimorphisms, and monomorphisms, respectively, in \mathbf{Set} . We develop here these results in a more basic, systematic and detailed way to prepare, in an appropriate way, our investigations of categories of diagrammatic specifications.

B.2.1 Pushouts/Pullbacks of Directed Multi-Graphs

In the following, we show that pushouts, respectively pullbacks in the category \mathbf{Graph} are indeed given by pushouts, respectively pullbacks in the underlying category \mathbf{Set} .

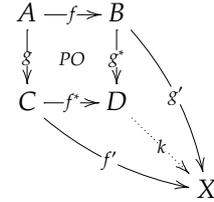
PUSHOUT OF
(DIRECTED
MULTI-)GRAPHS

Proposition B.2.1 (Pushout of (directed multi-)graphs). *A pushout $B \xrightarrow{g^*} D \xleftarrow{f^*} C$ of a span $B \xleftarrow{g} A \xrightarrow{f} C$ of graph morphisms is obtained by constructing componentwise a pushout in \mathbf{Set} for the underlying maps between sets of vertices and sets of edges, respectively. $\text{src}^D : D_E \rightarrow D_V$ and $\text{trg}^D : D_E \rightarrow D_V$ are the unique mediating maps such that $(g_E^*; \text{src}^D = \text{src}^B; g_V^*$ and $f_E^*; \text{src}^D = \text{src}^C; f_V^*)$ or $(g_E^*; \text{trg}^D = \text{trg}^B; g_V^*$ and $f_E^*; \text{trg}^D = \text{trg}^C; f_V^*)$, respectively.*

That is, due to the construction of pushouts in \mathbf{Set} , for each edge $e : x \rightarrow y \in D_E$ the $\text{src}^D(e)$, respectively $\text{trg}^D(e)$ is given by:

$$\text{src}^D(e) = \begin{cases} g_V^*(\text{src}^B(e')) & \text{if } \exists e' \in B_E \text{ with } g_E^*(e') = e \\ f_V^*(\text{src}^C(e')) & \text{else } \exists e' \in C_E \text{ with } f_E^*(e') = e \end{cases}$$

$$\text{trg}^D(e) = \begin{cases} g_V^*(\text{trg}^B(e')) & \text{if } \exists e' \in B_E \text{ with } g_E^*(e') = e \\ f_V^*(\text{trg}^C(e')) & \text{else } \exists e' \in C_E \text{ with } f_E^*(e') = e \end{cases}$$



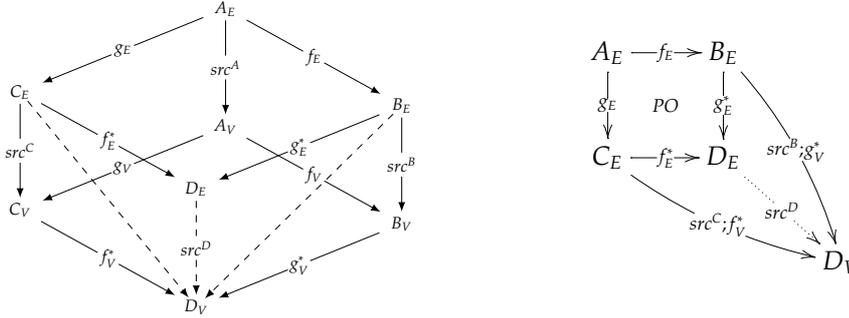
Proof. “ \Leftarrow ”: **Componentwise pushouts in \mathbf{Set} provide pushouts in \mathbf{Graph}**

Existence of source and target maps and homomorphism property: the top face of the cube in the figure below shows the pushout for edges in \mathbf{Set} , while the pushout for vertices in \mathbf{Set} is shown in its bottom face. We have

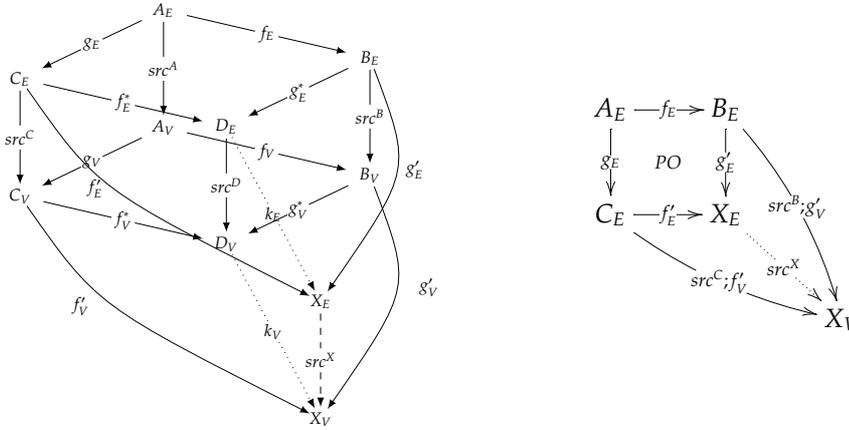
B.2. Category of (Directed Multi-)Graphs

$$\begin{array}{l}
 g_E; src^C; f_V^* = src^A; g_V; f_V^* \\
 = src^A; f_V; g_V^* \\
 = f_E; src^B; g_V^*
 \end{array}
 \left| \begin{array}{l}
 g \text{ graph morphism} \\
 \text{bottom face is commutative} \\
 f \text{ graph morphism}
 \end{array} \right.$$

In addition, analogously, $g_E; trg^C; f_V^* = f_E; trg^B; g_V^*$, thus the uniqueness of mediating morphisms for the pushout for edges entails indeed the existence of unique mappings src^D , respectively trg^D satisfying the equations above. The validity of these equations means, at the same time, that (g_V^*, g_E^*) defines a graph morphism $g^* : B \rightarrow D$ and that (f_V^*, f_E^*) defines a graph morphism $f^* : C \rightarrow D$, respectively.



Universal property: assume graph X and two graph morphisms f' and g' so that $g; f' = f; g'$ (see figures below).



1. There are unique mediating mappings k_E and k_V , as top and bottom faces of the cube are pushouts.
2. By the construction of (k_V, k_E) and the homomorphism property of f', f^*, g' and g^* , respectively, we get $f_E^*; k_E; src^X = f_E^*; src^D; k_V$, and $g_E^*; k_E; src^X = g_E^*; src^D; k_V$, and $f_E^*; k_E; trg^X = f_E^*; trg^D; k_V$, and

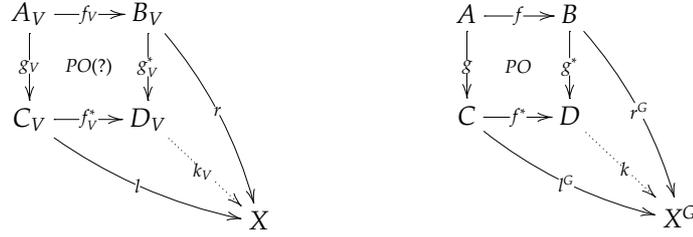
B. CASE STUDY: ADHESIVENESS

$g_E^*; k_E; \text{trg}^X = g_E^*; \text{trg}^D; k_V$, thus, uniqueness of mediating morphisms for the top pushout entails $k_E; \text{src}^X = \text{src}^D; k_V$ and $k_E; \text{trg}^X = \text{trg}^D; k_V$. That is, (k_V, k_E) defines indeed a graph morphism $k : D \rightarrow X$.

“ \Rightarrow ”: **Pushouts in Graph imply componentwise pushouts in Set**

Let a pushout $B \xrightarrow{s^*} D \xleftarrow{f^*} C$ of a span $B \xleftarrow{f} A \xrightarrow{g} C$ of graph morphisms be given.

Pushout for vertices: we consider a set X and a maps $l : C_V \rightarrow X$, $r : B_V \rightarrow X$ such that $g_V; l = f_V; r$.



Existence of mediating map: we construct a graph X^G as follows:

$$X_V^G := X, X_E^G := X \times X, \text{src}^{X^G} := \pi_1, \text{trg}^{X^G} := \pi_2.$$

The map r can be extended then to a graph morphism $r^G : B \rightarrow X^G$:

$$r_V^G := r, r_E^G := \langle \text{src}^B; r, \text{trg}^B; r \rangle,$$

that is, r_E^G is the unique map such that the following diagram commutes

$$\begin{array}{ccccc} B_V & \xleftarrow{\text{src}^B} & B_E & \xrightarrow{\text{trg}^B} & B_V \\ \downarrow & & \downarrow r_E^G & & \downarrow r \\ X & \xleftarrow{\pi_1} & X \times X & \xrightarrow{\pi_2} & X \end{array}$$

This ensures also that (r_V^G, r_E^G) defines a graph morphism. $l^G : C \rightarrow X^G$ is defined analogously.

By construction and assumption, we have $f_V; r_V^G = g_V; l_V^G$ and further

$$\begin{array}{l} f_E; r_E^G = f_E; \langle \text{src}^B; r, \text{trg}^B; r \rangle \\ = \langle f_E; \text{src}^B; r, f_E; \text{trg}^B; r \rangle \\ = \langle \text{src}^A; f_V; r, \text{trg}^A; f_V; r \rangle \\ = \langle \text{src}^A; g_V; l, \text{trg}^A; g_V; l \rangle \\ = \langle g_E; \text{src}^C; l, g_E; \text{trg}^C; l \rangle \\ = g_E; \langle \text{src}^C; l, \text{trg}^C; l \rangle \\ = g_E; l_E^G \end{array} \quad \left| \begin{array}{l} \text{definition } r_E^G \\ \text{pre-composition with tuples} \\ f \text{ graph morphism} \\ \text{assumption} \\ g \text{ graph morphism} \\ \text{pre-composition with tuples} \\ \text{definition } l_E^G \end{array} \right.$$

B.2. Category of (Directed Multi-)Graphs

In such a way, we have $f; r^G = g; l^G$ in **Graph**, thus, there exists a unique graph morphism $k : D \rightarrow X^G$ such that $g^*; k = r^G$ and $f^*; k = l^G$. Especially, we have a mediating map $k_V : D_V \rightarrow X$, such that $g_V^*; k_V = r_V^G = r$ and $f_V^*; k_V = l_V^G = l$.

Uniqueness of mediating map: for any $\bar{k} : D_V \rightarrow X$, such that $g_V^*; \bar{k} = r$ and $f_V^*; \bar{k} = l$ we get $g^*; \bar{k}^G = r^G$ and $f^*; \bar{k}^G = l^G$ according to our constructions, thus the uniqueness of mediators in **Graph** implies $k = \bar{k}^G$ and, especially, $k_V = \bar{k}_V^G = \bar{k}$.

Pushout for edges: we consider a set Y and a maps $l : C_V \rightarrow Y, r : B_V \rightarrow Y$, such that $g_E; l = f_E; r$.



Existence of mediating map: we construct a graph Y^G as follows:

$$Y_V^G := \mathbf{1}, Y_E^G := Y, \text{src}^{Y^G} = \text{tr}g^{Y^G} := !_Y : Y \rightarrow \mathbf{1}$$

where $\mathbf{1}$ is a singleton set, i.e., a terminal object in **Set**. The map r can be extended then to a graph morphism $r^G : B \rightarrow Y^G$:

$$r_V^G := !_{B_V}, r_E^G := r.$$

$$\begin{array}{ccccc} B_V & \xleftarrow{\text{src}^B} & B_E & \xrightarrow{\text{tr}g^B} & B_V \\ \downarrow !_{B_V} & & \downarrow r & & \downarrow !_{B_V} \\ \mathbf{1} & \xleftarrow{!_Y} & Y & \xrightarrow{!_Y} & \mathbf{1} \end{array}$$

By the uniqueness of terminal maps, this ensures that (r_V^G, r_E^G) defines a graph morphism. $l^G : C \rightarrow X^G$ is defined analogously.

By construction and assumption, we have $f_E; r_E^G = g_E; l_E^G$ and further $f_V; r_V^G = g_V; l_V^G$, again due to the uniqueness of terminal maps. This gives $f; r^G = g; l^G$ in **Graph**, thus, there exists a unique graph morphism $k : D \rightarrow Y^G$ such that $g^*; k = r^G$ and $f^*; k = l^G$. Especially, we have a mediating map $k_E : D_E \rightarrow Y$, such that $g_E^*; k_E = r_E^G = r$ and $f_E^*; k_E = l_E^G = l$.

Uniqueness of mediating map: analogously to the case of pushouts for vertices. \square

Now we consider pullbacks.

PULLBACK OF
(DIRECTED
MULTI-)GRAPHS

Proposition B.2.2 (Pullback of (directed multi-)graphs). *A pullback $B \xleftarrow{f^*} A \xrightarrow{g^*} C$ of a cospan $B \xrightarrow{g} D \xleftarrow{f} C$ is obtained by constructing componentwise a pullback in **Set** for the underlying maps between sets of vertices and sets of edges, respectively. $\text{src}^A : A_E \rightarrow A_V$ and $\text{trg}^A : A_E \rightarrow A_V$ are the unique mediating maps, such that $(\text{src}^A; g_V^* = g_E^*; \text{src}^C$ and $\text{src}^A; f_V^* = f_E^*; \text{src}^B)$ or $(\text{trg}^A; g_V^* = g_E^*; \text{trg}^C$ and $\text{trg}^A; f_V^* = f_E^*; \text{trg}^B)$, respectively.*

That is, if we construct pullbacks in **Set** by Cartesian products and equalizer, we have for each edge $e = (e_C, e_B) \in A_E$:

$$\begin{aligned} \text{src}^A(e_C, e_B) &:= (\text{src}^C(e_C), \text{src}^B(e_B)) \\ \text{trg}^A(e_C, e_B) &:= (\text{trg}^C(e_C), \text{trg}^B(e_B)). \end{aligned}$$

$$\begin{array}{ccc} A & \xrightarrow{f^*} & B \\ \downarrow g^* & \text{PB} & \downarrow f \\ C & \xrightarrow{f} & D \end{array}$$

Proof.

