

A FRAMEWORK FOR MULTI-MODEL CONSISTENCY MANAGEMENT

**Doctoral Dissertation by
Patrick Stünkel**

Thesis submitted for
the degree of Philosophiae Doctor (PhD)
in
Computer Science:
Software Engineering, Sensor Networks and Engineering Computing



Department of Computer Science,
Electrical Engineering and Mathematical Sciences
Faculty of Engineering and Science
Western Norway University of Applied Sciences

September 16, 2021

©Patrick Stünkel, 2022

The material in this report is covered by copyright law.

Series of dissertation submitted to
the Faculty of Engineering and Science,
Western Norway University of Applied Sciences.

ISSN: 2535-8146

ISBN: 978-82-93677-99-4

Author: Patrick Stünkel

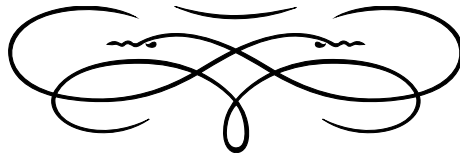
Title: A Framework for Multi-Model Consistency Management

Printed production:

Molvik Grafisk / Western Norway University of Applied Sciences

Bergen, Norway, 2022

to my parents



in memoriam *Michael Löwe*[†] (1956 - 2019)

PREFACE

Dear Reader,

whatever might have brought you here, I can assure you that I will appreciate every second that you may spend on these pages! There is a significant but unquantified amount of work hours that went into writing this text and still there are parts that could have deserved more refined formulations, more examples, more time. It is certainly a challenge to put all the insights and knowledge attained over this four year long period of my life into one comprehensive text (= linear sequence of characters). Yet, I hope that this result marks a satisfying attempt and that it, in spite of the common impression, will find its way into the hands of more people than the small circle of supervisors, close colleagues, and opponents. I also hope that you may find the diagrams in this thesis appealing, the presented concepts useful, and/or the discussions thought-provoking.

Voss, 16.09.2021

SCIENTIFIC ENVIRONMENT

The author of this thesis has been employed as a Ph.D. research fellow at the *Department of Computer Science, Electrical Engineering (Institutt for datateknologi, elektroteknologi og realfag (IDER))* at the *Western Norway University of Applied Sciences (Høgskulen på Vestlandet (HVL))* and is enrolled in the Ph.D. program *Computer Science: Software Engineering, Sensor Networks and Engineering Computing* at HVL.

The research presented in this thesis has been conducted in the Software Engineering research group at IDER under supervision of Prof. Yngve Lamo (HVL), Prof. Harald König (HVL/FHDW Hannover), and Prof. Adrian Rutle (HVL).

ACKNOWLEDGMENTS

There are 3 + 1 names that have to be mentioned first. Without them, this thesis would not exist. *Yngve Lamo* was the initiator behind my PhD position at HVL and has been a constant source of ideas, inspiration, and general advice. *Harald König* was the one that encouraged me to apply for a PhD position at HVL. From this moment on, he helped me in various ways during all stages of this PhD project. He had a significant impact on the quality of this thesis and the formal aspects of my work. Furthermore, he has been a focal point during my visits to Hannover, providing hospitality and insightful discussions. *Adrian Rutle*, with whom I spent innumerable hours drafting new ideas, polishing joint papers, hunting for dangling words, and refining formulations. He also invited me to his home. We went skiing together and always remained the focal point when I looked for general advice about work and life in Norway. These three persons read the many early drafts of this thesis, and I am greatly indebted for their support.

In addition to these central people, I must mention *Michael Löwe*, who left us way too early in 2019. His sharp mind, seemingly endless energy, and unique way of thinking things to their last consequences was what got me hooked and finally lead me to pursue an academic career in Computer Science. His intellect and advice, even though at times being very “direct”, is sadly missed. Thus, I want to dedicate this thesis to his memory.

Of course, there are many more people whose influence brought this project to a good end. *Uwe Wolter* was the one that helped me to consolidate my somewhat fragmented knowledge about category theory originating from a brief exposure during my masters. *Zinovy Diskin* was the one who opened the last doors to the higher spheres of category theory by pointing me in the right direction. He must also be credited for inviting me to Canada to conduct my research stay at McMaster University in Hamilton, Ontario. I definitely learned a lot during our intensive period of work together, which, however, had to be ended prematurely admits to a global pandemic.

At HVL, I want to thank all my colleagues for making working there always a pleasant experience: *Kristin, Pål, Lars Michael*, and *Håvard* for being the superiors that an employee could wish for by keeping formalities and politics out of the way so I could focus on my research; *Violet* and *Volker* for inviting me over for board games several times and, in 2018, giving me the opportunity to visit Brasil; and all those fellow PhD students I spent the office, lunches, and chats in front of the coffee machine with: *Alex & Angela, Justus, Rui, Michele, Tim, Suresh, Anton, Faustin, Saleh, Rizwan*. Most notably, there are *Fernando* with whom I spent a lot of time skiing and pub crawling during my first months in Bergen, and, of course, my board neighbour *Frikk* with whom I had countless insightful philosophical discussions.

Outside the office building, probably the most important person in Bergen is *Bjørnar Skaar Haveland*. He was the first real Norwegian that I talked who was not my supervisor. He invited me to move into his “Kollektivet”. Moving in into Nøstegaten was probably one of my best decisions ever and helped me so much in my first months in Bergen, far from home. Thus, I want to thank all those who lived or stayed at Nøstegaten while I was there: *Henrik, Anders & Astrid, Camilla, Runa & Osamah, Janine*. And also *Elsa* (for

keeping on building a canoe polo community in Bergen) and the other two German expatriates, *Seb & Carsten*, for good times in the mountains together.

Another big acknowledgment goes out to the “paddle family” (in no particular order: *Victor, Ruben, Rafal, Julian & Tess, Martin & Nicole, Espen, Marit, Eirik, Helge, Halfdan, Mie, Mia, Sasha, Rob, Kamilla & Johann, Rasmus & Vilde, Mariann, Jasper, Claire, Nico & Marelene, Mark Basso, Marcio, John, Lucie & Lee, Silje & Lars Georg, Martin & Marita, Johann (ISL), Herrmund, Derek & Tora*) for numerous “best days ever” on, in and by the river. Most of these names are or had been located in Voss, which has become my new home after returning to Norway after my research stay. In this small town, I have mention two persons especially: *Dag Sandvik* was the first person that took me paddling in Norway and has since become a good friend always providing tasty food, fantastic coffee, and good (philosophical) conversations. *Preben Ukvitne* is probably the person I paddled the most with. He let me stay at his place on very short notice after leaving Canada head over heels.

Last but not least, there are all my friends and my family back in Germany. Living in a different country, keeping contact and friendships over long distances is not always easy. But knowing that it was possible with my friends back in Germany is undoubtedly uplifting. I would like to thank my parents, uncle *Thomas*, and my badass grandfather *Manfred* for always supporting me. *Anne & Charly* for always inviting me over when I am back in Germany and visiting me in Norway. *Simon Hirt* for all the good times on the river and visiting me several times in Norway with *Nadja, Pascal* and *Anna*. *Daniel Steinmeyer* for agreeing to read the final draft of this thesis on very short notice and giving very extremely constructive remarks. My oldest friends *Philipp, Dustin, Basti, Tom*, and *Johannes* for the possibility to always come back home and continue exactly where we left off. And, of course, the Linden “hood” *Fabi, Lisa, Steffen*, and *Jessi* for inviting me again and again for breakfast and board games.

ABSTRACT

Software systems have become crucial for society and the economy to function. Constantly they are permeating more and more application domains. Also, they are getting increasingly integrated with already existing systems. As a result, the development and maintenance of such systems are getting increasingly complex as well. Abstraction is a central key to tackle this complexity. Thus, the software engineering research discipline conceives software systems through the means of models, i.e. simplified representations of reality that simultaneously describe and prescribe the structure and the behaviour of these systems. When engineering system's integration, the overall design involves multiple models at the same time because each model focuses on a specific aspect of the overall system. This setting is commonly called a *multi-model*. A major open challenge in software engineering is to maintain the *consistency* of a multi-model: The individual members of a multi-model are often overlapping and are authored by distinct stakeholders, which may result in inconsistencies, which negatively affect the quality of the system under development. Hence, there is a necessity for a recurring activity, which I am going to call *multi-model consistency management* throughout this thesis. This general activity includes several challenging software engineering sub-problems such as aligning heterogeneous models, automatic consistency verification and model repair, therefore touching on a range of related research fields. In this thesis, I address limitations with existing approaches and develop a solution to the issue of multi-model consistency management. The fundamental novelty of my solution is its ability to handle multi-models of arbitrary arity because established approaches are mainly limited to binary situations.