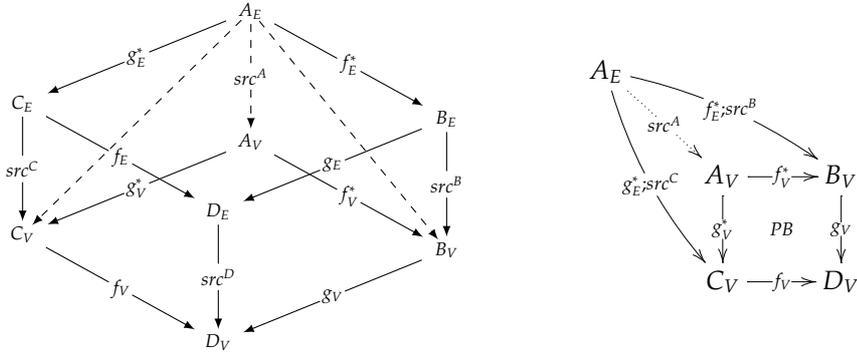
" \Leftarrow ": **Componentwise pullbacks in Set provide pullbacks in Graph**

Existence of source and target maps and homomorphism property: the top face of the cube in the figure below shows the pullback for edges in **Set**, while the pullback for vertices in **Set** is shown in its bottom face. We have

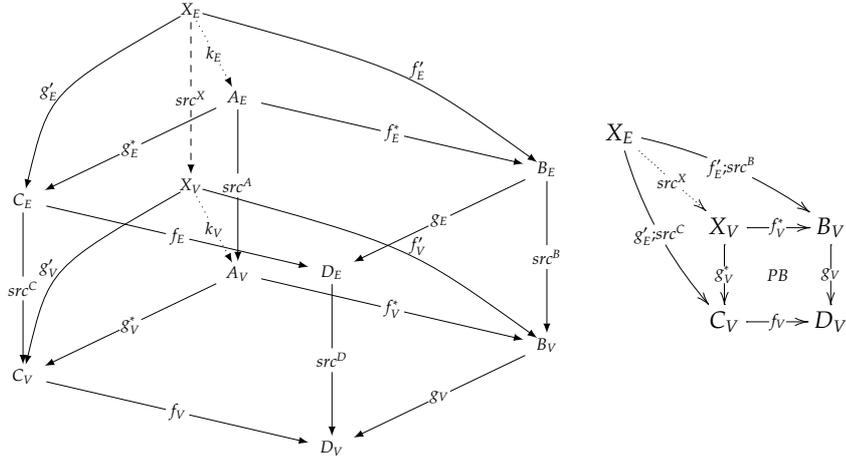
$$\begin{array}{l} g_E^*; \text{src}^C; f_V \\ = f_E^*; g_E; \text{src}^D \\ = f_E^*; \text{src}^B; g_V \end{array} \quad \left| \begin{array}{l} f \text{ graph morphism} \\ \text{bottom face is commutative} \\ g \text{ graph morphism} \end{array} \right.$$

and, analogously, $g_E^*; \text{trg}^C; f_V = f_E^*; \text{trg}^B; g_V^*$, thus, the uniqueness of mediating morphisms for the pullback for vertices entails indeed the existence of unique mappings src^D , respectively trg^D satisfying the equations above. The validity of these equations means, at the same time, that (g_V^*, g_E^*) defines a graph morphism $g^* : A \rightarrow C$ and that (f_V^*, f_E^*) defines a graph morphism $f^* : A \rightarrow B$, respectively.

B.2. Category of (Directed Multi-)Graphs



Universal property: assume graph X and two graph morphisms f' and g' , so that $g'; f = f'; g$ (see figures below).



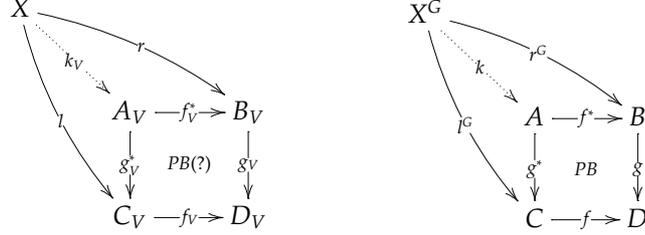
1. There are unique mappings k_E and k_V , as top and bottom faces of the cube are pullbacks.
2. By the construction of (k_V, k_E) and the homomorphism property of f' , f^* , g' and g^* , respectively, we get $k_E; src^A; f_V^* = src^X; k_V; f_V^*$, and $k_E; src^A; g_V^* = src^X; k_V; g_V^*$, and $k_E; trg^A; f_V^* = trg^X; k_V; f_V^*$, and $k_E; trg^A; g_V^* = trg^X; k_V; g_V^*$, thus, the uniqueness of mediating morphisms for the bottom pullback entails $k_E; src^A = src^X; k_V$ and $k_E; trg^A = trg^X; k_V$. That is, (k_V, k_E) defines indeed a graph morphism $k : X \rightarrow A$.

" \Rightarrow ": Pullbacks in Graph imply pullbacks of their components in Set

Let a pullback $B \xleftarrow{f^*} A \xrightarrow{g^*} C$ of a cospan $B \xrightarrow{g} D \xleftarrow{f} C$ of graph morphisms be given.

B. CASE STUDY: ADHESIVENESS

Pullbacks for vertices: we consider a set X and maps $l : X \rightarrow C_V$, $r : X \rightarrow B_V$ such that $l; g_V = r; f_V$.



Existence of mediating map: we construct a graph X^G as follows:

$$X_V^G := X, X_E^G := \emptyset, \text{src}^{X^G} = \text{trg}^{X^G} := !_X : \emptyset \rightarrow X,$$

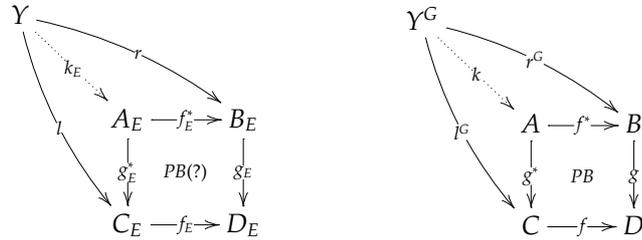
where \emptyset is the empty set, i.e. the initial object in **Set**. The map r can be extended then to a graph morphism $r^G : X^G \rightarrow B$:

$$r_V^G := r, r_E^G := !_{B_E}.$$

By the uniqueness of initial maps, this ensures that (r_V^G, r_E^G) defines a graph morphism $r^G : X^G \rightarrow B$. $l^G : X^G \rightarrow C$ is defined analogously. We have $l_V^G; f_V = r_V^G; g_V$ by assumption and $l_E^G; f_E = r_E^G; g_E$ by uniqueness of initial maps. This means $l^G; f = r^G; g$ in **Graph**, thus, there exists a unique graph morphism $k : X^G \rightarrow A$, such that $k; g^* = l^G$ and $k; f^* = r^G$. Especially, we have $k_V; g_V^* = l_V^G$ and $k_V; f_V^* = r_V^G = r$, as desired.

Uniqueness of mediating map: analogously to the case of pushouts for vertices.

Pullbacks for edges: we consider a set Y and maps $l : Y \rightarrow C_E$, $r : Y \rightarrow B_E$ such that $l; g_E = r; f_E$.



Existence of mediating map: we construct a graph Y^G as follows:

$$Y_V^G := Y + Y, Y_E^G := Y, \text{src}^{Y^G} := \kappa_1, \text{trg}^{Y^G} := \kappa_2.$$

The map r can be extended then to a graph morphism $r^G : Y^G \rightarrow B$:

$$r_V^G := [r; \text{src}^B, r; \text{trg}^B], r_E^G := r,$$

i.e. r_V^G is the unique map such that the following diagram commutes:

$$\begin{array}{ccccc} Y & \xrightarrow{\kappa_1} & Y + Y & \xleftarrow{\kappa_2} & Y \\ \downarrow r & & \downarrow r_V^G & & \downarrow r \\ B_E & \xrightarrow{\text{src}^B} & B_V & \xleftarrow{\text{trg}^B} & B_E \end{array}$$

This ensures also that (r_V^G, r_E^G) defines a graph morphism $r^G : Y^G \rightarrow B$. $l^G : Y^G \rightarrow C$ is defined analogously.

By construction and assumption, we have $l_E^G; f_E = r_E^G; g_E$ and, dually to the case of pushouts for nodes, we can show $l_V^G; f_V = r_V^G; g_V$. This means $l^G; f = r^G; g$ in **Graph**, thus, there exists a unique graph morphism $k : Y^G \rightarrow A$, such that $k; g^* = l^G$ and $k; f^* = r^G$. Especially, we have $k_V; g_V^* = l_V^G$ and $k_V; f_V^* = r_V^G = r$, as desired.

Uniqueness of mediating map: analogously to the case of pushouts for vertices. □

B.2.2 Van Kampen Property for Directed Multi-Graphs

In the following, we consider the VK property for **Graph**. Because this time the construction of pushouts does not rely on a union construction anymore, we expect the category to be adhesive.

Lemma B.2.1 (Monic = Pullback).

A morphism $f : A \rightarrow B$ in a category **C** is a monomorphism iff the figure on the right is a pullback diagram.

$$\begin{array}{ccc} A & \xrightarrow{id_A} & A \\ \downarrow id_A & & \downarrow f \\ A & \xrightarrow{f} & B \end{array}$$

MONIC = PULLBACK

Corollary B.2.1 (Monomorphism in (directed multi-)graphs). Monomorphisms in **Graph** are given by componentwise monomorphisms in **Set** i.e. a graph morphism $f : A \rightarrow B$ is a monomorphism iff $f_V : A_V \rightarrow B_V$ and $f_E : A_E \rightarrow B_E$ are monomorphisms. MONOMORPHISM
IN (DIRECTED
MULTI-)GRAPHS

Proof. Because the identities in **Graph** are componentwise identities, we get by Lemma B.2.1, Proposition B.2.2 and again Lemma B.2.1 for any graph morphism in $f : A \rightarrow B$:

- f is a monomorphism in **Graph**
- $\iff A \xleftarrow{id_A} A \xrightarrow{id_A} A$ is the pullback of $A \xrightarrow{f} B \xleftarrow{f} A$
- $\iff A_V \xleftarrow{id_{A_V}} A_V \xrightarrow{id_{A_V}} A_V$ is the pullback of $A_V \xrightarrow{f_V} B_V \xleftarrow{f_V} A_V$ and $A_E \xleftarrow{id_{A_E}} A_E \xrightarrow{id_{A_E}} A_E$ is the pullback of $A_E \xrightarrow{f_E} B_E \xleftarrow{f_E} A_E$ in **Set**
- $\iff f_V : A_V \rightarrow B_V$ and $f_E : A_E \rightarrow B_E$ are monomorphism in **Set**

□

EPIC = PUSHOUT

Lemma B.2.2 (Epic = Pushout). *A morphism $f : A \rightarrow B$ in a Category **C** is an epimorphism iff the figure on the right is a pushout diagram.*

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow f & & \downarrow id_B \\ B & \xrightarrow{id_B} & B \end{array}$$

EPIMORPHISM IN
(DIRECTED
MULTI-)GRAPHS

Corollary B.2.2 (Epimorphism in (directed multi-)graphs). *Epimorphisms in **Graph** are given by componentwise epimorphisms in **Set** i.e. a graph morphism $f : G \rightarrow H$ is an epimorphism iff $f_V : G_V \rightarrow H_V$ and $f_E : G_E \rightarrow H_E$ are epimorphisms.*

Proof. Because the identities in **Graph** are componentwise identities, we get by Lemma B.2.2, Proposition B.2.1 and again Lemma B.2.2 for any graph morphism in $f : A \rightarrow B$:

- f is an epimorphism in **Graph**
- $\iff B \xrightarrow{id_B} B \xleftarrow{id_B} B$ is the pushout of $B \xleftarrow{f} A \xrightarrow{f} B$
- $\iff B_V \xrightarrow{id_{B_V}} B_V \xleftarrow{id_{B_V}} B_V$ is the pushout of $B_V \xleftarrow{f_V} A_V \xrightarrow{f_V} B_V$ and $B_E \xrightarrow{id_{B_E}} B_E \xleftarrow{id_{B_E}} B_E$ is the pushout of $B_E \xleftarrow{f_E} A_E \xrightarrow{f_E} B_E$
- $\iff f_V : A_V \rightarrow B_V$ and $f_E : A_E \rightarrow B_E$ are epimorphism in **Set**

□

Considering all facts about the category of directed multi-graph **Graph**, we can conclude that it is adhesive [36] and that it inherits adhesiveness from category **Set**.

Proposition B.2.3. *Pushouts along monomorphism are VK squares in **Graph**.*

Proof. Assume a commutative cube (2) in **Graph** (See Definition 4.1.1). As composition of graph morphism is defined componentwise, we have (2) commutes in **Graph** iff the corresponding two cubes for vertices and edges commute in **Set**. We assume top face in (2) is a pushout and both back faces in (2) are pullbacks. By Proposition B.2.1 and Proposition B.2.2 follows:

$$\text{top face in (1) is a pushout} \iff \text{top face in (1) for } (1_V) \text{ and } (1_E) \text{ are pushouts and}$$

B.2. Category of (Directed Multi-)Graphs

back faces in (2) are pullbacks \iff back faces in (2) for (2_V) and (2_E) are pullbacks

We have by Proposition B.2.1, Proposition B.2.2 and **Set** is adhesive:

bottom face in (2) is pushout \iff bottom faces in (2_V) and (2_E) are pushouts
 \iff front faces in (2_V) and (2_E) are pullbacks
 \iff front faces in (2) are pullbacks \square

Let us review Figure B.1, which shows a counterexample for the VK property in **SGraph**. The cube in the figure is not a valid VK cube even though the top and bottom faces are pushouts and both back faces are pullbacks. Figure B.3 shows the analog example for **Graph**. This time we also have to map the edges, as we can have more than one edge having the same source and target vertex. In contrast to the earlier example, we get two edges in the upper pushout graph. Therefore, we can correctly trace back the edges in the right front face of the cube that shows in contrast to Figure B.1 a valid pullback diagram. Hence, Figure B.3 shows a valid VK cube.

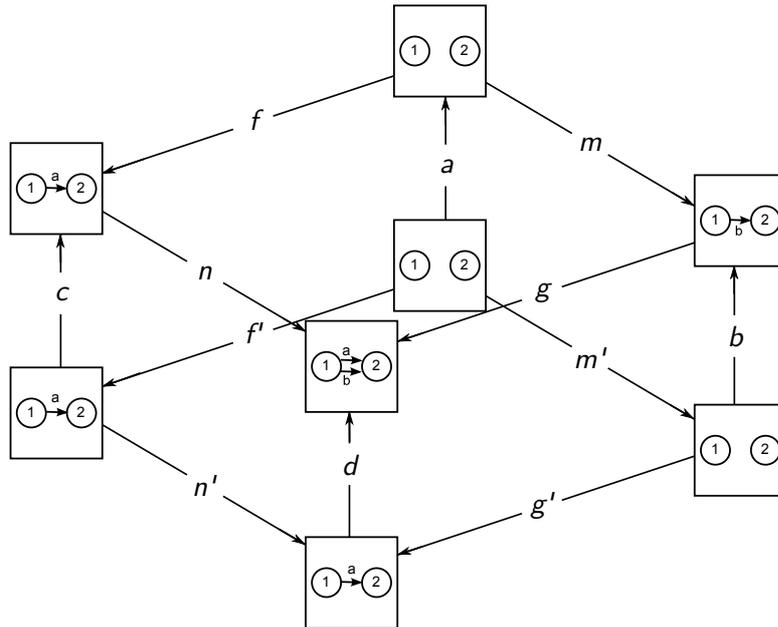


Figure B.3: Example VK property in **Graph**

B.3 Category of DPF Specifications

Now, we consider **Spec**, the category of DPF specifications, and show that it is not adhesive for a similar reason **SGraph** is not adhesive. Therefore, let us recall the main definitions from [121] and [120] in the following:

SIGNATURE	Definition B.3.1 (Signature). A signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$ consists of a set of predicate symbols Π^Σ and a mapping α^Σ which assigns a (multi-)graph to each predicate symbol $\pi \in \Pi^\Sigma$. $\alpha^\Sigma(\pi)$ is called the <i>arity</i> of the predicate symbol π .
ATOMIC CONSTRAINT	Definition B.3.2 (Atomic constraint). Given a signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$, an atomic constraint (π, δ) on a (multi-)graph S consists of a predicate symbol $\pi \in \Pi^\Sigma$ and a graph morphism $\delta : \alpha^\Sigma(\pi) \rightarrow S$.
SPECIFICATION	Definition B.3.3 (Specification). Given a signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$, a specification $\mathfrak{S} = (S, C^\mathfrak{S} : \Sigma)$ consists of a multi-graph S and a set $C^\mathfrak{S}$ of atomic constraints (π, δ) on S with $\pi \in \Pi^\Sigma$.
SPECIFICATION MORPHISM	Definition B.3.4 (Specification morphism). Given two specifications $\mathfrak{S} = (S, C^\mathfrak{S} : \Sigma)$ and $\mathfrak{S}' = (S', C^{\mathfrak{S}'} : \Sigma)$, a specification morphism $\phi : \mathfrak{S} \rightarrow \mathfrak{S}'$ is a graph morphism $\phi : S \rightarrow S'$ such that $(\pi, \delta) \in C^\mathfrak{S}$ implies $\phi_C(\pi, \delta) := (\pi, \delta; \phi) \in C^{\mathfrak{S}'}$.

$$\begin{array}{ccc}
 & \delta; \phi & \\
 & \curvearrowright & \\
 & = & \\
 \alpha^\Sigma(\pi) & \xrightarrow{\delta} S & \xrightarrow{\phi} S'
 \end{array}$$

Definition B.3.5 (Category of specifications). Given a signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$, the category **Spec**(Σ) has all specifications $\mathfrak{S} = (S, C^\mathfrak{S} : \Sigma)$ as objects and all specification morphisms $\phi : \mathfrak{S} \rightarrow \mathfrak{S}'$ as morphisms between specifications \mathfrak{S} and \mathfrak{S}' .

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

Remark B.3.1 (Monomorphism for specifications). Monomorphism in **Spec**(Σ) are the morphism where the underlying graph morphism is a monomorphism. Definition B.3.4 ensures that the translation $\phi_C : C^\mathfrak{S} \rightarrow C^{\mathfrak{S}'}$ of atomic constraints becomes, in this case, also an injective mapping. Compare Remark B.1.1, where we have the same effect for the translation of edges.

B.3.1 Pushouts and Pullbacks in Spec

In this section, we will consider the general construction of pushouts and pullbacks in Spec relying on the existence of pushouts and pullbacks in Graph , as done in [151], for generalized sketches¹. This restricts the task primarily on the consideration of atomic constraints in the context of Spec .

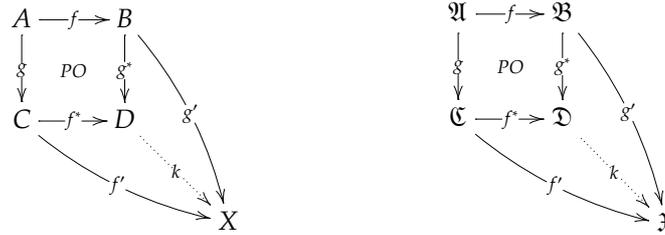
Note, the category of (typed) **conformant** specifications [120, 121] will not be considered here, as **conformance** relies on arbitrary semantics assigned to atomic constraints that are in general not specified in terms of graph morphism and commuting diagrams.

Similar to pushouts in SGraph , pushouts in Spec are based on pushouts in Graph and the union of sets (of constraints).

Proposition B.3.1 (Pushout of specifications). *A pushout $\mathfrak{D} = (D, C^{\mathfrak{D}}; \Sigma)$ of a span $\mathfrak{B} \xleftarrow{f} \mathfrak{A} \xrightarrow{g} \mathfrak{C}$ of a specification morphisms is obtained by constructing a pushout in Graph for the underlying graph morphisms and by defining the set of atomic constraints $C^{\mathfrak{D}}$ as follows:*

PUSHOUT OF
SPECIFICATIONS

$$C^{\mathfrak{D}} := \{(\pi, \delta; g^*) \mid (\pi, \delta) \in C^{\mathfrak{B}}\} \cup \{(\pi, \delta; f^*) \mid (\pi, \delta) \in C^{\mathfrak{C}}\}$$



Proof.

Morphism property: we have $f_C^*(C^{\mathfrak{C}}) \subseteq C^{\mathfrak{D}}$ and $g_C^*(C^{\mathfrak{B}}) \subseteq C^{\mathfrak{D}}$, by construction, thus the graph morphisms f^* and g^* constitute specification morphisms $f^* : \mathfrak{C} \rightarrow \mathfrak{D}$ and $g^* : \mathfrak{B} \rightarrow \mathfrak{D}$, respectively.

Universal property:

1. There exists for all specification morphisms $g' : \mathfrak{B} \rightarrow \mathfrak{X}$ and $f' : \mathfrak{C} \rightarrow \mathfrak{X}$ with $g; f' = f; g'$ i.e. $g; f' = f; g'$ a unique graph morphism $k : D \rightarrow X$ with $f^*; k = f'$ and $g^*; k = g'$.

¹The construction for pushouts and pullbacks in Spec can also be found in [121] and [120] for restricted cases.

B. CASE STUDY: ADHESIVENESS

2. Because f^* and f' are specification morphisms, we have for any $\pi \in \Sigma$ and $(\pi, \delta_C) \in C^{\mathbb{C}}$

$$(\pi, \delta_C; f') = (\pi, \delta_C; f^*; k) \subseteq C^{\mathfrak{X}}$$

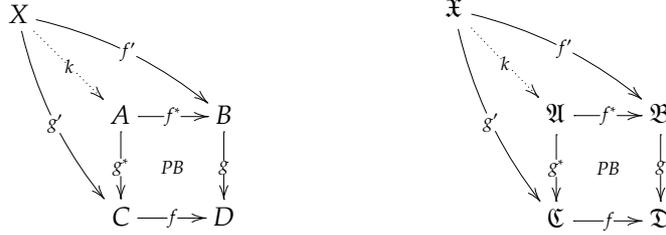
and analogously $(\pi, \delta_B; g^*; k) \subseteq C^{\mathfrak{X}}$ for any $\pi \in \Sigma$ and $(\pi, \delta_B) \in C^{\mathfrak{B}}$. Due to the construction of $C^{\mathfrak{D}}$, this ensures $(\pi, \delta_D; k) \subseteq C^{\mathfrak{X}}$, for all $(\pi, \delta_D) \in C^{\mathfrak{D}}$, thus k establishes a specification morphism $k : \mathfrak{D} \rightarrow \mathfrak{X}$. \square

Similar to pullbacks in **SGraph**, pullbacks in **Spec** rely as well on pullbacks in **Graph** as on intersection and equalizers in **Set**.

Proposition B.3.2 (Pullback of specifications). *A pullback $\mathfrak{A} = (A, C^{\mathfrak{A}}; \Sigma)$ of a cospan $B \xrightarrow{g} D \xleftarrow{f} C$ is obtained by constructing a pullback in **Graph** for the underlying graph morphisms and by defining the set of atomic constraints $C^{\mathfrak{A}}$ as follows:*

PULLBACK OF SPECIFICATIONS

$$C^{\mathfrak{A}} := \{(\pi, \delta : \alpha^{\Sigma}(\pi) \rightarrow A) \mid (\pi, \delta; f^*) \in C^{\mathfrak{B}} \text{ and } (\pi, \delta; g^*) \in C^{\mathbb{C}} \text{ and } (\pi, \delta; f^*; g) = (\pi, \delta; g^*; f) \in C^{\mathfrak{D}}\}$$



Proof.

Morphism property: by construction, we have $f_C^*(C^{\mathfrak{A}}) \subseteq C^{\mathfrak{B}}$ and $g_C^*(C^{\mathfrak{A}}) \subseteq C^{\mathbb{C}}$ thus the graph morphisms f^* and g^* constitute specification morphisms $f^* : \mathfrak{A} \rightarrow \mathfrak{B}$ and $g^* : \mathfrak{A} \rightarrow \mathbb{C}$, respectively.

Universal property:

1. There exists for all specification morphisms $f' : \mathfrak{X} \rightarrow \mathfrak{B}$ and $g' : \mathfrak{X} \rightarrow \mathbb{C}$ with $f'; g = g'; f$ a unique graph morphism $k : X \rightarrow A$ with $k; f^* = f'$ and $k; g^* = g'$.

2. Since f^* and f' are specification morphisms we have for any $\pi \in \Sigma$ and $(\pi, \delta_X) \in C^{\mathfrak{X}}$

$$(\pi, \delta_X; f') = (\pi, \delta_X; k; f^*) \subseteq C^{\mathfrak{B}}$$

and analogously $(\pi, \delta_X; k; g^*) \subseteq C^{\mathfrak{C}}$ for any $\pi \in \Sigma$ and $(\pi, \delta_X) \in C^{\mathfrak{X}}$. Because we have $k; f^*; g = k; g^*; f$, by assumption, the construction of $C^{\mathfrak{A}}$ ensures, in such a way, $(\pi, \delta_X; k) \subseteq C^{\mathfrak{A}}$ for any $\pi \in \Sigma$ and $(\pi, \delta_X) \in C^{\mathfrak{X}}$ thus k establishes a specification morphism $k : \mathfrak{X} \rightarrow \mathfrak{A}$.

□

B.3.2 Van Kampen Property in Spec

In Section B.1, we considered SGraph and in particular pushouts in SGraph. In Subsection B.1.2, we have presented a counterexample for adhesiveness in SGraph and realized that SGraph is not adhesive due to the fact that pushouts rely on a union construction on edges.

Now, we analyze pushouts in Spec. Because pushouts in Spec rely again on a union construction, we expect the category Spec of specifications not to be adhesive. Indeed, Spec is not adhesive. Figure B.4 shows a counterexample for Spec. Predicates are given as concrete sets. The top and the bottom faces are pushouts along monomorphisms. The back faces are pullbacks. In addition, the left front face is a pullback. However, the right front face is not a pullback, as $\pi_1 \in C^{\mathfrak{B}}$ and $\pi_1 \in C^{\mathfrak{D}'}$ but $\pi_1 \notin C^{\mathfrak{B}'}$.

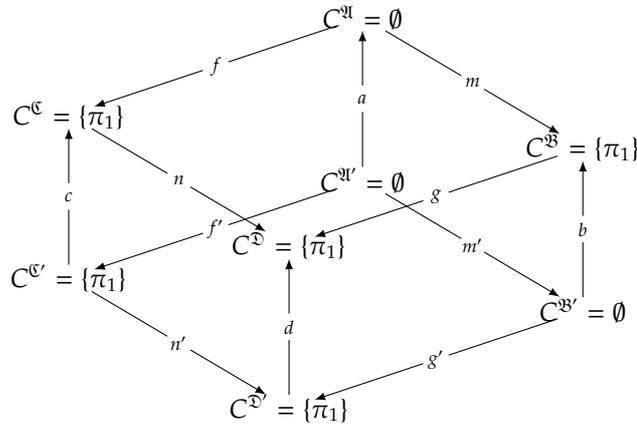


Figure B.4: Counter example VK property in Spec

Note, that the reason is exactly the same as before in SGraph. Pushouts in Spec rely on a union construction and hence predicates cannot be traced

$$\begin{array}{ccc}
 C^{\mathfrak{D}} = \{\pi_1\} & \leftarrow^n & C^{\mathfrak{B}} = \{\pi_1\} \\
 \uparrow d & & \uparrow b \\
 C^{\mathfrak{D}'} = \{\pi_1\} & \leftarrow^{n'} & C^{\mathfrak{B}'} = \{\pi_1\}
 \end{array}
 \quad PB$$

Figure B.5: Corrected pullback in Spec

back. The valid pullback according to the definition above is shown in Figure B.5.

In contrast to SGraph, we have seen that category Graph is adhesive and that the main difference between both is that the pushout construction in Graph only relies on pushouts in Set. In the following, we will consider the category of generalized DPF Specifications that “repairs” the category of “usual” DPF specification so that we obtain an adhesive one.

B.4 The Category of Generalized DPF Specifications

In the following, we analyze the definitions of Section B.3 to prepare for a generalization².

To lift the definitions in Section B.3 to a more structured and abstract level, we make, first, explicit the arity mapping $\alpha^\Sigma : \Pi^\Sigma \rightarrow \mathbf{Graph}_0$ where \mathbf{Graph}_0 denotes the set of objects in the category Graph.