Software engineering is a constructive scientific discipline and, therefore, this thesis follows a constructive approach. In particular, three artefacts are constructed: (i) A conceptual model of the problem domain; (ii) A formalism called *comprehensive systems* (iii) A prototype tool called CORR LANG. Each artefact marks a contribution to the existing body of scientific knowledge providing value for both researchers and practitioners in software engineering. The conceptual model provides *methodological value* for software engineers, software architects and decision-makers dealing with multi-model consistency management problems by providing guidelines for analysing concrete scenarios. The formalism provides *theoretical value* and allows to abstractly represent and reason about inter-related software models in a technology-independent way, which can be used to develop solutions on a high level of abstraction. Simultaneously, it represents a generalisation of existing formalisms that have been used to describe the synchronisation of multi-models. Finally, the prototype provides *practical value*, which on the one hand, demonstrates the implementability of the conceptual and the formal framework, and on the other hand, it is directly applicable for end-users to solve concrete multi-model consistency management problems.

SAMANDRAG

Programvaresystema har vorte naudsynt for samfunnet og økonomien i sitt virke. Dei gjennomsyrrar stadig fleire område. Samstundes aukar graden av integrasjonen mellom systema. Som et resultat har utviklinga og vedlikehald av slike systema vorte stadig meir komplekst. Abstraksjon er sjølvve nøkkelen for å takla denne kompleksiteten. Difor forstår forskingsdomenet *programvareutvikling* (Software Engineering) programvaresystema gjennom modellar, dvs. forenkla bilete av røynda som på ei og same tid skildrar og spesifiserer strukturen og framferda til slike system. Når ein jobbar med ei mengd av samankopla system må ein bruke fleire modellar fordi kvar modell omfattar berre eitt einskild aspekt ved heile det underliggende systemet. Situasjonen omtalast gjerne som ein *multimodell* (multi-model). Ein situasjon som representerer eit uløyst problem i vitskapen bak programvareutviklinga. Det kan nemleg vera utfordrande å halda alle modellar konsistente fordi det er indre samanknytningar mellom dei enkelte delane av ein multi-modell som oppstår gjennom overlappingar og relasjonar mellom dei enkelte elementa. Når fleire aktørar modifiserer dei einskildte modellane slik, kan det oppstå inkonsekvensar som har negativ innverknad på kvaliteten til det overordna systemet som er under utvikling. Det er såleis naudsynt med systematisk gjentakande aktivitetar eg vil samanfatta under omgrepet *multimodell konsistenshandtering* (multi-model consistency management). Dette emne som då skal undersøkjast i denne avhandlinga inneberer ulike utfordrande problem. Til dømes, tilpassingar av heterogene modellar, automatisk konsistensverifisering og modellreparasjon. I denne oppgåva tek eg opp svakheitlar ved eksisterande tilnærmingar og presenterer ei løysning på problemet med multimodell konsistenshandtering. Den mest grunnleggande nyvinninga i løysninga mi er evna til å handtera multimodellar av vilkårlig aritet, fordi etablerte tilnærmingar hovudsakleg er avgrensa til binære situasjonar.

Programvareutvikling er ein konstruktiv vitskapeleg disiplin, og difor følgjer denne avhandlinga ein konstruktiv tilnærming. Spesielt verte tre artefaktar konstruert (i) Ein konseptuell modell av problemdomenet; (ii) Ein formalisme kalla *omfattande system* (comprehensive systems) (iii) Eit prototypeverktøy kalla CORR LANG. Kvar av desse artefaktane er eit bidrag til den eksisterande mengda av vitskapeleg kunnskap, med verdi for både forskarar og praktikarar innan programvareutvikling. Den konseptuelle modellen har ein *metodisk verdi* for programvareingeniørar, programvarearkitektar og avgjerdstakarar som jobbar med problema knytta til konsistensstyring av fleire modeller. Formalismen har *teoretisk verdi* og gjev ein høve til å representera innbyrdes relaterte programvaremodellar på ein abstrakt, teknologiavhengig måte, noko som i neste rekke opnar for å resonnerer over dei. Samstundes representerer denne formalismen ein generalisering av eksisterande formalismar som har blitt brukt til å skildre synkronisering av multimodellar tidlegare. Til slutt gjev prototypen *praktisk verdi*. På den eine sida demonstrerer den at det konseptuelle og formelle rammeverket er gjennomførleg, og på den andre sida representera den eit løysning som direkte kan takast i bruk for multimodell konsistenshandtering.

LIST OF ABBREVIATIONS

BNF backus naur form.

BPMN Business Process Model and Notation.

BX bidirectional transformations.

CR consistency rule.

CS computer science.

CSP constraint satisfaction problem.

Decision Model and Notation Decision Model and Notation.

DPO double pushout.

DSL domain specific language.

EHR electronic health Record.

EMF Eclipse Modeling Framework.

ER entity-relationship.

FHIR Fast Healthcare Interoperability Resources.

FOL first order logic.

GG Graph Grammar.

GP general practitioner.

GT graph transformation.

HL7 Health Level 7.

HLR high level replacement.

HTTP HyperText Transfer Protocol.

ICD International Classification of Diseases.

ICT information and communications technology.

IDE integrated development environment.

JSON JavaScript Object Notation.

LOINC Logical Observation Identifiers Names and Codes.

MDA Model-Driven Architecture.

MDSE model-driven software engineering.

MOF Meta Object Facility.

OCL Object Constraint Language.

OO object orientation.

OWL Web Ontology Language.

PL programming language.

RDF Resource Description Framework.

REST REpresentational State Transfer.

RPC Remote Procedure Call.

SAT satisfiability.

SE software engineering.

SMT satisfiability modulo theories.

SNOMED-CT Systematized Nomenclature Of Medicine - Clinical Terms.

SOA service oriented architecture.

SOAP Simple Object Access Protocol.

SQL Structured Query Language.

TGG triple graph grammar.

UML Unified Modelling Language.

URL unique resource locator.

WS web service.

XMI XML Metadata Interchange.

XML eXtensible Markup Language.

XSD XML Schema Definition.

Contents

Preface	i
Scientific Environment	iii
Acknowledgments	v
Abstract	vii
Samandrag	ix
List of abbreviations	xi
I INTRO	1
1 Introduction	3
1.1 Challenges: Interoperability, Consistency, and Traceability	3
1.2 Solutions: Software Engineering and Modelling	6
1.3 Motivational Scenarios	10
1.3.1 Semantic Interoperability between Software Systems	10
1.3.2 Consistency of Software Design Documents	17
1.4 Research Project: Multi-model Consistency Management	23
2 Method	27
2.1 Philosophy of Science	27
2.1.1 Philosophy: A (short) historical account	27
2.1.2 Science: The Demarcation Problem	29
2.1.3 Research Methodology	30
2.2 Research Methodology in Constructive Sciences	31
2.3 Research Methodology in Software Engineering	33
2.4 Research Methodology in this PhD project	34
II CONTRIBUTIONS	35
3 Conceptualisation	37
3.1 Existing Concepts & Ideas	37
3.1.1 View-based Software Development	37
3.1.2 (In)consistency Management	38

3.1.3	Traceability Management	40
3.1.4	Consistency Management in UML	41
3.1.5	Multi-View Modeling	41
3.1.6	Metamodeling	42
3.1.7	Model Transformation	43
3.1.8	Model Management	44
3.1.9	Megamodeling	45
3.1.10	Model Repair	46
3.1.11	Bidirectional Transformations	47
3.1.12	Coupled Evolution	49
3.2	Generic Model Management Framework	50
3.2.1	Artefacts	50
3.2.2	Operations	51
3.3	Multi-Model Consistency Management Framework	54
3.3.1	Model Spaces	54
3.3.2	Commonalities	55
3.3.3	Consistency Rules	56
3.3.4	Model Repair	57
3.3.5	Architectures	57
3.3.6	Summary: Conceptual Model	59
4	State of the Art	61
4.1	Method	61
4.2	Feature Model	63
4.2.1	Models	64
4.2.2	Change	65
4.2.3	Conformance	66
4.2.4	Correspondence	68
4.2.5	Matching	72
4.2.6	Consistency	72
4.2.7	Verification	73
4.2.8	Repair	74
4.3	Observations	79
4.4	Demonstration of Selected Approaches	80
4.4.1	Echo	80
4.4.2	Epsilon	83
4.4.3	eMoflon	87
4.5	Summary & Identified Limitations	89
5	Formalisation	91
5.1	Representation	92
5.1.1	Formalising Models	92
5.1.2	Formalising Conformance	98
5.1.3	Formalising Change	104
5.1.4	Formalising Correspondence	109
5.2	Verification	114

5.2.1	Verification via Merging: Colimit	117
5.2.2	Verification via Weaving: Comprehensive System	121
5.3	Restoration	126
5.3.1	Back- and Forth-propagation	126
5.3.2	Adhesivity	128
5.3.3	Triple Graph Grammars and Graph Diagrams	133
5.4	Summary & Future Directions	136
6	Implementation	141
6.1	First Iteration: GraphQL Federation	142
6.1.1	Background: Web Services & GraphQL	142
6.1.2	Problem Statement: Federation	145
6.1.3	Existing Tool: Apollo Federation	147
6.1.4	Solution Design: Declarative Schema Merging	149
6.1.5	Solution Implementation: GraphQLIntegrator	152
6.2	Second Iteration: Model Management Functionality	154
6.2.1	Tech Spaces: Integrating EMF	155
6.2.2	Comprehensive Systems: Generalising the Federation	157
6.3	Third Iteration: Consistency Management Functionality	159
6.3.1	Integration of existing verification tools	159
6.3.2	Common Constraints: INTEGRITY & FORALL	160
6.3.3	Model Management via Goals	162
6.4	Summary & Future Directions	164
7	Validation	167
7.1	Validation of the Conceptual Framework	167
7.2	Validation of the Formalism	169
7.3	Validation of the Tool	170
7.3.1	Feasibility	170
7.3.2	Features	178
7.3.3	Scalability	180
III	OUTRO	189
8	Conclusion	191
8.1	Summary	191
8.2	Threats to Validity	192
8.3	Related Work	193
8.3.1	Industrial Solutions	193
8.3.2	Academic Approaches	194
8.4	Future Work	195
8.5	Conclusion	196
	Bibliography	236
A	Literature Study Refinements	237

B Proofs	243
B.1 Proof of Theorem 8	243
B.2 Proof of Theorem 9	245
B.3 Proof of Theorem 5	246
B.4 Proof of Proposition 13	247
B.5 Proof of Theorem 14	247
B.6 Proof of Theorem 15	249
B.7 Proof of Proposition 17	250
B.8 Proof of Theorem 18	251
C Category Theory Essentials	253
C.1 History and Background	253
C.2 Introduction	254
C.3 Important Concepts	262

Part I

INTRO

“No problem is so big or so complicated that it can’t be run away from!”

—Linus van Pelt in *The Peanuts* (Charles M. Schulz)

CHAPTER 1

INTRODUCTION

It is hard to think of any aspect of our everyday life that is not already permeated or at least affected by information and communications technology (ICT). For instance, purchasing an item in an online retail store, paying for it by credit card, booking an appointment at a doctor, reading a PhD thesis in a PDF reader etc. This phenomenon has been recognised by computer scientists [201], social scientists [83], economists [423], policy makers and journalists [16] alike. These days, it appears in the public discussion under the name “*digitization*”, which describes the ever-increasing importance of ICT, especially in sectors, which traditionally were not associated with computing.

Arguably, the most important part of ICT is *Software*. Software development and maintenance are complex processes, which involve a multitude of different stakeholders and constantly changing requirements. Thus, there are many examples of software projects that were delayed, went over budget, or simply failed [63, 137, 315, 358].

Considering the socio-economic relevance, *efficient* (i.e. sustainable resource consumption) delivery of *correct* (i.e. meeting the requirements), *safe* (i.e. absence of unintended behaviour), and *secure* (i.e. protection against intrusion and abuse) software systems marks a major challenge in the 21st century.

1.1 Challenges: Interoperability, Consistency, and Traceability

Software is the designation for the entirety of all non-material computer programs. A program is a collection of machine-readable instructions that tell a computer how to act in order to perform a specific task. In this thesis, I will often speak of software *systems* (Greek: “*systema*” = composition) to highlight the fact that a software product generally consists of several smaller parts called *components* or *modules*. Software systems have different shapes, complexity and purposes. According to Ivari [248], the most common purposes of software systems are *to automate* (i.e. to reduce the share of manual activities), *to augment* (i.e. to provide assistance during a manual activity), *to mediate* (i.e. to act as means of communication), or *to inform*.

Nowadays, software systems have become more and more *integrated* [38], i.e. formerly independent software systems have to interact with each other. Examples are given by novel applications such as *Industry 4.0*, which describes the integration of industrial assembly lines with internet technology, and *Internet of Things*, which describes the integration of home appliances with internet technology. Another example is the growing number of applications and platforms that allow citizens to interact

Introduction

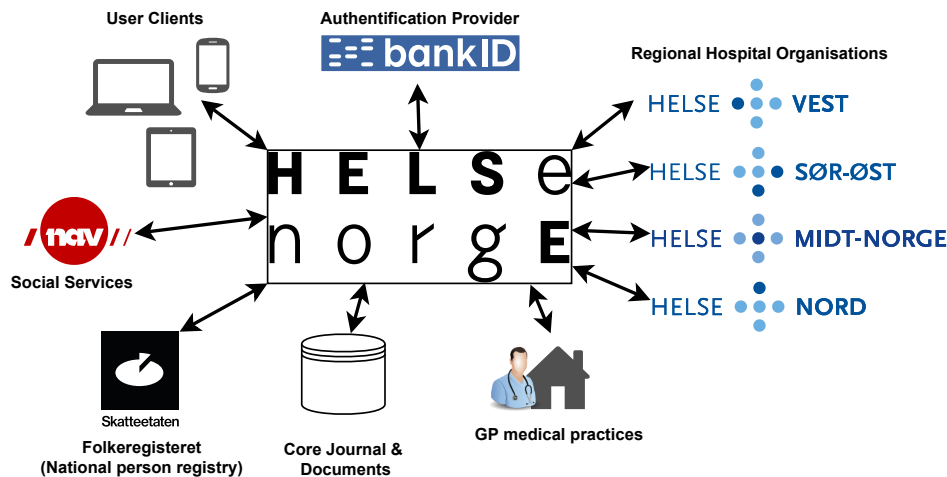


Fig. 1.1: Helsenorge

with public authorities, one concrete representative being the web portal *HelseNorge*¹ in Norway. HelseNorge is an interface for Norwegian residents to electronically interact with the national healthcare system and to inspect and review their personal medical information. As a web portal, HelseNorge communicates with several other software systems. A simplified overview is sketched in Fig. 1.1. *HelseNorge* is accessible through various types of client devices. User authentication and authorisation are governed by a digital authorisation provider (“bankID”). An authorised user can inspect their hospital journals, which are maintained by the different regional hospital organisations in Norway. Further, she/he can communicate and book appointments with her/his GP. Moreover, the portal provides a digital mail box for the correspondence between citizens and health care providers, allows to record core medical information (e.g. allergies or vaccinations) in a central place, and is integrated with the national social service provider (“NAV”) and the national person registry (“Folkeregisteret”). All parties in Fig. 1.1 operate one or multiple software systems, which most likely were built at various points in times, using different technologies, and created for diverse purposes. This organisational, functional, and technical heterogeneity is a potential source of *interoperability* issues [33].

Definition 1.1 Interoperability [95]

Interoperability is the ability of two or more systems or components to exchange information and to use the information that has been exchanged.

The *Healthcare Information and Management Systems Society, Inc. (HIMMS)*, an industrial consortium fostering usage and improved quality of ICT in health care, distinguishes four levels of interoperability [223]: *Foundational* interoperability describes the basic ability of two or more systems to communicate in the first place. *Structural* interoperability describes their ability to process the exchanged information. *Semantic* interoperability describes their ability to interpret this information in “the same way”. Finally, *organisational* interoperability refers to high-level aspects such as business

¹www.helsenorge.no

1.1 Challenges: Interoperability, Consistency, and Traceability

processes, regulations, laws and social factors. Semantic interoperability marks one of the main motivations for the work presented in this thesis. This issue is a long-standing research topic [382, 473] and requires the discovery and alignment of *shared concepts* in the domains of the respective software system [33]. The latter is conceptually similar to *consistency* management in software design and development and forms the second motivation for this thesis.

Definition 1.2 Consistency [95]

Consistency describes the degree of uniformity, standardisation, and freedom from contradiction among the documents or parts of a system or component.

As mentioned in the beginning, software development involves multiple stakeholders, various technologies, and requirements that are constantly changing. Hence, software systems are seldom built by a single person. Instead, there are multiple teams, each focusing on a different component or aspect of the system under development. These teams produce various artefacts when designing the system, e.g. a list of functional requirements, a plan of the system architecture, a diagram showing the domain entities and their relationships and so on. These artefacts describe different parts of the *same* system. Consequentially, there are *overlaps* between these artefacts. Problems occur when overlapping system specifications differ. Such *inconsistencies* can lead to misunderstandings, incompatible components, unwanted behaviour or even system failure. Allegedly, the company *Airbus* lost approx. 6.1 billion dollars due to inconsistencies in design documents [496], and an investigation [437] on the MARS Climate Orbiter accident revealed that its crash was due to an inconsistency (mismatch between metric and imperial units). Inconsistencies arise when some of the design artefacts change but not all occurrences of overlaps are changed accordingly. Thus there is a need for *traceability* among design artefacts.

Definition 1.3 Traceability [95]

1. The degree to which a relationship can be established between two or more products of the development process [...].
2. The degree to which each element in a software development product establishes its reason for existing; [...].

The term traceability originated in the Requirements Engineering (RE) research discipline, i.e. the ability to follow the life of a requirement throughout the various stages of the software development process and their different design artefacts [459]. It was later generalised to describe any kind of relationship between artefacts in the software development process [7]. The issue of traceability has been investigated by a large number of studies, see [352, 434, 459]. Yet, evidence for rigorous use of traceability solutions in industrial settings is rare. Moreover, existing solutions are reported as ad-hoc, vendor-specific, restricted to a certain domain or having limited interoperability [138, 352, 462].