Second, we consider the slice category (\mathbf{Graph}/S) and the mapping $fst^S : (\mathbf{Graph}/S)_0 \rightarrow \mathbf{Graph}_0$ assigning to each object $\varphi : G \rightarrow S$ in (\mathbf{Graph}/S) the domain G . Then we consider the pullback of the cospan

$$\Pi^\Sigma \xrightarrow{\alpha^\Sigma} \mathbf{Graph}_0 \xleftarrow{fst^S} (\mathbf{Graph}/S)_0,$$

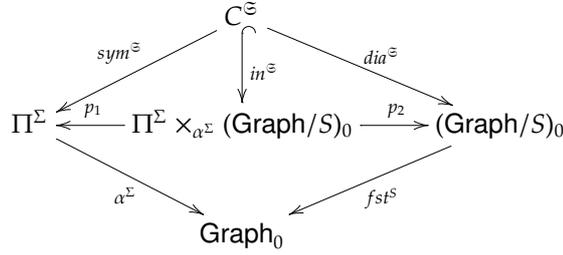
i.e., the set

$$\Pi^\Sigma \times_{\alpha^\Sigma} (\mathbf{Graph}/S)_0 = \{(\pi, \varphi) \in \Pi^\Sigma \times (\mathbf{Graph}/S)_0 \mid \alpha^\Sigma(\pi) = fst^S(\varphi)\}.$$

The main point is that our “traditional” atomic constraints from Definition B.3.2 are exactly the elements of $\Pi^\Sigma \times_{\alpha^\Sigma} (\mathbf{Graph}/S)_0$. That is, a “traditional” specification is given by a multi-graph S and a subset $C^\cong \subseteq \Pi^\Sigma \times_{\alpha^\Sigma} (\mathbf{Graph}/S)_0$.

²The formalization of this section is based on insights and ideas of Zinovy Diskin communicated in 2007.

B.4. The Category of Generalized DPF Specifications

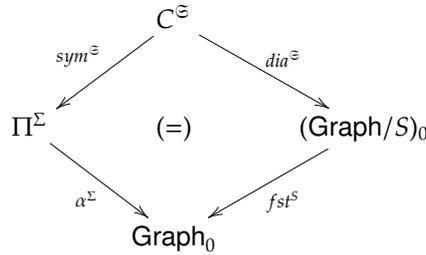


The important observation is that the maps $\text{sym}^{\mathfrak{E}} := \text{in}^{\mathfrak{E}}; p_1$ and $\text{dia}^{\mathfrak{E}} := \text{in}^{\mathfrak{E}}; p_2$ make the square commute and are jointly injective.

We moved from simple graphs to multi-graphs by introducing (identifiers for) edges independent of vertices and by dropping the requirement that an edge is uniquely determined by its source and target. Analogously, we introduce now (identifiers for) constraints independent of predicate symbols and carrier graphs and we drop the requirement that $\text{sym}^{\mathfrak{E}}$ and $\text{dia}^{\mathfrak{E}}$ are jointly monic.

Definition B.4.1 (Generalized specification). Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, a generalized specification $\mathfrak{E} = (S, C^{\mathfrak{E}}, \text{sym}^{\mathfrak{E}}, \text{dia}^{\mathfrak{E}})$ consists of a multi-graph S , a set $C^{\mathfrak{E}}$ of "constraint identifiers" and two maps $\text{sym}^{\mathfrak{E}} : C^{\mathfrak{E}} \rightarrow \Pi^{\Sigma}$ and $\text{dia}^{\mathfrak{E}} : C^{\mathfrak{E}} \rightarrow \text{Graph}_0$, such that the following diagram commutes:

GENERALIZED
SPECIFICATION



Remark B.4.1 (Generalized atomic constraints). In practice, the "constraint identifiers" will be often constructed as triples (l, π, δ) , with l a "label/tag" (indicating, for example, whom introduced the constraint), π a predicate symbol and $\delta : \alpha^{\Sigma}(\pi) \rightarrow S$ a graph morphism. In those practical cases, the maps $\text{sym}^{\mathfrak{E}}$ and $\text{dia}^{\mathfrak{E}}$ are given by the second and third projection, respectively.

For the definition of morphisms, we have to remind that any graph morphism $\phi_G : S \rightarrow S'$ induces a functor $\overline{\phi}_G : (\text{Graph}/S) \rightarrow (\text{Graph}/S')$ with $\overline{\phi}_G; \text{fst}^{S'} = \text{fst}^S$, which is defined by simple post-composition, i.e., $\overline{\phi}_G(\gamma) := \gamma; \phi_G$ for all objects $\gamma : G \rightarrow S$ in Graph/S .

MORPHISMS
BETWEEN
GENERALIZED
SPECIFICATIONS

Definition B.4.2 (Morphisms between generalized specifications). Given two generalized specifications $\mathfrak{S} = (S, C^{\mathfrak{S}}, \text{sym}^{\mathfrak{S}}, \text{dia}^{\mathfrak{S}})$ and $\mathfrak{S}' = (S', C^{\mathfrak{S}'}, \text{sym}^{\mathfrak{S}'}, \text{dia}^{\mathfrak{S}'})$, a specification morphism $f = (f_C, f_G) : \mathfrak{S} \rightarrow \mathfrak{S}'$ is given by a mapping $f_C : C^{\mathfrak{S}} \rightarrow C^{\mathfrak{S}'}$ and a graph morphism $f_G : S \rightarrow S'$, such that the following two diagrams commute:

$$\begin{array}{ccc} C^{\mathfrak{S}} & \xrightarrow{f_C} & C^{\mathfrak{S}'} \\ & \searrow \text{sym}^{\mathfrak{S}} & \swarrow \text{sym}^{\mathfrak{S}'} \\ & \Pi^{\Sigma} & \end{array} \qquad \begin{array}{ccc} C^{\mathfrak{S}} & \xrightarrow{f_C} & C^{\mathfrak{S}'} \\ \text{dia}^{\mathfrak{S}} \downarrow & & \downarrow \text{dia}^{\mathfrak{S}'} \\ (\text{Graph}/S)_0 & \xrightarrow{\overline{f_G}} & (\text{Graph}/S')_0 \end{array}$$

CATEGORY
GSpec(Σ)

Definition B.4.3 (Category of generalized specifications). Given a signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$, the category $\text{GSpec}(\Sigma)$ has all generalized specifications $\mathfrak{S} = (S, C^{\mathfrak{S}}, \text{sym}^{\mathfrak{S}}, \text{dia}^{\mathfrak{S}})$ as objects and all generalized specification morphisms $\phi : \mathfrak{S} \rightarrow \mathfrak{S}'$ as morphisms between generalized specifications \mathfrak{S} and \mathfrak{S}' .

The composition $\phi; \psi : \mathfrak{G} \rightarrow \mathfrak{R}$ of two (generalized) specification morphisms $\phi : \mathfrak{G} \rightarrow \mathfrak{H}$ and $\psi : \mathfrak{H} \rightarrow \mathfrak{R}$ is defined componentwise $\phi; \psi = (\phi_C, \phi_G); (\psi_C, \psi_G) := (\phi_C; \psi_C, \phi_G; \psi_G)$. The identity (generalized) specification morphism $\text{id}^{\mathfrak{G}} : \mathfrak{G} \rightarrow \mathfrak{G}$ is also defined componentwise $\text{id}^{\mathfrak{G}} = (\text{id}^{C^{\mathfrak{G}}}, \text{id}^G)$. This ensures that the composition of specification monomorphisms is associative and that identity specification morphism are left and right neutral with respect to composition.

The definitions above Provide us a category of generalized specifications and it should be possible to prove now that pushouts and pullbacks in this category are given by pushouts, respectively pullbacks of the underlying graphs plus the pushout, respectively pullbacks of the corresponding sets of identifiers! Because we used in the definitions above category Graph as an underlying category that is adhesive as well as category Set , we get a result that the category of generalized specifications GSpec is adhesive.

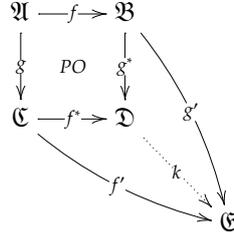
B.4.1 Pushouts and Pullbacks in GSpec

In the following, we show that a pushout, respectively pullback in category GSpec is indeed given by the pushout, respectively pullback in the underlying category of graphs, as well as the pushout, respectively pullbacks of the corresponding sets of identifiers.

Proposition B.4.1 (Pushout of generalized specifications). *A pushout $\mathfrak{B} \xrightarrow{\mathfrak{g}} \mathfrak{D} \xleftarrow{\mathfrak{f}} \mathfrak{C}$ of a span $\mathfrak{B} \xleftarrow{\mathfrak{f}} \mathfrak{A} \xrightarrow{\mathfrak{g}} \mathfrak{C}$ of generalized specification morphisms is obtained by constructing a pushout in Graph for the underlying graph morphisms as well as constructing a pushout in Set for the underlying maps between sets of constraint identifiers.*

PUSHOUT OF
GENERALIZED
SPECIFICATIONS

B.4. The Category of Generalized DPF Specifications

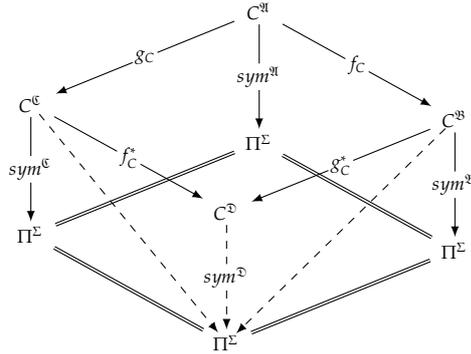


Proof.

“ \Leftarrow ”: **Componentwise pushouts in Graph and Set provide pushouts in GSpec.**

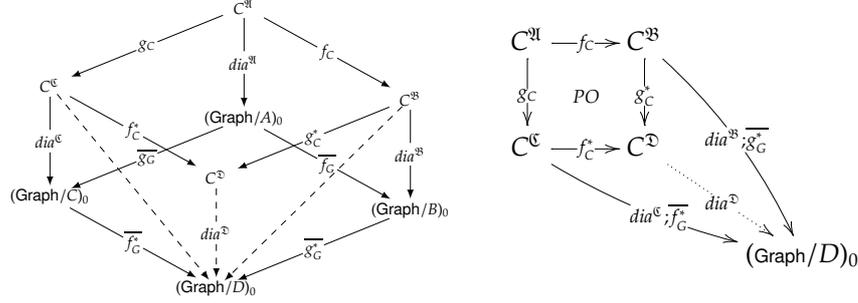
We consider the pushout $C^{\mathfrak{B}} \xrightarrow{g_C^*} C^{\mathfrak{D}} \xleftarrow{f_C^*} C^{\mathfrak{C}}$ of the span $C^{\mathfrak{B}} \xleftarrow{f_C} C^{\mathfrak{A}} \xrightarrow{g_C} C^{\mathfrak{C}}$ in **Set** and the pushout $B \xrightarrow{g_G^*} D \xleftarrow{f_G^*} C$ of the span $B \xleftarrow{f_G} A \xrightarrow{g_G} C$ in **Graph**.

Existence of symbol map: since (f_C, f_G) and (g_C, g_G) are specification morphisms we have $g_C; sym^{\mathfrak{C}} = sym^{\mathfrak{A}} = f_C; sym^{\mathfrak{B}}$, thus there exists a unique map $sym^{\mathfrak{D}} : C^{\mathfrak{D}} \rightarrow \Pi^{\Sigma}$ with $g_C^*; sym^{\mathfrak{D}} = sym^{\mathfrak{B}}$ and $f_C^*; sym^{\mathfrak{D}} = sym^{\mathfrak{C}}$.



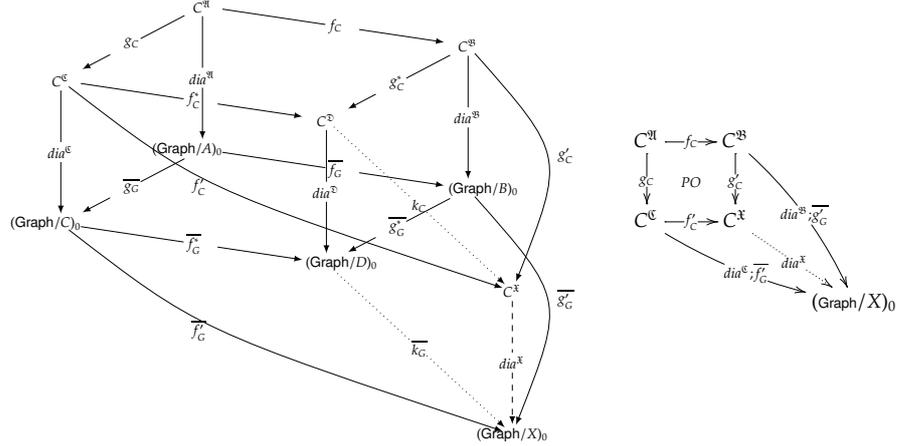
Existence of diagram map: the top face of the cube in the figure below shows the pushout for constraints in **Set**, while the commutative square induced by the pushout of the underlying graph morphisms is shown in its bottom face. Analogously to Proposition B.2.1, we obtain $g_C; dia^{\mathfrak{C}}; \overline{f_G^*} = f_C; dia^{\mathfrak{B}}; \overline{g_G^*}$, thus there exists a unique map $dia^{\mathfrak{D}} : C^{\mathfrak{D}} \rightarrow (\text{Graph}/S)_0$ with $g_C^*; dia^{\mathfrak{D}} = dia^{\mathfrak{B}}; \overline{g_G^*}$ and $f_C^*; dia^{\mathfrak{D}} = dia^{\mathfrak{C}}; \overline{f_G^*}$.

B. CASE STUDY: ADHESIVENESS



Morphism property: the equations above ensure that the pair $(g_C^*, \overline{g}_G^*)$ defines a specification morphism $g^* : \mathfrak{B} \rightarrow \mathfrak{D}$ and that the pair $(f_C^*, \overline{f}_G^*)$ defines a specification morphism $f^* : \mathfrak{C} \rightarrow \mathfrak{D}$, respectively.

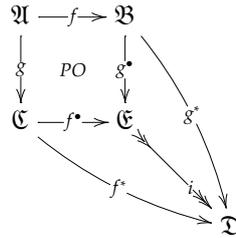
Universal property: assume specification \mathfrak{X} and two specification morphisms $f' : \mathfrak{C} \rightarrow \mathfrak{X}$ and $g' : \mathfrak{B} \rightarrow \mathfrak{X}$ so that $g; f' = f; g'$ (see figures below).