1.2 Solutions: Software Engineering and Modelling

The scientific discipline that investigates the issues related to software and its development is called software engineering (SE). Its history is intertwined with the history of computing itself [56]. Konrad Zuse's Z3 (1941) or Mauchly and Eckert's ENIAC (1943-1945) are considered to be the first "real" computers. Prior to these inventions, computers were merely an abstract idea, solely playing a role for some mathematicians and logicians [465]. During its infancy, computer programming was performed close to if not directly *on* the machine: Instructions were literally "hard-wired" such that there was no strict distinction between hardware and software. Hence, programming used to be an expert activity. It was performed by a small team of scientists with extensive knowledge of the inner workings of the machine. A major breakthrough fuelling the expansion of programming was the invention of *compilers* and the first high-level programming languages FORTRAN [27] and COBOL [414], which allowed to write instructions on a higher level of abstraction [484]. During the 1960's, programming grew from being an expert activity into a profession and the term "Software Engineering" was born. One of the first "big" commercial SE projects was the development of the mainframe operating system OS/360 at IBM. It uncovered many of the inherent difficulties of writing complex software systems with a big team of engineers. Many of these insights are reported in Brook's influential book [63]. Software projects at that time often exceeded time and budget, which coined the term "Software Crisis". This Software Crisis was the incentive for a NATO conference held in Garmisch in 1968 [356], which is associated with SE becoming its own research discipline [64]. Boehm's article [52] also played a major role in disseminating software engineering issues in the scientific community. Today, SE represents a distinct discipline within computer science (CS). Parnas [375] defines Software Engineering as the multi-person creation of multi-version software, while the IEEE standard glossary [95] defines it as follows:

Definition 1.4 Software Engineering [95]

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

SE is considered to be a *socio-technical* engineering discipline [158]. This means it is situated in the tension field between the study of people and their behaviour in relation to software on the one side and the study of (formal) foundations of software methods and tools on the other side. There has been a heated dispute [221] about the direction that should prevail in SE research. In his famous position paper [115], computer scientist Edsger W. Dijkstra promotes a rigorous use of mathematical methods as the solution to the Software Crisis. This line of thought gave rise to the *formal methods* research discipline [2, 50, 237]. Others emphasised the importance of the socio-organisational part of software engineering [431]. This research direction provided evidence about the efficient organisation of people in the software development process. Traditionally, software development adhered to a long-running sequential "waterfall" process, while nowadays, most software projects follow a more "agile" approach, i.e. delivering the functionality of a software system in small iterations [40]. Again others argue that due to its intrinsic complexity [62], there is not a single comprehensive approach but rather

a set of best practices [250, 375] and design patterns [191] that Software Engineers should apply. Taking an *Hegelian* stance, the “conclusion”, most likely, lies in the pluralism of all the knowledge, methods and tools [60] which have been aggregated throughout the more than 60-year-long history of software engineering.

model-driven software engineering (MDSE) is a major sub-discipline of software engineering, which I am going to focus on in this thesis. MDSE emphasizes the importance of *models* as the primary entity in the software development process. The term *model*, in the most general sense, refers to any artefact related to the creation of a software system. The design artefacts mentioned in Section 1.1 are archetypical example for such models.

Remark 1.1 *Model* – Etymology and Interpretation

The term *model* is arguably one of the most ambiguous terms in science. Its origin (Lat: “modulus” = measure, standard) alludes to its most broad conception: A model is a simplified representation of reality. It is used in this interpretation by the empirical sciences (e.g. *Bohr’s* atom model, *Newton’s* classical mechanics) and social sciences (e.g. *Schulz von Thun’s* four-sides communication model) to formulate scientific theories. Logicians use the term *model* to describe a formal construct that satisfies a given proposition. Statisticians use *model* as another name for assumptions over a dataset (population). The Artificial Intelligence (AI) sub-discipline *Machine Learning* uses the term *model* to describe a set of trained parameters for a machine learning algorithm. In Software Engineering, *model* is used in a rather undisciplined manner and may refer to various types of design documents. A detailed historical account of the term *model* in science is given in [417].

A speciality of the model-driven approach to software engineering is the *double nature* of software models. They are simultaneously *descriptive* and *prescriptive*. A model *describes* a certain component or aspect of the software systems under development, i.e. acting as a simplified representation of the problem domain for which the system is developed. Likewise, a model *prescribes* the structure or behaviour of the respective component or system aspect, i.e. acting as a specification. Sufficiently “formal” models can replace source code (the machine instructions written in a programming language) given the necessary *code generation* or *model execution* facilities [61].

To illustrate this idea, consider Fig. 1.2. The figure depicts a simplified representation of (an iteration of) the software development process. The process (iteration) begins in the *problem domain* with an informal description of the system’s requirements, often given in natural language. The problem domain and the requirements are *analysed* and translated into an abstract solution, which is technology independent. This development phase is called *design* and its products are *models*. Models guide the engineers during *programming*, i.e. the translation of the abstract technology-independent solution into a technology-specify solution (i.e. program code). The resulting program is generally augmented with code from existing software libraries and integrated with generic system components such as databases or web servers before it is automatically *compiled* into binary artefacts. These artefacts are the final result of the software development process and can be *deployed* on a given platform (computer hardware, virtual machine,

Introduction

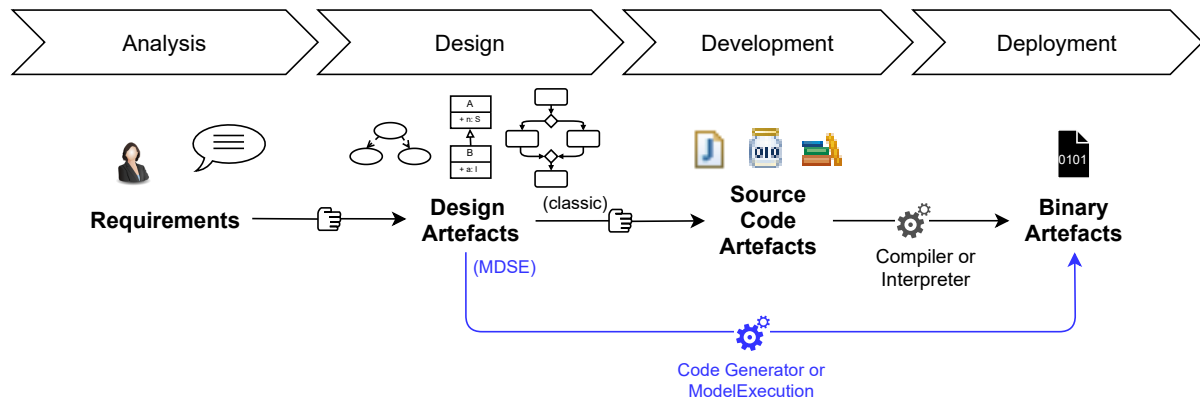


Fig. 1.2: Software Development Process: Classical and Model-Driven

hosted “cloud” infrastructures, etc.). In a “classical” software development process, the transitions between the different stages, except for the final compilation, are based on *manual* activities. In the *model-driven* variant, the models created during the design phase are directly passed on to a code generation or model execution engine, see the second branch in the lower half of Fig. 1.2. Thus, the “automation border” is shifted from the solution domain towards the problem domain.

The fundamental goal behind MDSE is to raise the *abstraction* level from the solution domain (technology) to the problem domain (people). The most obvious benefit of model-driven software engineering, conveyed by the presentation in Fig. 1.2, is *automation* [102, 476]: The generation of concrete artefacts (source code) from an abstract description (model) renders many repetitive and error-prone activities, including coding, redundant. This increases efficiency and possibly contributes to higher product quality. In recent years, the *communicative* aspect of MDSE received increased attention. This aspect is associated with *domain specific languages (DSLs)* [180, 321, 474]. This concept, first reported in [43], trades the expressiveness of a generic programming language against a less expressive but also less complex textual or graphical language. This language embodies the concepts of the problem domain, which alleviates the communication between domain experts and the software engineers because it abstracts away from technical details of the solution domain. A third benefit is that formal models allow the application of means for *verification*, *simulation* and *exploration* early on in the development process. Thus, conceptual inconsistencies in the abstract specification [249, 251] can be detected early and future system behaviour can be simulated [41].

MDSE has reportedly been successfully applied in industry [243, 486, 487], however, it is not universally adopted by the software industry yet [351]. Still, MDSE-like ideas can be observed throughout the history of software engineering. The introduction of high-level programming languages and compilers could be considered as a first attempt to raise the abstraction level in programming. With data types and control structures, programmers could express their solutions on a higher level of abstraction and did not have to reason about memory addresses and instruction counters any more. The first programming languages FORTRAN and COBOL can be considered as domain specific in the sense that they targeted a certain domain (scientific computing and accounting, respectively).

In general, behavioural and structural models have been used as software specifica-

tions since the very beginning. *Flow charts*, originally introduced by the psychologists Frank and Lillian Gilbreth and later adopted by the computer scientists John von Neumann and Hermann Goldstine [210], allowed to develop an algorithm independently of a concrete technology. The flowchart variant by Nassi and Shneiderman [355] as well as *state charts* [219] are other examples of behavioural models.

The introduction of relational databases facilitated the design of the *data model* independent of the program code. This sparked interest in structural modeling. Chen's [85] *entity-relationship (ER)* model is a method to describe a data model on the conceptual level in terms of entities and relations between them. The conceptual ER model is translated into a logical model (tables) understood by the database management system. This translation follows a standardised process. Hence, it is possible to automate it. This motivated research within *Computer Aided Software Engineering (CASE)* – inspired by Computer Aided Design (CAD) in other engineering disciplines. The final goal was to completely automate the software development process. CASE had a period of popularity during the 90s but it could never met the exaggerated expectations. Nowadays, some CASE tools still exists but are considered niche application.

Around the peak period of CASE's popularity, the *object orientation (OO)* paradigm, which is still prevalent today, spread among software developers. This novel approach to software design and programming required new notational systems for describing data structures and algorithms as flow charts and ER diagrams were not perfectly suited any more. An abundance of visual languages were created and eventually consolidated in 1997 when the *Object Management Group (OMG)*² adopted the graphical notation developed by Booch, Jacobsen and Rumbaugh [405] as the first version of the *Unified Modelling Language (UML)* [365], which remains the de-facto standard modeling language in software engineering. Inspired by the CASE approach, the "vision" of a *Model-Driven Architecture (MDA)* was formulated [388], which proposed to replace coding by UML modeling, i.e. software systems are automatically generated from a UML model that is refined through several stages. Just as CASE, MDA did not prevail but some of the concepts and technology that was developed in this context such as *Meta Object Facility (MOF)* [368] is still in use.

Kent [273] proposed a more broad and inclusive vision called *Model-Driven Engineering (MDE)*. He emphasizes that MDE is not strictly tied to OMG-standards and highlights the underlying ideas (e.g. utilising abstraction). This term developed into research domain, labelled by various similar acronyms³, which comprises scientific investigations concerning Unified Modelling Language (UML), DSLs, automated software development, Language Engineering, Metamodeling, as well as *formal modelling* (e.g. Z, B, VDM and Alloy [249]), and also approaches related to industrial engineering toolkits such as *Matlab/Simulink* [261]. In this thesis, I will stick to the term MDSE as I am working within software engineering and want to utilise the *prescriptive* power of models. I interpret MDSE more in a methodological (i.e. working with abstraction in a rigorous way) and less in a technological (i.e. focusing in the OMG-ecosystem) way. In

²an organisation originally founded to standardise an object-oriented communication protocol called CORBA

³The literature contains a whole family of related acronyms (e.g. MDSD, MBSE, ...) which arises from the cartesian product of {Model-Oriented (MO), Model-Based (MB), Model-Driven (MD)} and {Software Development (SD), Software Engineering (SE), Engineering (E)}.

Introduction

the SE dichotomy “people vs. technology”, MDSE leans more towards the technology side. However, people-oriented SE methodologists themselves propose an idea called *Domain Driven Design (DDD)* [160] which relies on the same basic ideas (i.e. starting with abstract domain models instead of coding). Commercial actors are currently “selling” MDSE under the name “lowcode” platforms, e.g. *OutSystems*⁴, *Mendix*⁵ or *Amazon Honeycode*⁶.

Concerning the challenges mentioned in Section 1.1, MDSE represents both a “means” as well as an “end” to these challenges: On the one hand, well-described *domain models* play a central role for achieving semantic interoperability. On the other hand, model traceability and consistency management are long-standing open research problems in MDSE, which hampers its further adoption by the industry [351], see the “tool-related” challenges in the MDSE state-of-the-art report in [72]

1.3 Motivational Scenarios

After having introduced both problem domain (i.e. software, interoperability, consistency, traceability) and solution domain (i.e. software engineering and modeling (MDSE)), it is now time to have a look at two concrete problem scenarios that act as the motivators for this PhD project.

1.3.1 Semantic Interoperability between Software Systems

Landscapes of software systems in organisations are highly heterogeneous. Simultaneously, *integration* of software systems is a common and important issue. Integration means to turn separate things into a “whole”. In this case, several independent software systems are aligned to create a *distributed system*, i.e. a *system of systems* [51]. Let me explicate this issue on a concrete example from the health-care domain, namely *hospitalising* a patient due to a broken leg. As in any other organisation, ICT plays an important role in supporting this process.

Fig. 1.3 depicts an architectural overview of the software systems involved. Upon arrival at the emergency room, an *electronic health Record (EHR)* is created, which will be continuously updated throughout the whole hospital stay. The software system storing these records is called a *journal system*. In the beginning, the patient’s record is augmented with externally available information (medical history, allergies, etc.). This information can be retrieved from the patient’s general practitioner (GP) by accessing their journal system. Planning the patient’s surgery involves a software system whose task it is to schedule the required resources (personnel, surgical suites, beds, ...). Furthermore, during the hospital stay, X-ray images and blood samples are taken and evaluated, which is performed by specialised software systems. Eventually, when the patient is discharged, a third-party payment collector is invoked, who invoices the patient for the received treatment. Thus, the overall picture includes (at least) six software systems, which communicate within and across three different organisations. Interoperability between these systems is indispensable: The medical history has to

⁴<https://www.outsystems.com>

⁵<https://www.mendix.com>

⁶<https://www.honeycode.aws/>

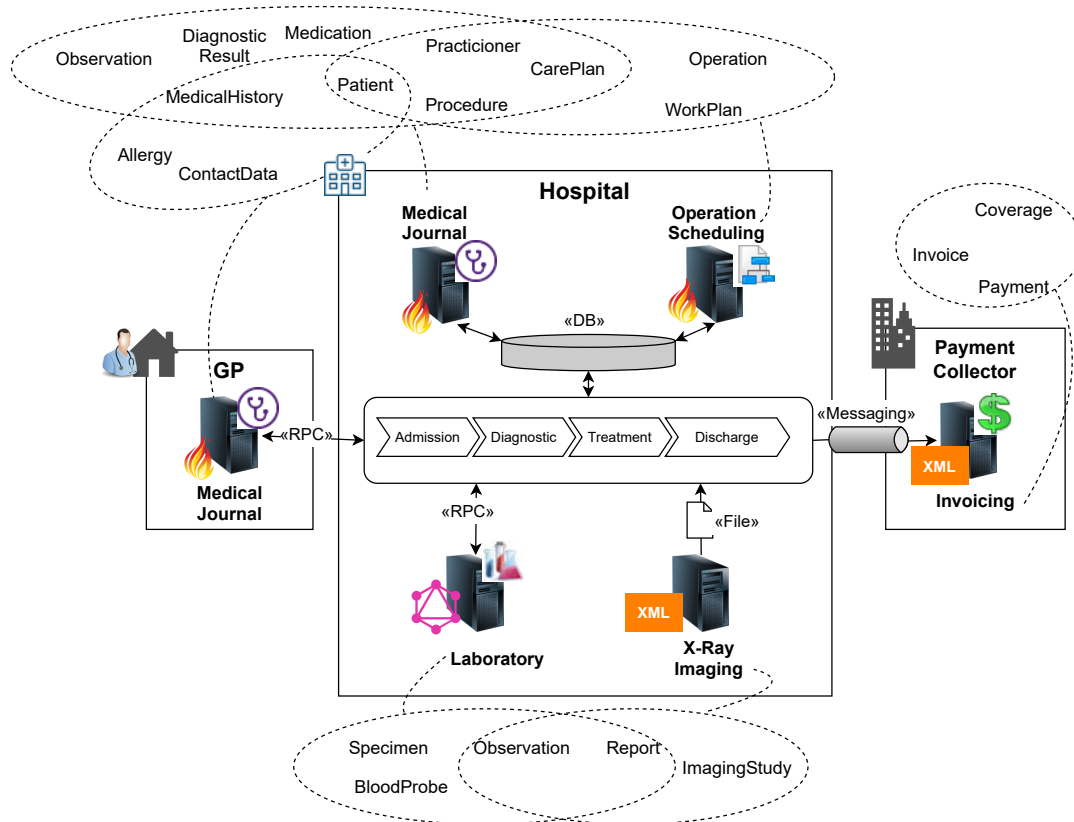


Fig. 1.3: Systems around a patient treatment process

be retrieved from the GP’s journal system, the blood test results must be sent to the journal system, and the payment collector requires the patient’s address and some general information about the received treatment to write the invoice. However, and this is especially true in the health-care domain [467, 481], interoperability between software systems is not always given. Missing interoperability means that data is transferred *manually*, which unnecessarily binds human resources and is another source of potential errors [159, 468].

Sheth [427]	HIMSS [223]	EIF [94]	Problem Domain
System	Foundational		Hardware, Operating Systems, Networks
Syntactic		Technical	Encodings, Formats, Protocols
Structural	Structural		Data representation and structures, Schemas
Semantic	Semantic	Semantic	Terminology, Meaning
-	Organisational	Organisational	Business processes, Services
-	-	Legal	Laws, Regulations

Table 1.1: Levels of interoperability

Sec. 1.1 discussed four levels of interoperability, which are proposed by the HIMSS society [223]. Both, more fine-grained and more coarse-grained categorisations have been presented in the literature. A selection is shown in Tab. 1.1. In [427], Sheth focuses on technical issues and further distinguishes between *system* and *syntax* related concerns on the foundational level, while the *European Integration Framework (EIF)* [94] considers all levels below semantic interoperability as “technical” but adds a

Introduction

new level of *legal* interoperability. Some researchers [22, 387] proposed a *pragmatic* interoperability level that lies between semantic and organisational interoperability. Each level addresses a different problem domain, similar to the OSI reference model for networking [457].