1. Due to the pushout in \mathbf{Set} , there is a unique mediating mapping k_C with $g_C^*; k_C = g_C'$, $f_C^*; k_C = f_C'$ and, due to the pushout in \mathbf{Graph} , there is a unique graph morphism k_G with $\overline{g}_G^*; k_G = \overline{g}_G'$, $\overline{f}_G^*; k_G = \overline{f}_G'$. This ensures, especially, that the bottom face in the diagram above is commutative.
2. By the construction of (k_C, k_G) and the morphism property of f' , f^* , g' and g^* , respectively, we get $f_C^*; k_C; dia^{\mathfrak{X}} = f_C^*; dia^{\mathfrak{D}}; \overline{k}_G$, and $g_C^*; k_C; dia^{\mathfrak{X}} = g_C^*; dia^{\mathfrak{D}}; \overline{k}_G$, thus the uniqueness of mediating morphisms for the top pushout entails $k_C; dia^{\mathfrak{X}} = dia^{\mathfrak{D}}; \overline{k}_G$. This means that the unique pair (k_C, k_G) defines indeed a specification morphism $k : \mathfrak{D} \rightarrow \mathfrak{X}$.

“ \Rightarrow ”: Pushouts in GSpec imply pushouts of its components in Graph and Set

Assume a pushout $\mathfrak{B} \xrightarrow{g^*} \mathfrak{C} \xleftarrow{f^*} \mathfrak{A}$ of a span $\mathfrak{B} \xleftarrow{f} \mathfrak{A} \xrightarrow{g} \mathfrak{C}$ of generalized specification morphisms:



As shown, we can also construct componentwise a pushout $\mathfrak{B} \xrightarrow{g^*} \mathfrak{D} \xleftarrow{f^*} \mathfrak{A}$ of the same span.

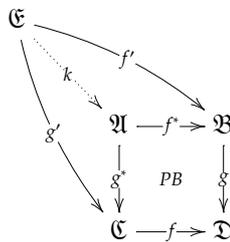
- \Rightarrow As pushouts are unique up to isomorphism, we get an isomorphism $i = (i_C, i_G) : \mathfrak{D} \rightarrow \mathfrak{C}$ in GSpec.
- \Rightarrow As composition and identities in GSpec are given by composition and identities in Set and Graph, respectively, i_C and i_G become isomorphisms in Set and Graph, respectively.
- \Rightarrow As pushouts are closed under isomorphisms, this means that the components of $\mathfrak{B} \xrightarrow{g^*} \mathfrak{C} \xleftarrow{f^*} \mathfrak{A}$ in Set and Graph also constitute pushouts in Set and Graph, respectively.

□

We have seen that a pushout in GSpec can be obtained by constructing a pushout in Graph for the underlying graph morphisms, as well as constructing a pushout in Set for the underlying mappings between sets of constraint identifiers. Now, we show that also pullbacks can be obtained in an analogue manner.

Proposition B.4.2 (Pullback of generalized specifications). *A pullback $\mathfrak{B} \xleftarrow{f^*} \mathfrak{A} \xrightarrow{g^*} \mathfrak{C}$ of a cospan $\mathfrak{B} \xrightarrow{g} \mathfrak{D} \xleftarrow{f} \mathfrak{C}$ in GSpec is obtained by constructing a pullback in Graph for the underlying graph morphisms and a pullback in Set for the underlying mappings between sets of constraint identifiers.*

PULLBACK OF
GENERALIZED
SPECIFICATIONS



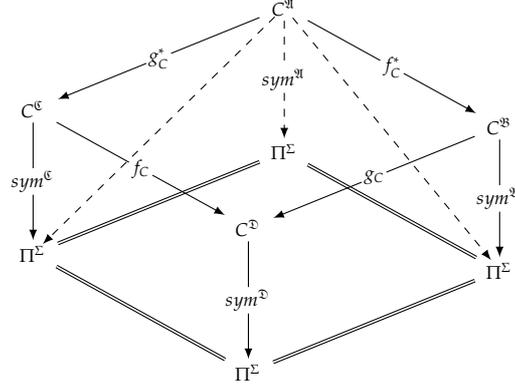
Lemma B.4.1. For any pullback $B \xleftarrow{f_C^*} A \xrightarrow{g_C^*} C$ of a cospan $B \xrightarrow{g_C} D \xleftarrow{f_C} C$ in **Graph** the span $(\text{Graph}/B)_0 \xleftarrow{\overline{f_C}} (\text{Graph}/A)_0 \xrightarrow{\overline{g_C}} (\text{Graph}/C)_0$ is a pullback in **Set** of the cospan $(\text{Graph}/B)_0 \xrightarrow{\overline{g_C}} (\text{Graph}/D)_0 \xleftarrow{\overline{f_C}} (\text{Graph}/C)_0$.

Proof.

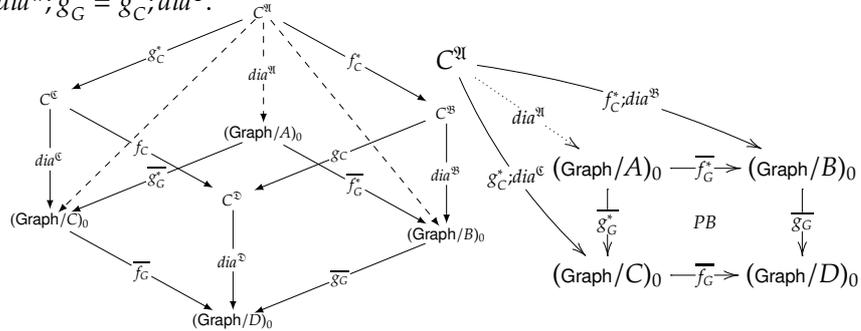
“ \Leftarrow ”: **Componentwise pullbacks in Graph and Set provide pullbacks in GSpec**

We consider the pullback $C^{\mathfrak{B}} \xleftarrow{f_C^*} C^{\mathfrak{A}} \xrightarrow{g_C^*} C^{\mathfrak{C}}$ of the cospan $C^{\mathfrak{B}} \xrightarrow{g_C} C^{\mathfrak{D}} \xleftarrow{f_C} C^{\mathfrak{C}}$ in **Set** and the pullback $B \xleftarrow{f_C^*} A \xrightarrow{g_C^*} C$ of the cospan $B \xrightarrow{g_C} D \xleftarrow{f_C} C$ in **Graph**.

Existence of symbol map: as (f_C, f_G) and (g_C, g_G) are specification morphisms, we have $f_C^*; \text{sym}^{\mathfrak{B}} = g_C^*; \text{sym}^{\mathfrak{C}}$, thus there exists a unique map $\text{sym}^{\mathfrak{A}} : C^{\mathfrak{A}} \rightarrow \Pi^{\Sigma}$ with $f_C^*; \text{sym}^{\mathfrak{B}} = \text{sym}^{\mathfrak{A}}$ and $g_C^*; \text{sym}^{\mathfrak{C}} = \text{sym}^{\mathfrak{A}}$.

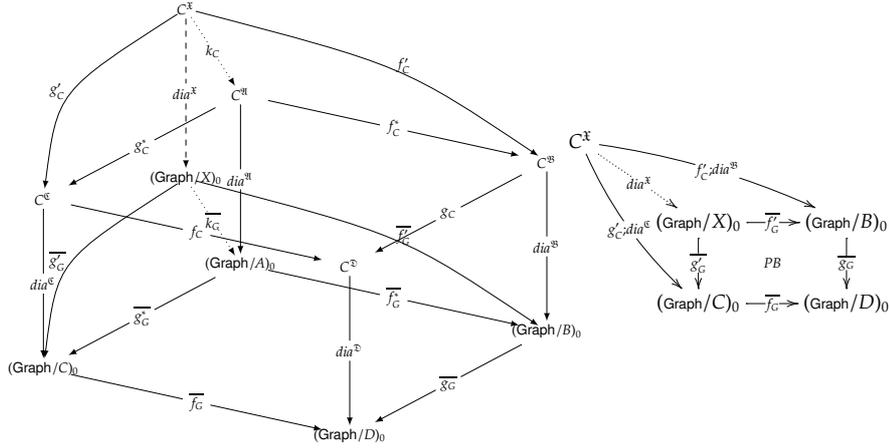


Existence of diagram map: the top face of the cube in the figure below shows the pullback for constraints in **Set**, while the pullback according to Lemma B.4.1 is shown in the bottom face. Analogously to Proposition B.2.2, we obtain $g_C^*; \text{dia}^{\mathfrak{C}}; \overline{f_G} = f_C^*; \text{dia}^{\mathfrak{B}}; \overline{g_G}$, thus there exists a unique map $\text{dia}^{\mathfrak{A}} : C^{\mathfrak{A}} \rightarrow (\text{Graph}/S)_0$ with $\text{dia}^{\mathfrak{A}}; \overline{f_G} = f_C^*; \text{dia}^{\mathfrak{B}}$ and $\text{dia}^{\mathfrak{A}}; \overline{g_G} = g_C^*; \text{dia}^{\mathfrak{C}}$.



Morphism property: the equations above ensure that the pair (f_C^*, f_G^*) defines a specification morphism $f^* : \mathfrak{A} \rightarrow \mathfrak{B}$ and that the pair (g_C^*, g_G^*) defines a specification morphism $g^* : \mathfrak{A} \rightarrow \mathfrak{C}$, respectively.

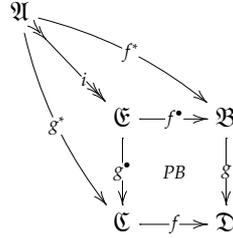
Universal property: assume a generalized specification \mathfrak{X} and two generalized specification morphisms $f' : \mathfrak{X} \rightarrow \mathfrak{B}$ and $g' : \mathfrak{X} \rightarrow \mathfrak{C}$ so that $g'; f = f'; g$ (see figures below).



1. Due to the pullback in **Set**, there is a unique mediating mapping k_C with $k_C; f_C^* = f'_C$, $k_C; g_C^* = g'_C$ and, due to the pullback in **Graph**, there is a unique graph morphism k_G with $k_G; f_G^* = f'_G$, $k_G; g_G^* = g'_G$. This ensures, especially, that the bottom face in the diagram above is commutative.
2. By the construction of (k_C, k_G) and the morphism property of f' , f^* , g' and g^* , respectively, we get $k_C; dia^{\mathfrak{A}}; \overline{f_G^*} = dia^{\mathfrak{X}}; \overline{k_G}; \overline{f_G^*}$, and $k_C; dia^{\mathfrak{A}}; \overline{g_G^*} = dia^{\mathfrak{X}}; \overline{k_G}; \overline{g_G^*}$, thus the uniqueness of mediating morphisms for the bottom pullback entails $k_C; dia^{\mathfrak{A}} = dia^{\mathfrak{X}}; \overline{k_G}$. This means that the unique pair (k_C, k_G) defines indeed a specification morphism $k : \mathfrak{X} \rightarrow \mathfrak{A}$.

" \Rightarrow ": Pullbacks in GSpec imply pullbacks of its components in Graph and Set

Assume a pullback $\mathfrak{B} \xleftarrow{f^*} \mathfrak{C} \xrightarrow{g^*} \mathfrak{D}$ of a cospan $\mathfrak{B} \xrightarrow{g} \mathfrak{D} \xleftarrow{f} \mathfrak{C}$ of generalized specification morphisms:



As shown, we can also construct componentwise a pullback $\mathfrak{B} \xleftarrow{f^*} \mathfrak{A} \xrightarrow{g^*} \mathfrak{C}$ of the same cospan.

- \implies As pullbacks are unique up to isomorphism, we get an isomorphism $i = (i_C, i_B) : \mathfrak{A} \rightarrow \mathfrak{C}$ in \mathbf{GSpec} .
- \implies As composition and identities in \mathbf{GSpec} are given by composition and identities in \mathbf{Set} and \mathbf{Graph} , respectively, i_C and i_B become isomorphisms in \mathbf{Set} and \mathbf{Graph} , respectively.
- \implies As pullbacks are closed under isomorphisms, this means that the components of $\mathfrak{B} \xleftarrow{f^*} \mathfrak{C} \xrightarrow{g^*} \mathfrak{C}$ in \mathbf{Set} and \mathbf{Graph} also constitute pushouts in \mathbf{Set} and \mathbf{Graph} , respectively.

□

B.4.2 Van Kampen Property in \mathbf{GSpec}

In the following, we consider the VK property for \mathbf{GSpec} . Because this time the construction of pushouts does not rely on a union construction anymore, we expect the category to be adhesive.

MONOMORPHISM
IN (GENERALIZED)
SPECIFICATIONS

Corollary B.4.1 (Monomorphism in (generalized) specifications). *Monomorphisms in \mathbf{GSpec} are given by componentwise monomorphisms in \mathbf{Set} respectively \mathbf{Graph} i.e. a specification morphism $f = (f_C, f_B) : \mathfrak{A} \rightarrow \mathfrak{B}$ is a monomorphism iff $f_C : C^{\mathfrak{A}} \rightarrow C^{\mathfrak{B}}$ is a monomorphism in \mathbf{Set} and $f_B : A \rightarrow B$ is a monomorphism in \mathbf{Graph} .*

Proof. Because the identities in \mathbf{GSpec} are componentwise identities, we get by Lemma B.2.1, Proposition B.4.2 and again Lemma B.2.1 for any generalized specification morphism in $f : \mathfrak{A} \rightarrow \mathfrak{B}$:

- f is a monomorphism in \mathbf{GSpec}
- $\iff \mathfrak{A} \xleftarrow{id^{\mathfrak{A}}} \mathfrak{A} \xrightarrow{id^{\mathfrak{A}}} \mathfrak{A}$ is the pullback of $\mathfrak{A} \xrightarrow{f} \mathfrak{B} \xleftarrow{f} \mathfrak{A}$
- $\iff A \xleftarrow{id^A} A \xrightarrow{id^A} A$ is the pullback of $A \xrightarrow{f_C} B \xleftarrow{f_C} A$ in \mathbf{Graph} and $C^{\mathfrak{A}} \xleftarrow{id^{C^{\mathfrak{A}}}} C^{\mathfrak{A}} \xrightarrow{id^{C^{\mathfrak{A}}}} C^{\mathfrak{A}}$ is the pullback of $C^{\mathfrak{A}} \xrightarrow{f_C} C^{\mathfrak{B}} \xleftarrow{f_C} C^{\mathfrak{A}}$ in \mathbf{Set}
- $\iff f_C : A \rightarrow B$ is a monomorphism in \mathbf{Graph} and $f_C : C^{\mathfrak{A}} \rightarrow C^{\mathfrak{B}}$ is a monomorphism in \mathbf{Set}

□

Corollary B.4.2 (Epimorphism in (generalized) specifications). *Epimorphisms in \mathbf{GSpec} are given by componentwise monomorphisms in \mathbf{Set} , respectively \mathbf{Graph} i.e. a specification morphism $f = (f_C, f_G) : \mathfrak{A} \rightarrow \mathfrak{B}$ is an epimorphism iff $f_C : C^{\mathfrak{A}} \rightarrow C^{\mathfrak{B}}$ in \mathbf{Set} and $f_G : A \rightarrow B$ is an epimorphism in \mathbf{Graph} .*

Proof. Because the identities in \mathbf{GSpec} are componentwise identities, we get by Lemma B.2.2, Proposition B.4.1 and again Lemma B.2.2 for any generalized specification morphism in $f : \mathfrak{A} \rightarrow \mathfrak{B}$:

$$\begin{aligned}
 & f \text{ is an epimorphism in } \mathbf{GSpec} \\
 \iff & \mathfrak{B} \xrightarrow{id^{\mathfrak{B}}} \mathfrak{B} \xleftarrow{id^{\mathfrak{B}}} \mathfrak{B} \text{ is the pushout of } \mathfrak{B} \xleftarrow{f} \mathfrak{A} \xrightarrow{f} \mathfrak{B} \\
 \iff & B \xrightarrow{id^B} B \xleftarrow{id^B} B \text{ is the pushout of } B \xleftarrow{f_G} A \xrightarrow{f_G} B \text{ in } \mathbf{Graph} \text{ and} \\
 & C^{\mathfrak{B}} \xrightarrow{id^{C^{\mathfrak{B}}}} C^{\mathfrak{B}} \xleftarrow{id^{C^{\mathfrak{B}}}} C^{\mathfrak{B}} \text{ is the pushout of } C^{\mathfrak{B}} \xleftarrow{f_C} C^{\mathfrak{A}} \xrightarrow{f_C} C^{\mathfrak{B}} \text{ in } \mathbf{Set} \\
 \iff & f_G : A \rightarrow B \text{ is an epimorphism in } \mathbf{Graph} \text{ and } f_C : C^{\mathfrak{A}} \rightarrow C^{\mathfrak{B}} \text{ is an} \\
 & \text{epimorphism in } \mathbf{Set}
 \end{aligned}$$

□

Considering all facts about the category of generalize DPF specification \mathbf{GSpec} , we can conclude that it is adhesive [36] and that it inherits adhesiveness from category \mathbf{Set} .