Beginning on the topmost organisational levels, a legal and organisational *framework* is required, which enables interoperability. Most of the intricacies related to this aspect are of managerial and juridical nature and thus outside the scope of this thesis. From a software-oriented point of view it is important to note that the common consensus on an interoperable system architecture is given by the so-called *service oriented architecture* (SOA) [222]: Every system is conceived as provider for a set of *services*. A service realises a concrete business functionality (e.g. storing a medical record, retrieving a blood test result), has well-defined in- and outputs, and can be called other systems or a human actor. Services are *orchestrated* to perform a business process. The SOA-concept is also embodied in ICT management and architecture frameworks, such as ITIL [247] and TOGAF [220]. A recent industry trend called *Microservice Architecture* [12, 183, 185] is based on the same underlying idea and promotes to design a complex software system as a collection of smaller independent and domain specific software systems communicating with each other.

The bottommost system interoperability level considers the technical solutions to physically connect systems with each other. With the ubiquity of the internet and internet technologies such as IP, TCP, and UDP, connecting software systems is not a major difficulty any more, but there are network safety (timeouts, connection failures, message loss) and security (privacy, authorisation, encryption) considerations which have to be addressed. These issues are investigated within their respective research domains [454] and are also outside the scope of this thesis.

Given physical inter-system connections, syntactical interoperability considers the different protocols and formats that are used to facilitate information exchange over the network. According to [240], there are four approaches for achieving information exchange between software systems, which are also mentioned in Fig. 1.3:

File Transfer Systems communicate via writing and reading files to and from a shared location.

Shared Database Systems communicate via storing their data in the same database.

Remote Procedure Call (RPC) Systems communicate via calling each other's procedures over the network.

Messaging Systems communicate via writing and reading messages to and from a central *message broker*.

There is a variety of tools and implementations behind each approach: For instance, file transfer mechanisms are included in every major operating systems and programming language; open-source and/or commercial Database Management Systems are encountered in every enterprise system landscape; there are programming-language-specific RPC mechanisms but arguably the most popular RPC implementation strategy today, web services (WSs), utilises the HyperText Transfer Protocol (HTTP). Messaging sys-

tems have predominantly been associated with commercial products⁷ in the past, but recently there have been more popular open-source solutions⁸. All four approaches have different advantages and disadvantages, see [240, 282], but share the common requirement for data *serialisation* and *deserialisation*. Internally, a software system organises its data in terms of memory addresses and pointers (data structures). When data is written to disk or sent through a network interface, it has to be translated (serialised) into a linear sequence of bytes. To make sure that this sequence of bytes can be read and translated back (deserialised) to a memory-based data structure at the receiving system, a *shared definition* of the serialisation format is required. The textual formats eXtensible Markup Language (XML) and JavaScript Object Notation (JSON) are highly prevalent and are being used as file formats, for storing objects in databases, for encoding WS-requests and -responses, and also as a message format of message brokers. There are also binary formats⁹, which offer a more compact encoding but require an additional infrastructure for reading and writing, e.g. code generators.

Generic serialisation formats are equipped with a *schema*. A schema is a formal description of the possible types of data elements and their internal structure. All schema-related issues are considered on the level of structural interoperability. The concept of a schema was originally introduced in databases. Relational database management systems require the existence of a schema before data can be written to the database. XML and JSON offer optional schema facilities and so do many non-relational databases. The advantage of optional schemas is the greater flexibility. The downside is the greater risk of communication errors since there will always be an (at least implicit) schema that must be adhered to by the serialisation and deserialisation procedure. If the schema is not explicitly formulated it is still hard-wired into the program code. An explicit schema causes overhead but has the advantage that it allows to inspect and reason about the format, enables automation (e.g. generating (de-)serialisation procedure from a given schema), and allows to verify the well-formedness of the serialisation result.

A schema – explicit or implicit – partially captures a semantic aspect as it (partly) represents the specific domain of the software system [33], which leads us to the level of semantic interoperability. From a pragmatic point of view, the goal on this level is to make sure that all systems interpret the exchanged information in “the same way” [387]. Because of distributed ownership and different purposes, the conceptual data models – also called *domain models* [160] – differ. However, there are *shared concepts* In Fig. 1.3, the conceptual data models are indicated by

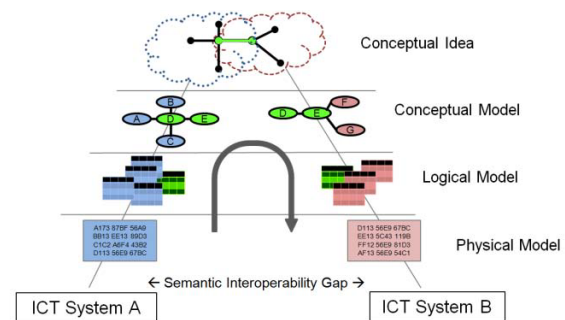


Fig. 1.4: Semantic Interoperability Gap [33]

⁷E.g. TIBCO (<https://www.tibco.com>) or IBM WebSphere (<https://www.ibm.com/products/mq>).

⁸E.g. as RabbitMQ (<https://www.rabbitmq.com>) or Apache Kafka (<https://kafka.apache.org>).

⁹E.g. Protocol Buffers (<https://developers.google.com/protocol-buffers>), Apache Thrift (<https://thrift.apache.org/>), or Apache Avro (<http://avro.apache.org/>)

Introduction

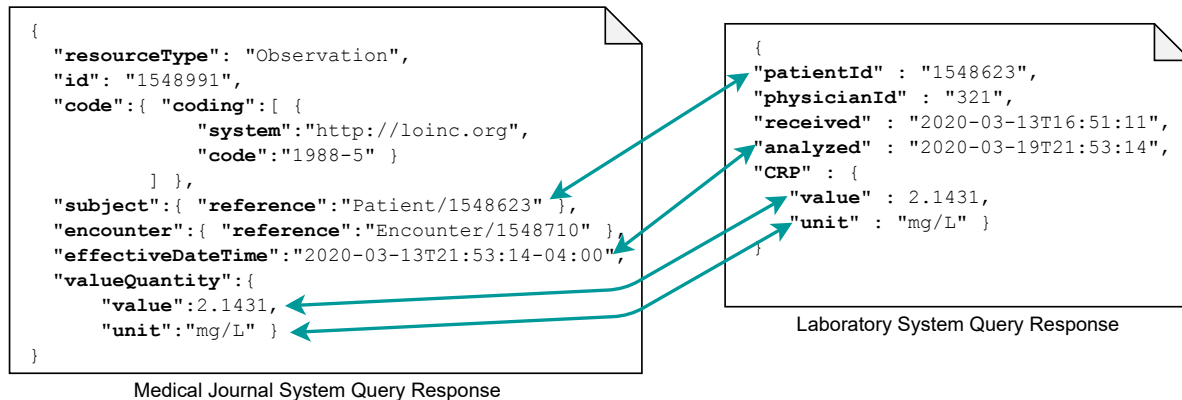


Fig. 1.5: Exchanged Blood Tests: Data view

dashed ovals¹⁰, which contain terms that the respective systems operate with. These ovals partly overlap, which hints to shared or exchanged information. The latter manifests in the so-called *semantic interoperability gap* [33], which is shown in Fig. 1.4: Different conceptual models lead to different logical models (= schemas), which again lead to different physical encodings.

Taking an example from the hospital scenario in Fig. 1.3: The results of the blood tests that are processed by the laboratory systems should be transferred to the patient record in the journal system. It is more than likely that the physical representations of blood test results differ between both systems, see Fig. 1.5. This figure shows two concrete representations of the same information, both encoded as JSON documents (i.e. syntactical interoperability is achieved). But the structure of these documents obviously differs. Carefully looking at the content reveals identical information, which is highlighted by the dashed lines in Fig. 1.5. To transfer this data between the systems, the document must be transformed while preserving the semantic equalities a.k.a. the *data mapping problem* [136].

According to the survey in [33], there are two approaches to address this issue:

Standardisation i.e. forcing *homogeneity*. Either on the level of conceptual models or on the level of logical models.

Mediation i.e. performing intermediate translations or transformations.

There are many examples of the first approach: In the industry domain, *RosettaNet* [106] is a well-known standardised message schema. In the health care domain, Health Level 7 (HL7)¹¹ groups together a set of standards electronically representing messages, clinical documents or clinical knowledge, the most recent one being Fast Healthcare Interoperability Resources (FHIR). The difficulty with the standardisation and its top-down nature is that with a growing number of systems and growing number of use cases and domains, the common standard tends to become rather complex or partly *generic* to accommodate for the plethora of requirements coming from different stakeholders [33]. Moreover, it is not always feasible to enforce a standardised data model on already existing systems. For instance, the development effort for this

¹⁰These ovals can be seen interpreted as “bounded contexts” in the sense of DDD [160]

¹¹7 stands for OSI-layer seven, i.e. the application layer.

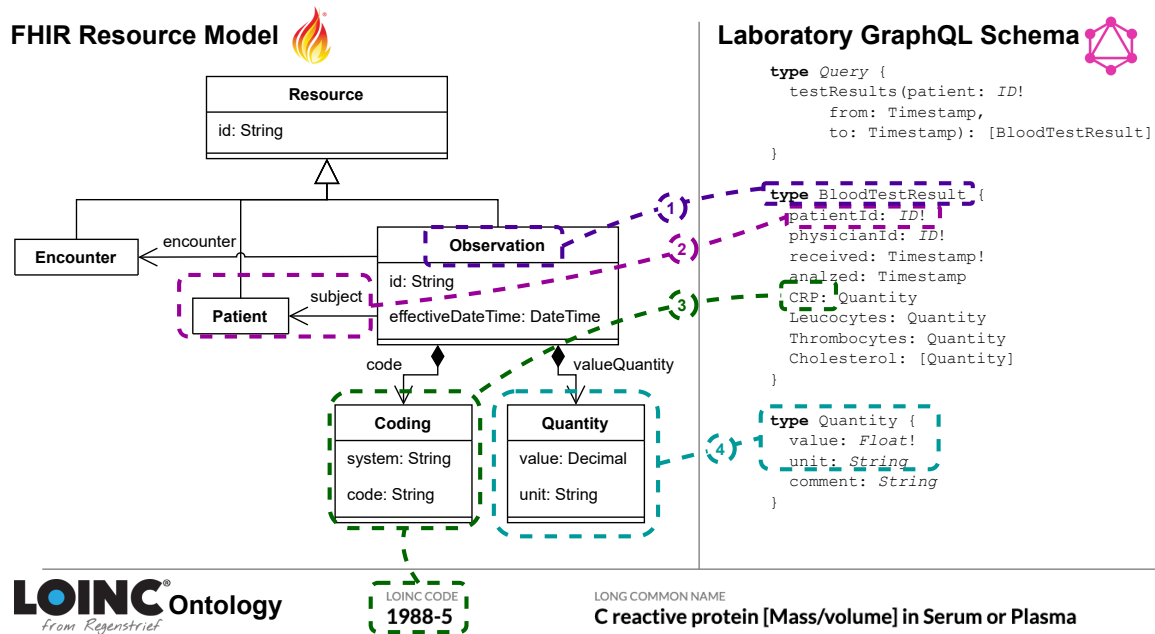


Fig. 1.6: Exchanged Blood Tests: Schema view

adaptation may be unreasonably high. It may even be impossible to achieve if the respective software system is bought from an external vendor.

Mediation, as a bottom-up approach, avoids these problems. In this approach, heterogeneity is accepted and for different schemas intermediate translations are defined. To avoid ad-hoc solutions, researchers propose to use *semantic web* technology [184, 317, 382]. Semantic web technology is based on *ontologies* [444]. The latter is a formal and therefore electronically processable description of objects of the real world¹². To apply semantic web technology, schemas have to be annotated with references to concepts defined in an ontology such that schema translation and process orchestration can be (partly) automatised [337] with the help of specialised tools.

Applying the idea of mediation to the example from Fig. 1.5, one has to analyse related elements on the schema level, see Fig. 1.6. Assume for this scenario that the data model of the journal system is based on the standardised FHIR resource model while the laboratory system is based on a home-grown solution, where the schema is defined using the GraphQL, a modern WS technology. FHIR defines a collection of health-related resource types, e.g. Observations, Patients. An (excerpt of) this data model is shown in Fig. 1.6, depicted using UML class diagram syntax. FHIR is generic in the sense that an Observation can represent various types of medical observations. Thus, there is an element called coding, which allows referencing an existing ontology. In the health care domain, several such ontologies exist. For example, the WHO *International Classification of Diseases (ICD)*¹³, the *Systematized Nomenclature Of Medicine - Clinical*

¹²The name originally stems from philosophy, i.e. the study existence, being and reality, see also Chap. 2.

¹³<https://icd.who.int/browse11/l-m/en>

Introduction

Terms (SNOMED-CT)¹⁴, or Logical Observation Identifiers Names and Codes (LOINC)¹⁵. “LOINC is a common language for clinical and laboratory observations” and thus an appropriate ontology for annotating observations. Concretely, for the example in Fig. 1.5, there is a LOINC element representing the result of a “C-reactive protein (CRP)”¹⁶ blood test with the code 1988-5. The laboratory system has a custom-made schema, which is presented in textual syntax. The schema consists of types, which represent complex elements (think of “classes” in UML) that have fields, which either have scalar type (think of “attributes” in UML) or complex type (think of “references” in UML). Fig. 1.6 shows how the CRP TestResult object can be identified with a Observation with LOINC-code 1988-5, which visualised by dotted lines. Thus, the mappings from Fig. 1.5 appear again in Fig. 1.6 – this time on the type level.

Ontologies and semantic web technologies are a powerful approach to address schema mediation, in particular, and semantic interoperability in general [33]. However, from an SE perspective there are still several unsolved problems [107]: First, an ontology seldom covers all elements of a domain model. Secondly, ontologies such as SNOMED-CT or LOINC contain a lot of ambiguities, which eventually necessitates human supervision and expert domain knowledge to align schema elements and ontology terms correctly. Thirdly, implementations of the semantic web idea are currently tied to a specific technology stack comprising technologies such as Resource Description Framework (RDF) or Web Ontology Language (OWL). Meanwhile, software practitioners have adopted different and more lightweight technologies for developing WSs [49]. The authors of [107] therefore propose a pragmatic model-driven approach that bridges the gap between the “academic” semantic web approach (focus on formal foundations and logic) and “practical” WS implementations (focus on messaging patterns).

MDSE and ontologies are conceptually similar but are investigated by two disparate scientific communities. Their relationship has been further analysed in [26]: The biggest conceptual difference is that MDSE focuses more on *prescription* while *ontologies* focus on description. Moreover, they employ disparate technologies and tools (UML/MOF vs. RDF/OWL). Yet, both domains are founded on the idea to work with abstract representation of reality, i.e. models.

Thus, after the above discussion, the following practical issues still remain:

- Detection of shared concepts among the schemas of different software systems,
- representation of such shared concepts,
- definition of data type mappings based on shared concepts,
- maintaining the consistency of data elements instantiating shared concepts.

The final bullet point of the preceding list describes a general issue with distributed storage of information. In every system landscape as displayed in Fig. 1.3 there will be multiple “replicas” of the same information. For example, there will be multiple Patient records (one for the hospital, one for GP, one for the payment collector, ...). A

¹⁴<https://www.snomed.org/>

¹⁵<https://loinc.org/>

¹⁶A common blood test for detecting inflammations.

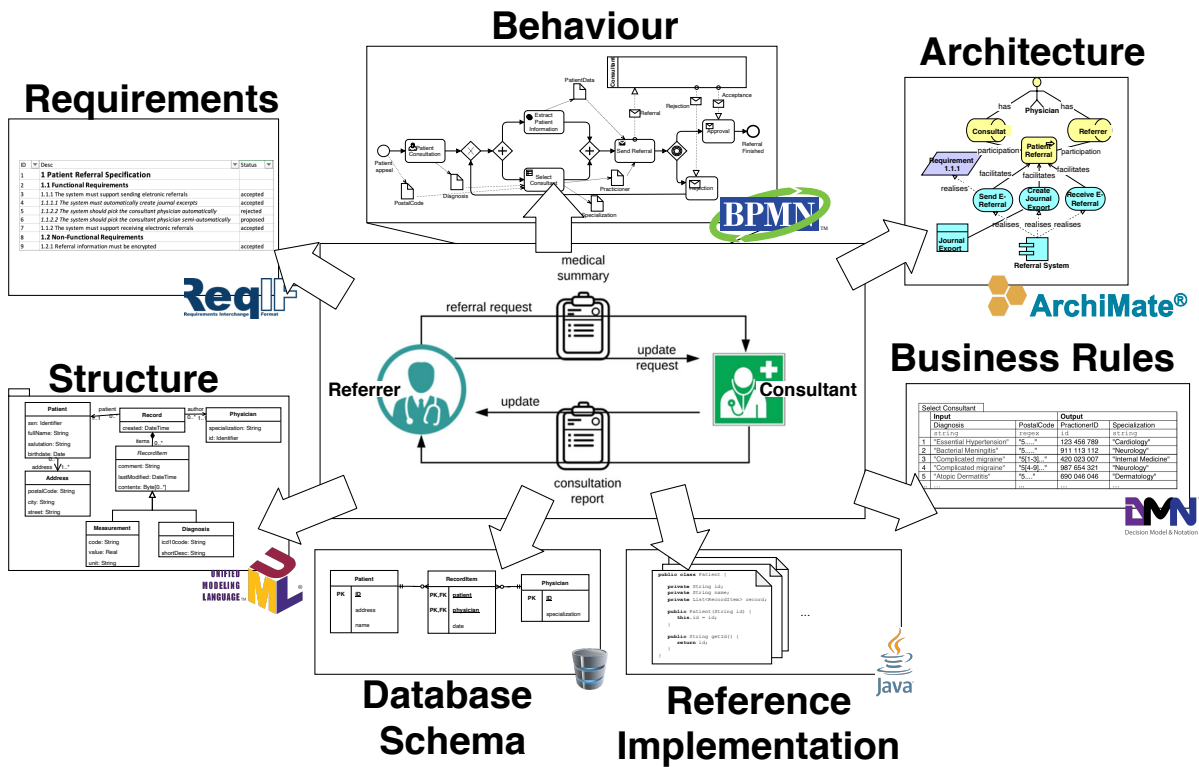


Fig. 1.7: Designing a “Patient Referral”: Big Picture

record generally comprises several *features* or *fields*. In case of the Patient, there will be a birthdate, an address, the contact information of a relative etc. If one of this fields is changed, e.g. the respective person moves to another address, that information must be changed for all replicas, otherwise it will be inconsistent. Hence, I conclude with formulating the following *consistency rule* (CR), which will be revisited several times throughout this thesis and thematically leads to the second motivational scenario.