Proposition B.4.3. *Pushouts along monomorphism are VK squares in \mathbf{GSpec} .*

Proof. Assume a commutative cube (2) in \mathbf{GSpec} (See Definition 4.1.1). Because composition of generalized specification morphisms is defined componentwise, we have (2) commutes in \mathbf{GSpec} iff the corresponding two cubes for constraints commute in \mathbf{Set} and for graphs commute in \mathbf{Graph} ³. We assume the top face in (2) is a pushout and both back faces in (2) are pullbacks. By Proposition B.4.1 and Proposition B.4.2 follows:

$$\begin{aligned}
 \text{top face in (1) is a pushout} & \iff \text{top face in (1) for } (1_C) \text{ and } (1_G) \\
 & \text{are pushouts and} \\
 \text{back faces in (2) are} & \iff \text{back faces in (2) for } (2_C) \text{ and} \\
 \text{pullbacks} & \text{ } (2_G) \text{ are pullbacks}
 \end{aligned}$$

We have by Proposition B.2.1, Proposition B.2.2 and \mathbf{Set} is adhesive:

$$\begin{aligned}
 \text{bottom face in (2) is pushout} & \iff \text{bottom faces in } (2_C) \text{ and } (2_G) \\
 & \text{are pushouts} \\
 & \iff \text{front faces in } (2_C) \text{ and } (2_G) \text{ are} \\
 & \text{pullbacks} \\
 & \iff \text{front faces in (2) are pullbacks}
 \end{aligned}$$

□

³That implies that the corresponding cubes for vertices and edges commute in \mathbf{Set} .

Let us review Figure B.4, which shows a counterexample for the VK property in Spec . The cube in the figure is not a valid VK cube even though the top and bottom faces are pushouts and both back faces are pullbacks. Figure B.6 shows the analog example for GSpec . This time we also have to map the constraints, as we can have more than one constraint with the same predicate symbol and the same arity mapping. In contrast to the earlier example, we get two constraints in the upper pushout graph. Therefore, we can correctly trace back the constraints in the right front face of the cube that shows in contrast to Figure B.4 a valid pullback diagram. Hence, Figure B.6 shows a valid VK cube.

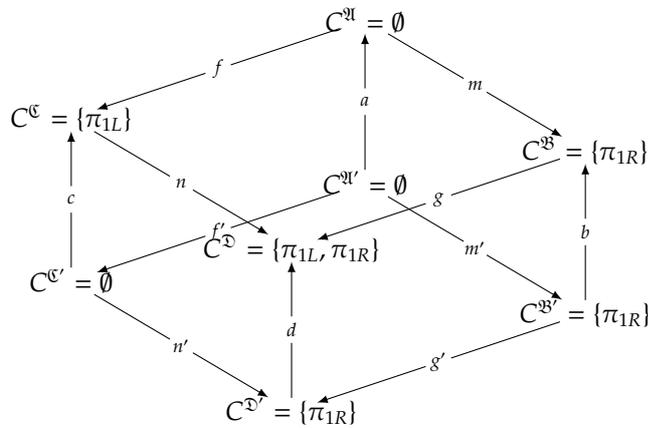


Figure B.6: Example VK property in GSpec

B.5 Conclusion

In this chapter, it has been examined how DPF's category of specifications fits into the context of adhesive categories. First, we showed that the concept of DPF specification, as it has been defined and used up in [33, 120, 121], does not provide an adhesive category of specifications. Then, a generalization of the concept of specification has been presented that provides an adhesive category GSpec . Furthermore, we showed that this generalization step is analogous to the step going from the category SGraph of simple directed graphs to the category Graph of directed multi-graphs. This should give the reader an intuitive understanding about adhesiveness, elements must be able to be traced.

Because category GSpec is adhesive, it fits into the presented framework of coupled transformations. However, here we have to note that

category Spec already fits into it. The reason is that DPFs atomic constraints are only used on the meta-model level or model level, i.e. in particular that the typing morphism are graph morphisms and not specification morphisms. Therefore, the adhesiveness of the underlying category is sufficient to apply the DPF framework in the context of the coupled transformation framework presented in this thesis.

In this chapter, the definition of both categories Spec and GSpec is based on the category Graph . Instead of Graph , it is possible to use other base categories Base [33] to define corresponding categories Spec and GSpec . There should not be any problems in generalizing the concepts and results presented to arbitrary adhesive base categories Base . In addition, in this general case, the step from category Spec to category GSpec has the effect that GSpec is adhesive while Spec is not. An analog result we should get for adhesive HLR categories. If we also get an analog result in case of a weak adhesive category as base category is a question for future research.

Bibliography

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, USA, 2nd edition, 2006. ISBN 0321486811.
- [2] AndromDA. *Project Web Site*, 2014. <http://www.andromda.org>.
- [3] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformation. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings of MoDELS 2010: 13th International Conference on Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2010. ISBN 978-3-642-16144-5. DOI [10.1007/978-3-642-16145-2_9](https://doi.org/10.1007/978-3-642-16145-2_9).
- [4] Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, 2003. DOI [10.1109/MS.2003.1231149](https://doi.org/10.1109/MS.2003.1231149).
- [5] Colin Atkinson, Ralph Gerbig, and Bastian Kennel. On-the-Fly Emendation of Multi-level Models. In Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitrios S. Kolovos, editors, *Proceedings of ECMFA 2012: 8th European Conference on Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 194–209. Springer, 2012. DOI [10.1007/978-3-642-31491-9_16](https://doi.org/10.1007/978-3-642-31491-9_16).
- [6] Atlas Transformation Language. *User Guide*, 2009. http://wiki.eclipse.org/ATL/User_Guide.
- [7] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *Electronic Communications of the EASST*, 1, 2006.
- [8] Herman Balsters, Bert O. de Brock, and Stefan Conrad, editors. *Proceedings of FoMLaDO 2000: 9th International Workshop on Foundations*

- of Models and Languages for Data and Objects*. Lecture Notes in Computer Science. Springer, September 2001. ISBN 9783540422723.
- [9] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of SIGMOD 1987: 11th ACM International Conference on Management of Data*, pages 311–322. ACM, 1987.
- [10] Richard Barker. *Case*Method: Entity Relationship Modelling*. Addison-Wesley Professional, 1990. ISBN 978-0-201-41696-1.
- [11] Michael Barr and Charles Wells. *Category Theory for Computing Science (3rd Edition)*. Les Publications CRM, Montreal, 1999. ISBN 2921120313.
- [12] Daniela Berardi, Andrea Cali, Diego Calvanese, and Giuseppe Di Giacomo. Reasoning on UML Class Diagrams. *Artificial Intelligence*, 168:70–118, 2005.
- [13] Philip A. Bernstein, Todd J. Green, Sergey Melnik, and Alan Nash. Implementing Mapping Composition. *International Journal on Very Large Data Bases*, 17(2):333–353, 2008.
- [14] Jean Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005. DOI [10.1007/s10270-005-0079-0](https://doi.org/10.1007/s10270-005-0079-0).
- [15] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of ASE 2001: 16th IEEE International Conference on Automated Software Engineering*, pages 273–280, 2001. ISBN 978-0-7695-1426-0. DOI [10.1109/ASE.2001.989813](https://doi.org/10.1109/ASE.2001.989813).
- [16] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *Software and Systems Modeling*, 11(2):227–250, 2012. DOI [10.1007/s10270-011-0199-7](https://doi.org/10.1007/s10270-011-0199-7).
- [17] Paul Boehm, Harald-Reto Fonio, and Annegret Habel. Amalgamation of Graph Transformations: A Synchronization Mechanism. *Journal of Computer and System Sciences*, 34(2-3):377–408, June 1987. ISSN 0022-0000. DOI [10.1016/0022-0000\(87\)90030-4](https://doi.org/10.1016/0022-0000(87)90030-4).
- [18] Marco Cadoli, Diego Calvanese, and Toni Mancini. Finite satisfiability of UML class diagrams by Constraint Programming. In *Proceedings of DL 2004: 1st International Workshop on Description Logics*, volume 104. CEUR-WS.org, 2004.

-
- [19] Noam Chomsky. Three Models for the Description of Language. *IEEE Transactions on Information Theory*, 2(3):113–124, 1956. DOI [10.1109/TIT.1956.1056813](https://doi.org/10.1109/TIT.1956.1056813).
- [20] Noam Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2(2):137–167, 1959. DOI [10.1016/S0019-9958\(59\)90362-6](https://doi.org/10.1016/S0019-9958(59)90362-6).
- [21] Antonio Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, Department of Computer Science, University of L'Aquila, Italy, January 2008.
- [22] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proceedings of EDOC 2008: 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231. IEEE Computer Society, 2008. ISBN 978-0-7695-3373-5. DOI [10.1109/EDOC.2008.44](https://doi.org/10.1109/EDOC.2008.44).
- [23] Kajal T. Claypool, Jing Jin, and Elke A. Rundensteiner. SERF: Schema Evaluation through an Extensible, Re-usable and Flexible Framework. In *Proceedings of ACM CIKM 1998: 7th International Conference on Information and Knowledge Management*, pages 314–321. ACM, November 1998. ISBN 1-58113-061-9. DOI [10.1145/288627.288672](https://doi.org/10.1145/288627.288672).
- [24] Andrea Corradini, Tobias Heindel, Frank Hermann, and Barbara König. Sesqui-Pushout Rewriting. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Proceedings of ICGT 2006: 3rd International Conference on Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 30–45. Springer, September 2006. ISBN 3-540-38870-2. DOI [10.1007/11841883_4](https://doi.org/10.1007/11841883_4).
- [25] Carlo Curino, Hyun Jin Moon, MyungWon Ham, and Carlo Zaniolo. The PRISM Workwench: Database Schema Evolution without Tears. In Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors, *Proceedings of ICDE 1999: 25th International Conference on Data Engineering*, pages 1523–1526. Proceedings of ICDE 1999: 25th International Conference on Data Engineering, 2009. ISBN 978-0-7695-3545-6.
- [26] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Update Rewriting and Integrity Constraint Maintenance in a Schema Evolution Support System: PRISM++. *Proceedings of VLDB 2010: 36th International Conference on Very Large Database Endowment*, 4(2): 117–128, 2010.

BIBLIOGRAPHY

- [27] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the Database Schema Evolution Process. *International Journal on Very Large Data Bases*, 22(1):73–98, 2013.
- [28] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000. ISBN 978-0-201-30977-5.
- [29] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *2nd OOPSLA Workshop on Generative Techniques in the Context of MDA*, 2003.
- [30] Juan de Lara and Gabriele Taentzer. Automated Model Transformation and Its Validation Using AToM³ and AGG. In Alan F. Blackwell, Kim Marriott, and Atsushi Shimojima, editors, *Proceedings of Diagrams 2004: 3rd International Conference on Diagrammatic Representation and Inference*, volume 2980 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2004. ISBN 3-540-21268-X. DOI [10.1007/978-3-540-25931-2_18](https://doi.org/10.1007/978-3-540-25931-2_18).
- [31] Alin Deutsch and Val Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. *International Journal on Very Large Data Bases*, pages 201–212, 2003.
- [32] Zinovy Diskin. Generalized sketches as an algebraic graph-based framework for semantic modeling and database design. Technical Report 9701, University of Latvia, Riga, Latvia, August 1997.
- [33] Zinovy Diskin and Uwe Wolter. A Diagrammatic Logic for Object-Oriented Visual Modeling. In *Proceedings of ACCAT 2007: 2nd Workshop on Applied and Computational Category Theory*, volume 203/6 of *Electronic Notes in Theoretical Computer Science*, pages 19–41. Elsevier, 2008. DOI [10.1016/j.entcs.2008.10.041](https://doi.org/10.1016/j.entcs.2008.10.041).
- [34] Eclipse Modeling Framework. *Project Web Site*. <http://www.eclipse.org/emf/>.
- [35] Eclipse Xtext. *Xtext*. <http://www.eclipse.org/Xtext>.
- [36] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006. ISBN 978-3-540-31187-4. DOI [10.1007/3-540-31188-2](https://doi.org/10.1007/3-540-31188-2).
- [37] Hartmut Ehrig, Frank Hermann, and Ulrike Prange. Cospan DPO Approach: An Alternative for DPO Graph Transformation. *EATCS Bulletin*, 98:139–149, 2009. URL <http://www.eatcs.org/images/bulletin/beatcs98.pdf>.

-
- [38] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, USA, 6th edition, 2010. ISBN 0136086209,978-0136086208.
- [39] Claudia Ermel, Enrico Biermann, Johann Schmidt, and Angeline Warning. Visual Modeling of Controlled EMF Model Transformation using Henshin. *Electronic Communications of the EASST*, 32, 2010.
- [40] Ronald Fagin. Inverting Schema Mappings. *ACM Transactions on Database Systems*, 32(4), 2007.
- [41] Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran, and Joëlle Madec. Schema and Database Evolution in the O2 Object Database System. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *Proceedings of VLDB 1995: 21th International Conference on Very Large Databases*, pages 170–181. Morgan Kaufmann, September 1995. ISBN 1-55860-379-4.
- [42] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. ISBN 0321712943.
- [43] David S. Frankel and John Parodi. *The MDA Journal: Model Driven Architecture Straight From The Masters*. Meghan-Kiffer Press, 2004. ISBN 978-0-929652-25-2.
- [44] Fujaba Developer Team. *The Fujaba Tool Suite*. <http://www.fujaba.de/>.
- [45] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing Model Adaptation by Precise Detection of Metamodel Changes. In Richard F. Paige, Alan Hartman, and Arend Rensink, editors, *Proceedings of ECMDA-FA 2009: 5th European Conference on Model-Driven Architecture Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 34–49. Springer, June 2009. ISBN 978-3-642-02673-7. DOI [10.1007/978-3-642-02674-4_4](https://doi.org/10.1007/978-3-642-02674-4_4).
- [46] Pérez González, Alberto Carlos, Fabian Buettner, Robert Clarisó, and Jordi Cabot. EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *Proceedings of FormSERA 2012: 1st Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches*, pages 44–50, Zurich, Suisse, June 2012. IEEE Computer Society. ISBN 978-1-4673-1907-2. DOI [10.1109/FormSERA.2012.6229788](https://doi.org/10.1109/FormSERA.2012.6229788).
- [47] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for Software Engineering*. Wiley, 2008. ISBN 978-0-470-03036-3.
- [48] Graphical Modeling Framework. *Project Web Site*. <http://www.eclipse.org/modeling/gmp>.

- [49] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004. ISBN 978-0-471-20284-4.
- [50] GrGen.NET: transformation of structures made easy. *Project Web Site*. <http://grgen.net>.
- [51] Boris Gruschko, Dimitrios Kolovos, and Richard Paige. Towards Synchronizing Models with Evolving Metamodels. In Dalila Tamzalit, editor, *Proceedings of MoDSE 2007: 1st International Workshop on Model-Driven Software Evolution*, pages 1–9, March 2007.
- [52] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001. DOI [10.1017/S0960129501003425](https://doi.org/10.1017/S0960129501003425).
- [53] Kahina Hassam, Salah Sadou, Vincent Le Gloahec, and Régis Fleurquin. Assistance System for OCL Constraints Adaptation during Metamodel Evolution. In Tom Mens, Yiannis Kanellopoulos, and Andreas Winter, editors, *Proceedings of CSMR 2011: 15th European Conference on Software Maintenance and Reengineering*, pages 151–160. IEEE Computer Society, March 2011. ISBN 978-0-7695-4343-7. DOI [10.1109/CSMR.2011.21](https://doi.org/10.1109/CSMR.2011.21).
- [54] Frank Hermann, Hartmut Ehrig, and Claudia Ermel. Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks. In Marsha Chechik and Martin Wirsing, editors, *Proceedings of FASE 2009: 12th International Conference on Fundamental Approaches to Software Engineering*, volume 5503 of *Lecture Notes in Computer Science*, pages 325–339. Springer, 2009. ISBN 978-3-642-00592-3. DOI [10.1007/978-3-642-00593-0_22](https://doi.org/10.1007/978-3-642-00593-0_22).
- [55] Markus Herrmannsdoerfer and Daniel Ratiu. Limitations of Automating Model Migration in Response to Metamodel Adaptation. In Sudipto Ghosh, editor, *Proceedings of MoDELS 2009: 12th International Conference on Model Driven Engineering Languages and Systems*, volume 6002 of *Lecture Notes in Computer Science*, pages 205–219, 2009. ISBN 978-3-642-12260-6. DOI [10.1007/978-3-642-12261-3_20](https://doi.org/10.1007/978-3-642-12261-3_20).
- [56] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Jürgens. Automatability of Coupled Evolution of Metamodels and Models in Practice. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Proceedings of MoDELS 2008: 11th International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 645–659. Springer, 2008. ISBN 978-3-540-87874-2. DOI [10.1007/978-3-540-87875-9_45](https://doi.org/10.1007/978-3-540-87875-9_45).

-
- [57] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Jürgens. COPE - Automating Coupled Evolution of Metamodels and Models. In Sophia Drossopoulou, editor, *Proceedings of ECOOP 2009: 23rd European Conference on Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 52–76. Springer, 2009. ISBN 978-3-642-03012-3. DOI [10.1007/978-3-642-03013-0_4](https://doi.org/10.1007/978-3-642-03013-0_4).
- [58] Markus Herrmannsdoerfer, Sander Vermolen, and Guido Wachsmuth. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Proceedings of SLE 2010: 3rd International Conference on Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 163–182. Springer, 2010. ISBN 978-3-642-19439-9. DOI [10.1007/978-3-642-19440-5_10](https://doi.org/10.1007/978-3-642-19440-5_10).
- [59] Markus Herrmannsdörfer. *Evolutionary Metamodeling*. PhD thesis, Department of Informatics, Technical University Munich, Germany, 2011.
- [60] Wolfgang Hesse. More matters on (meta-)modelling: remarks on Thomas Kühne’s “matters”. *Software and Systems Modeling*, 5(4): 387–394, 2006. DOI [10.1007/s10270-006-0033-9](https://doi.org/10.1007/s10270-006-0033-9).
- [61] Tassilo Horn and Jürgen Ebert. The GReTL Transformation Language. In Jordi Cabot and Eelco Visser, editors, *Proceedings of ICMT 2011: 4rd International Conference on Theory and Practice of Model Transformation*, volume 6707 of *Lecture Notes in Computer Science*, pages 183–197. Springer, June 2011. ISBN 978-3-642-21731-9. DOI [10.1007/978-3-642-21732-6_13](https://doi.org/10.1007/978-3-642-21732-6_13).
- [62] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009. ISBN 978-3-642-00283-0. DOI [10.1007/b95112](https://doi.org/10.1007/b95112).
- [63] Peter T. Johnstone, Stephen Lack, and Pawel Sobocinski. Quasi-toposes, Quasiadhesive Categories and Artin Glueing. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *Proceedings of CALCO 2007: 2rd International Conference on Algebra and Coalgebra in Computer Science*, volume 4624 of *Lecture Notes in Computer Science*, pages 312–326. Springer, 2007. ISBN 978-3-540-73857-2. DOI [10.1007/978-3-540-73859-6_21](https://doi.org/10.1007/978-3-540-73859-6_21).
- [64] Frédéric Jouault and Jean Bézivin. KM3: A DSL for Metamodel Specification. In Roberto Gorrieri and Heike Wehrheim, editors, *Proceedings of FMOODS 2006: 8th International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of

- Lecture Notes in Computer Science*, pages 171–185. Springer, June 2006. ISBN 3-540-34893-X. DOI [10.1007/11768869_14](https://doi.org/10.1007/11768869_14).
- [65] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *Proceedings of ASE 2011: 26st IEEE/ACM International Conference on Automated Software Engineering*, pages 163–172. IEEE Computer Society, November 2011. DOI [10.1109/ASE.2011.6100050](https://doi.org/10.1109/ASE.2011.6100050).
- [66] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. Consistency-Preserving Edit Scripts in Model Versioning. In *Proceedings of ASE 2013: 28st IEEE/ACM International Conference on Automated Software Engineering*, November 2013.
- [67] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley, 2008. ISBN 978-0-470-03666-2.
- [68] Stuart Kent. Model Driven Engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Proceedings of IFM 2002: 3rd International Conference on Integrated Formal Methods*, Lecture Notes in Computer Science, pages 286–298. Springer, 2002. ISBN 978-3-540-43703-1. DOI [10.1007/3-540-47884-1_16](https://doi.org/10.1007/3-540-47884-1_16).
- [69] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Professional, 2003. ISBN 978-0-321-19442-8.
- [70] Harald König. Trouble with Wrong Adjoints. Technical Report 02012/05, Fakultät für Informatik und Wirtschaftsinformatik, Fachhochschule für die Wirtschaft Hannover, Germany, 2012. <http://fhdwdev.ha.bib.de/cgi-bin/forschungsreihe>.
- [71] Harald König, Michael Löwe, and Christoph Schulz. Model Transformation and Induced Instance Migration: A Universal Framework. In Adenilso da Silva Simão and Carroll Morgan, editors, *Proceedings of SBMF 2011: 14th Brazilian Symposium on Formal Methods, Foundations and Applications*, volume 7021 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2011. ISBN 978-3-642-25031-6. DOI [10.1007/978-3-642-25032-3_1](https://doi.org/10.1007/978-3-642-25032-3_1).
- [72] Harald König, Uwe Wolter, and Michael Löwe. Characterizing Van Kampen Squares via Descent Data. In Ulrike Golas and Thomas Soboll, editors, *Proceedings of ACCAT 2012: 7nd Workshop on Applied and Computational Category Theory*, volume 93 of *Electronic Proceedings in Theoretical Computer Science*, pages 61–81. Elsevier, 2012. DOI [10.4204/EPTCS.93.4](https://doi.org/10.4204/EPTCS.93.4).

-
- [73] Christian Krause, Johannes Dyck, and Holger Giese. Metamodel-Specific Coupled Evolution Based on Dynamically Typed Graph Transformations. In Keith Duddy and Gerti Kappel, editors, *Proceedings of ICMT 2013: 6rd International Conference on Theory and Practice of Model Transformation*, volume 7909 of *Lecture Notes in Computer Science*, pages 76–91. Springer, June 2013.
- [74] Thomas Kühne. Matters of (meta-)modeling. *Software and Systems Modeling*, 5(4):369–385, 2006. DOI [10.1007/s10270-006-0017-9](https://doi.org/10.1007/s10270-006-0017-9).
- [75] Thomas Kühne. Clarifying matters of (meta-)modeling: an author’s reply. *Software and Systems Modeling*, 5(4):395–401, 2006. DOI [10.1007/s10270-006-0034-8](https://doi.org/10.1007/s10270-006-0034-8).
- [76] Ivan Kurtev, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Model-Based DSL Frameworks. In *Proceedings of OOPSLA 2006: 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 602–616. ACM, 2006. ISBN 978-1-59593-491-8. DOI [10.1145/1176617.1176632](https://doi.org/10.1145/1176617.1176632).
- [77] Stephen Lack and Pawel Sobocinski. Adhesive Categories. In Igor Walukiewicz, editor, *Proceedings of FoSSaCS 2004: 7th Foundations of Software Science and Computation Structures*, volume 2987 of *Lecture Notes in Computer Science*, pages 273–288, 2004. ISBN 3-540-21298-1. DOI [10.1007/978-3-540-24727-2_20](https://doi.org/10.1007/978-3-540-24727-2_20).
- [78] Stephen Lack and Pawel Sobocinski. Toposes Are Adhesive. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Proceedings of ICGT 2006: 3rd International Conference on Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, pages 184–198. Springer, September 2006. ISBN 3-540-38870-2. DOI [10.1007/11841883_14](https://doi.org/10.1007/11841883_14).
- [79] Ralf Lämmel. Grammar Adaptation. In José Nuno Oliveira and Pamela Zave, editors, *Proceedings of FME 2001: Formal Methods for Increasing Software Productivity: 1st International Symposium of Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 550–570. Springer, 2001. ISBN 3-540-41791-5. DOI [10.1007/3-540-45251-6_32](https://doi.org/10.1007/3-540-45251-6_32).
- [80] Yngve Lamo, Florian Mantz, Adrian Rutle, and Juan de Lara. A declarative and bidirectional model transformation approach based on graph co-spans. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming, PPDP '13*, pages 1–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2154-9. DOI [10.1145/2505879.2505900](https://doi.org/10.1145/2505879.2505900).

BIBLIOGRAPHY

- [81] Yngve Lamo, Xiaoliang Wang, Florian Mantz, Øyvind Bech, Anders Sandven, and Adrian Rutle. DPF Workbench: A Multi-Level Language Workbench for MDE. *Proceedings of the Estonian Academy of Sciences*, 62:3–15, March 2013. DOI [10.3176/proc.2013.1.02](https://doi.org/10.3176/proc.2013.1.02).
- [82] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. <http://leeseshia.org>, 2011.
- [83] Maurizio Lenzerini and Paolo Nobili. On the satisfiability of dependency constraints in entity-relationship schemata. *Information Systems*, 15(4):453–461, 1990. ISSN 0306-4379. DOI [10.1016/0306-4379\(90\)90048-T](https://doi.org/10.1016/0306-4379(90)90048-T).
- [84] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, Feng Shi, Chris Buskirk, and Gabor Karsai. A semi-formal description of migrating domain-specific models with evolving domains. *Software and Systems Modeling*, pages 1–17, January 2013. ISSN 1619–1366. DOI [10.1007/s10270-012-0313-5](https://doi.org/10.1007/s10270-012-0313-5).
- [85] Xue Li. A Survey of Schema Evolution in Object-Oriented Databases. In *Proceedings of TOOLS 1999: 31st International Conference on Objects, Components, Models and Patterns*, pages 362–371. IEEE Computer Society, 1999. ISBN 0-7695-0393-4.
- [86] Peter Linz. *An Introduction to Formal Language and Automata*. Jones and Bartlett Publishers, Inc., USA, 2006. ISBN 0763737984.
- [87] Michael Löwe. Algebraic Approach to Single-Pushout Graph Transformation. *Theoretical Computer Science*, 109(1&2):181–224, 1993. DOI [10.1016/0304-3975\(93\)90068-5](https://doi.org/10.1016/0304-3975(93)90068-5).
- [88] Michael Löwe. Graph Rewriting in Span-Categories. Technical Report 02010/02, Fakultät für Informatik und Wirtschaftsinformatik, Fachhochschule für die Wirtschaft Hannover, Germany, 2010. <http://fhdwdev.ha.bib.de/cgi-bin/forschungsreihe>.
- [89] Michael Löwe, Harald König, Christoph Schulz, and Marius Schultchen. Algebraic Graph Transformations with Inheritance. Technical report, Fakultät für Informatik und Wirtschaftsinformatik, Fachhochschule für die Wirtschaft Hannover, Germany, March 2013. <http://fhdwdev.ha.bib.de/cgi-bin/forschungsreihe>.
- [90] Florian Mantz. Syntactic Quality Assurance Techniques for Software Models. Diploma thesis, Department of Mathematics and Informatics, Philipps University in Marburg, Germany, August 2009.

-
- [91] Florian Mantz and Uwe Wolter. The Advantage of Using Co-span Graph Transformations for Meta-model Evolution. In *Proceedings of NWPT 2013: 25th Nordic Workshop on Programming Theory*, pages 1–4, November 2013.
- [92] Florian Mantz, Alessandro Rossini, Adrian Rutle, Yngve Lamo, and Uwe Wolter. Towards a Formal Approach to Metamodel Evolution. In *Proceedings of NWPT 2010: 22nd Nordic Workshop on Programming Theory*, pages 52–54, November 2010. ISBN 978-952-12-2478-2.
- [93] Florian Mantz, Stefan Jurack, and Gabriele Taentzer. Graph Transformation Concepts for Meta-Model Evolution Guaranteeing Permanent Type conformance Throughout Model Migration. In Andy Schürr, Dániel Varró, and Gergely Varró, editors, *Proceedings of AGTIVE 2011: 4th International Symposium on Applications of Graph Transformations with Industrial Relevance*, volume 7233 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2011. ISBN 978-3-642-34175-5. DOI [/10.1007/978-3-642-34176-2_3](https://doi.org/10.1007/978-3-642-34176-2_3).
- [94] Florian Mantz, Gabriele Taentzer, and Yngve Lamo. Co-Transformation of Type and Instance Graphs Supporting Merging of Types with Retyping. In *Proceedings of GCM 2012: 4th International Workshop on Graph Computation Models*, pages 47–58, September 2012. <http://gcm2012.imag.fr/proceedingsGCM2012.pdf>.
- [95] Florian Mantz, Gabriele Taentzer, and Yngve Lamo. Well-formed Model Co-evolution with Customizable Model Migration. *Electronic Communications of the EASST*, 58:1–13, March 2013. ISSN 1863-2122.
- [96] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. DOI [10.1016/j.entcs.2005.10.021](https://doi.org/10.1016/j.entcs.2005.10.021).
- [97] Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. *Science of Computer Programming*, 76(12): 1223–1246, 2011. DOI [10.1016/j.scico.2011.01.002](https://doi.org/10.1016/j.scico.2011.01.002).
- [98] Bart Meyers, Manuel Wimmer, Antonio Cicchetti, and Jonathan Sprinkle. A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST*, 42, 2011.
- [99] MOLA - MOdel transformation LAnguage. *Project Web Site*. <http://mola.mii.lu.lv>.
- [100] Alan Nash, Philip A. Bernstein, and Sergey Melnik. Composition of Mappings Given by Embedded Dependencies. *ACM Transactions on Database Systems*, 32(1):4, 2007.

BIBLIOGRAPHY

- [101] Isaac Nassi and Ben Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, August 1973. ISSN 0362-1340. DOI [10.1145/953349.953350](https://doi.org/10.1145/953349.953350).
- [102] OASIS. *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. Organization for the Advancement of Structured Information Standards (OASIS), April 2007.
- [103] Object Management Group. *Web site*. <http://www.omg.org>.
- [104] Object Management Group. *MDA Guide*, June 2003. <http://www.omg.org/cgi-bin/doc?omg/03-06-01>.
- [105] Object Management Group. *Object Constraint Language Specification*, February 2010. <http://www.omg.org/spec/OCL/2.2/>.
- [106] Object Management Group. *Query/View/Transformation Specification*, January 2011. <http://www.omg.org/spec/QVT/1.1>.
- [107] Object Management Group. *Unified Modeling Language Specification*, August 2011. <http://www.omg.org/spec/UML/2.4.1/>.
- [108] Object Management Group. *XML Metadata Interchange Specification*, August 2011. <http://www.omg.org/spec/XMI/2.4.1/>.
- [109] Object Management Group. *Object Constraint Language Specification*, January 2012. <http://www.omg.org/spec/OCL/2.3.1/>.
- [110] Object Management Group. *Meta-Object Facility Specification*, June 2013. <http://www.omg.org/spec/MOF/2.4.1/>.
- [111] OMG Model Driven Architecture. *Web Site*. <http://www.omg.org/mda/>.
- [112] Fernando Orejas and Leen Lambers. Symbolic Attributed Graphs for Attributed Graph Transformation. *Electronic Communications of the EASST*, 30:1–25, February 2010. ISSN 1863-2122.
- [113] Fernando Orejas, Hartmut Ehrig, and Ulrike Prange. Reasoning with graph constraints. *Formal Aspects of Computing*, 22(3-4):385–422, 2010. DOI [10.1007/s00165-009-0116-9](https://doi.org/10.1007/s00165-009-0116-9).
- [114] PETE: Eclipse Prolog EMF Transformation Engine. *Project Web Site*. <http://www4.informatik.tu-muenchen.de/~schaetz/PETE/PETEFrame.html>.

-
- [115] Randal J. Peters and Ken Barker. Change Propagation in an Axiomatic Model of Schema Evolution for Objectbase Management Systems. In Herman Balsters, Bert O. de Brock, and Stefan Conrad, editors, *Proceedings of FoMLaDO 2000: 9th International Workshop on Foundations of Models and Languages for Data and Objects*, Lecture Notes in Computer Science, pages 142–162. Springer, September 2000. ISBN 9783540422723.
- [116] John F. Roddick. Schema Evolution in Database Systems - An Annotated Bibliography. *SIGMOD Record*, 21(4):35–40, 1992.
- [117] Louis M. Rose, Markus Herrmannsdoerfer, James R. Williams, Dimitrios S. Kolovos, Kelly Garcés, Richard F. Paige, and Fiona A. C. Polack. A Comparison of Model Migration Tools. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Proceedings of MoDELS 2010: 13th International Conference on Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2010. ISBN 978-3-642-16144-5. DOI [10.1007/978-3-642-16145-2_5](https://doi.org/10.1007/978-3-642-16145-2_5).
- [118] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model Migration with Epsilon Flock. In Laurence Tratt and Martin Gogolla, editors, *Proceedings of ICMT 2010: 3rd International Conference on Theory and Practice of Model Transformation*, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2010. ISBN 978-3-642-13687-0. DOI [10.1007/978-3-642-13688-7_13](https://doi.org/10.1007/978-3-642-13688-7_13).
- [119] Louis M. Rose, Markus Herrmannsdoerfer, Steffen Mazanek, Pieter Van Gorp, Sebastian Buchwald, Tassilo Horn, Elina Kalnina, Andreas Koch, Kevin Lano, Bernhard Schätz, and Manuel Wimmer. Graph and Model Transformation Tools for Model Migration: Empirical Results from the Transformation Tool Contest. *Software and Systems Modeling*, tbd, 2012. DOI [10.1007/s10270-012-0245-0](https://doi.org/10.1007/s10270-012-0245-0).
- [120] Alessandro Rossini. *Diagram Predicate Framework meets Model Versioning and Deep Metamodelling*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2011.
- [121] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.
- [122] Christoph Schulz, Michael Löwe, and Harald König. Composition of Model Transformations: A Categorical Framework. In Rohit Gheyi and David A. Naumann, editors, *Proceedings of SBMF 2012: 15th*

- Brazilian Symposium on Formal Methods, Foundations and Applications*, volume 7498 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2012. DOI [10.1007/978-3-642-33296-8_13](https://doi.org/10.1007/978-3-642-33296-8_13).
- [123] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Proceedings of WG 1994: 20th 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](https://doi.org/10.1007/3-540-59071-4_45).
- [124] Ed Seidewitz. What Models Mean. *IEEE Software*, 20(5):26–32, 2003. DOI [10.1109/MS.2003.1231147](https://doi.org/10.1109/MS.2003.1231147).
- [125] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.
- [126] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, 9th edition, 2012. ISBN 1118063333,978-1118063330.
- [127] Jonathan Sprinkle and Gabor Karsai. A Domain-Specific Visual Language for Domain Model Evolution. *Journal of Visual Languages and Computing*, 15(3–4):291–307, 2004. DOI [10.1016/j.jvlc.2004.01.006](https://doi.org/10.1016/j.jvlc.2004.01.006).
- [128] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.
- [129] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006. ISBN 978-0-470-02570-3.
- [130] A. Standards and Standards Association of Australia. *Information Processing - Documentation Symbols and Conventions for Data, Program and System Flowcharts, Program Network Charts and System Resources Charts*. Australian standard. Standards Australia, 1987. ISBN 9780726247279.
- [131] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0 (2nd Edition)*. Addison-Wesley Professional, 2008. ISBN 978-0-321-33188-5.
- [132] Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. Challenges in Model-Driven Software Engineering. In Michel R. V. Chaudron, editor, *Proceedings of MoDELS 2008: 11th International Conference on Model Driven Engineering Languages and Systems*, volume 5421 of *Lecture Notes in Computer Science*, pages 35–47. Springer, September 2008. ISBN 978-3-642-01647-9. DOI [10.1007/978-3-642-01648-6_4](https://doi.org/10.1007/978-3-642-01648-6_4).

-
- [133] Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, Technical University of Berlin, 1996.
- [134] Gabriele Taentzer. Instance Generation from Type Graphs with Arbitrary Multiplicities. *Electronic Communications of the EASST*, 47, 2012.
- [135] Gabriele Taentzer and Martin Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG - an Algebraic Graph Grammar System. In Hans Jürgen Schneider and Hartmut Ehrig, editors, *Dagstuhl Seminar on Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 1994. ISBN 3-540-57787-4. DOI [10.1007/3-540-57787-4_24](https://doi.org/10.1007/3-540-57787-4_24).
- [136] Gabriele Taentzer, Florian Mantz, and Yngve Lamo. Co-Transformation of Graphs and Type Graphs With Application to Model Co-Evolution. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proceedings of ICGT 2012: 6nd International Conference on Graph Transformations*, volume 7562 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2012. ISBN 978-3-642-33653-9.
- [137] Gabriele Taentzer, Florian Mantz, Thorsten Arendt, and Yngve Lamo. Customizable Model Migration Schemes for Meta-model Evolutions with Multiplicity Changes. In *Proceedings of MoDELS 2013: 16th International Conference on Model Driven Engineering Languages and Systems*, *Lecture Notes in Computer Science*, pages 254–270. Springer, 2013.
- [138] The EMF Henshin Transformation Tool. *Project Web Site*, 2010. <http://www.eclipse.org/modeling/emft/henshin/>.
- [139] The Entity MetaEdit+ Workbench. *Project Web Site*. <http://www.metacase.com/mep/>.
- [140] UML RSDS Model Transformation and Model-Driven Development Tools. *Project Web Site*. <http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/>.
- [141] Mark van den Brand, Zvezdan Protic, and Tom Verhoeff. A Generic Solution for Syntax-Driven Model Co-evolution. In Judith Bishop and Antonio Vallecillo, editors, *Proceedings of TOOLS 2011: 49th International Conference on Objects, Components, Models and Patterns*, volume 6705 of *Lecture Notes in Computer Science*, pages 36–51. Springer, June 2011. DOI [10.1007/978-3-642-21952-8_5](https://doi.org/10.1007/978-3-642-21952-8_5).

- [142] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):214–234, 2007. DOI [10.1016/j.scico.2007.05.004](https://doi.org/10.1016/j.scico.2007.05.004).
- [143] Sander Vermolen and Eelco Visser. Heterogeneous Coupled Evolution of Software Languages. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Proceedings of MoDELS 2008: 11th International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 630–644. Springer, 2008. ISBN 978-3-540-87874-2.
- [144] Sander Vermolen, Guido Wachsmuth, and Eelco Visser. Reconstructing Complex Metamodel Evolution. In Anthony M. Sloane and Uwe Alßmann, editors, *Proceedings of SLE 2011: 4rd International Conference on Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 201–221. Springer, July 2011. ISBN 978-3-642-28829-6. DOI [10.1007/978-3-642-28830-2](https://doi.org/10.1007/978-3-642-28830-2).
- [145] Eelco Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Proceedings of GTTSE 2007: Generative and Transformational Techniques in Software Engineering II, International Summer School*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373. Springer, 2007. ISBN 978-3-540-88642-6. DOI [10.1007/978-3-540-88643-3_7](https://doi.org/10.1007/978-3-540-88643-3_7).
- [146] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In Erik Ernst, editor, *Proceedings of ECOOP 2007: 21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer, July 2007. DOI [10.1007/978-3-540-73589-2_28](https://doi.org/10.1007/978-3-540-73589-2_28).
- [147] Dennis Wagelaar, Ludovico Iovino, Davide Di Ruscio, and Alfonso Pierantonio. Translational Semantics of a Co-evolution Specific Language with the EMF Transformation Virtual Machine. In Zhenjiang Hu and Juan de Lara, editors, *Proceedings of ICMT 2012: 5rd International Conference on Theory and Practice of Model Transformation*, volume 7307 of *Lecture Notes in Computer Science*, pages 192–207. Springer, May 2012. ISBN 978-3-642-30475-0. DOI [10.1007/978-3-642-30476-7](https://doi.org/10.1007/978-3-642-30476-7).
- [148] WebML. *Project Web Site*, 2014. <http://www.webml.org/>.
- [149] Manuel Wimmer, Angelika Kusel, Johannes Schönböck, Werner Retschitzegger, Wieland Schwinger, and Gerti Kappel. On Using Inplace Transformations for Model Co-evolution. In *Proceedings of*

MtATL 2010: 2nd International Workshop on Model Transformation with ATL. INRIA & Ecole des Mines de Nantes, 2010.

- [150] Uwe Wolter and Zinovy Diskin. The Next Hundred Diagrammatic Specification Techniques – An Introduction to Generalized Sketches. Technical Report 358, Department of Informatics, University of Bergen, Norway, July 2007.
- [151] Uwe Wolter and Zinovy Diskin. From Indexed to Fibred Semantics – The Generalized Sketch File. Technical Report 361, Department of Informatics, University of Bergen, Norway, October 2007.
- [152] Uwe Wolter and Florian Mantz. The Diagram Predicate Framework in View of Adhesive Categories. Technical Report 405, Department of Informatics, University of Bergen, Norway, February 2013.
- [153] Jia Yu and Rajkumar Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, 3(3–4):171–200, 2005. DOI [10.1007/s10723-005-9010-8](https://doi.org/10.1007/s10723-005-9010-8).