CR1 The values of every field of each replica of a shared data element (Persons, Observations, etc.) must be equal.

1.3.2 Consistency of Software Design Documents

Design and development of complex ICT systems and processes involve multiple stakeholders (clients, domain experts, business analysts, enterprise architects, software developers, system administrators) and *documents* (requirements, specifications, drawings, source code). To give a more concrete example, let us consider a software project aiming to develop an ICT solution facilitating *electronic patient referral*. A referral is “the act of sending a patient to another physician for ongoing management of a specific problem with the expectation that the patient will continue seeing the original physician for co-ordination of total care” [424]. It is an important and recurring process in the healthcare domain and ICT support for it is desirable [481]. There are several aspects that have to be addressed during the design and development process. An overview of the abstract referral process and its various aspects in Fig. 1.7.

Introduction

Each aspect has specific purpose and comprises one or more design documents, where each aspect is associated with a different *language*. The functional and non-functional *requirements* are denoted in the *Requirements Interchange Format (ReqIF)*¹⁷, an XML-based file format for storing and exchanging requirements. The content of a ReqIF document is basically a hierarchically-ordered list of all requirements written in natural language, which are augmented with optional meta-data (e.g. whether a requirement is mandatory or what the acceptance status of the requirement is). The *architecture* is described using the *ArchiMate*¹⁸ modeling language, which is often used in combination with the TOGAF architecture framework and allows enterprise and system architects to describe the system's architecture and context on an abstract level. The *behaviour* is described in terms of process models, which are denoted utilising *Business Process Model and Notation (BPMN)* [364], a popular process modeling language. The *structural* aspects (components and data types) are described using UML, the de-facto standard modeling language in software engineering. The *business rule* aspect embodies the domain specific expert knowledge. In this example, this knowledge is represented in the form of *decision tables*, which are encoded employing the standardised *Decision Model and Notation (Decision Model and Notation)* [369]. The *database schema* aspect describes the data format for eventually storing the process data persistently. It is described in terms of a relational schema technically realised via statements written in *Structured Query Language (SQL)*. Finally, the *reference implementation* contains a concrete programming language realisation of the patient referral application, which is given as *Java* code. Since every languages is assigned to exactly one aspect, I may sometimes use the aspect name and language interchangeably.

Fig. 1.7 reflects the principle of *Separation of Concerns*: Each stakeholder focuses on their particular domain and uses the tools and languages that are best suited for their individual use case. Forcing everyone to work on a single centralised specification document is considered unfeasible, even impossible [68, 87, 187, 462]. Hence, distributed system specifications are inevitable. This, however, bears the dangers of *fragmentation*, *redundancy* and eventually *inconsistencies* [280]. For example, imagine that the UML designer deletes a class, which the process designer just created a reference to. There are manifold possibilities of inconsistencies such as the *value consistency (CR1)* in Sec. 1.3.1 or more intricate *behaviour* or *interaction* inconsistencies [455, 462], which require an in-depth analysis to be detected. In general, *inconsistency* arises from violated *consistency rules*, which can be generic (like CR1), language-specific (e.g. the inheritance relationship among UML classes must be acyclic), domain specific (e.g. all entities representing clinical information must be annotated with LOINC codes), or project-specific (e.g. the implementation of the persistence layer must follow the *Repository* pattern [160]). The scope of a consistency rule may span one or more documents/aspects and there are varying degrees of importance, i.e. some rule violations may jeopardize the correct workings of the system immediately while other rules are merely project conventions.

Let us have a look at some concrete consistency rules within the patient referral scenario. During the first stage of the development process – the *analysis* – the more general aspects of the system specification play an important role, i.e. the requirements,

¹⁷<https://www.omg.org/spec/ReqIF/About-ReqIF/>

¹⁸<https://www.opengroup.org/archimate-forum/archimate-overview>

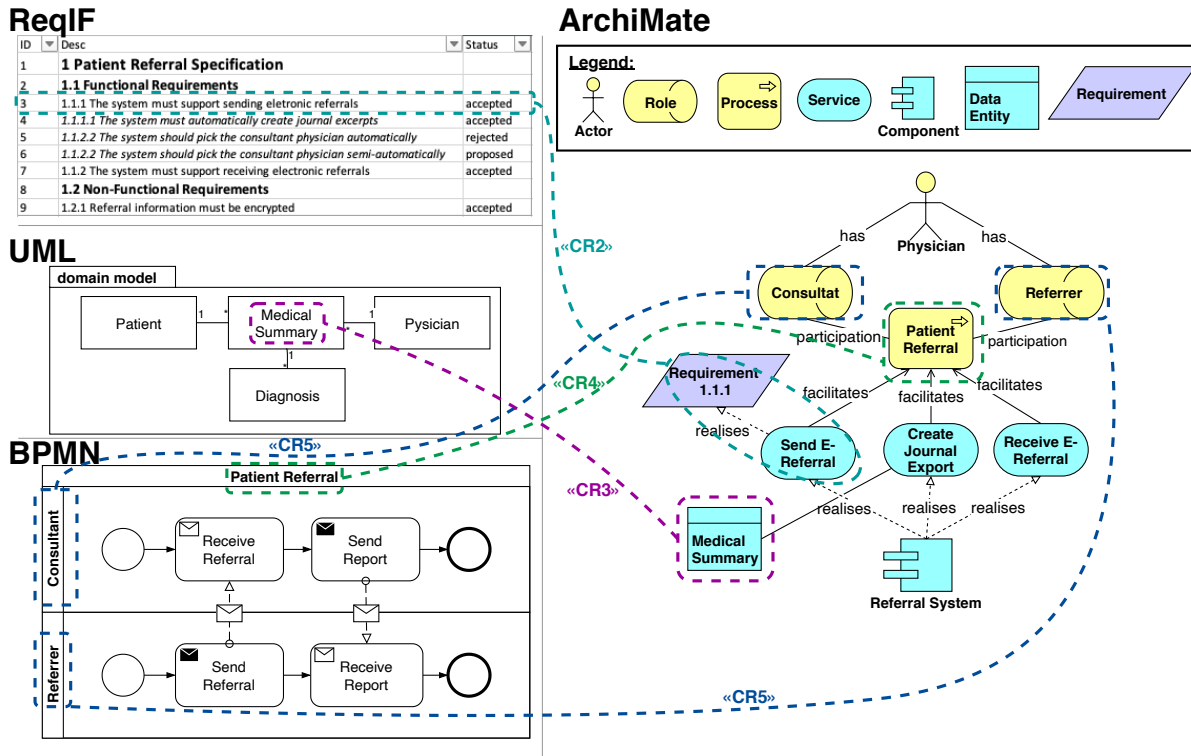


Fig. 1.8: E-Referral Analysis Stage: Requirements (ReqIF) – Architecture (ArchiMate) – Process (BPMN) – Data (UML)

the system architecture, the domain model and the abstract process model. Fig. 1.8 depicts the contents of the relevant design documents. The requirements are listed in a tabular form and the business architecture is depicted utilising the graphical *ArchiMate* syntax. A legend explicates the interpretation of the graphical elements. Domain and process model are presented in UML and BPMN syntax, respectively, and I assume the reader to be familiar with these visual languages. Moreover, Fig. 1.8 contains coloured dotted lines to highlight the fact that those elements are subject to specific consistency rules. Let us assume the following list of consistency rules in this scenario:

- CR2 Every *accepted functional requirement* must be implemented by an *application service*.
- CR3 Each *process element* within the *architecture model* must be refined by a *UML class*.
- CR4 Each *process element* within the *architecture model* must be refined by a *BPMN diagram*.
- CR5 Each *role* that participates in a process must appear represented as a *pool* in the respective BPMN diagram.

The consistency rules express the fact that there are “overlaps” among the ReqIF, ArchiMate, UML and BPMN documents: Some concepts (processes, roles, requirements, data entities) appear in multiple documents simultaneously at different stages of refinement. CR2–CR5 enforce that these concepts are represented accordingly. Overlaps are a special case of *traceability relations* [434], compare Def. 1.3 in Sec. 1.1. Hence, there

Introduction

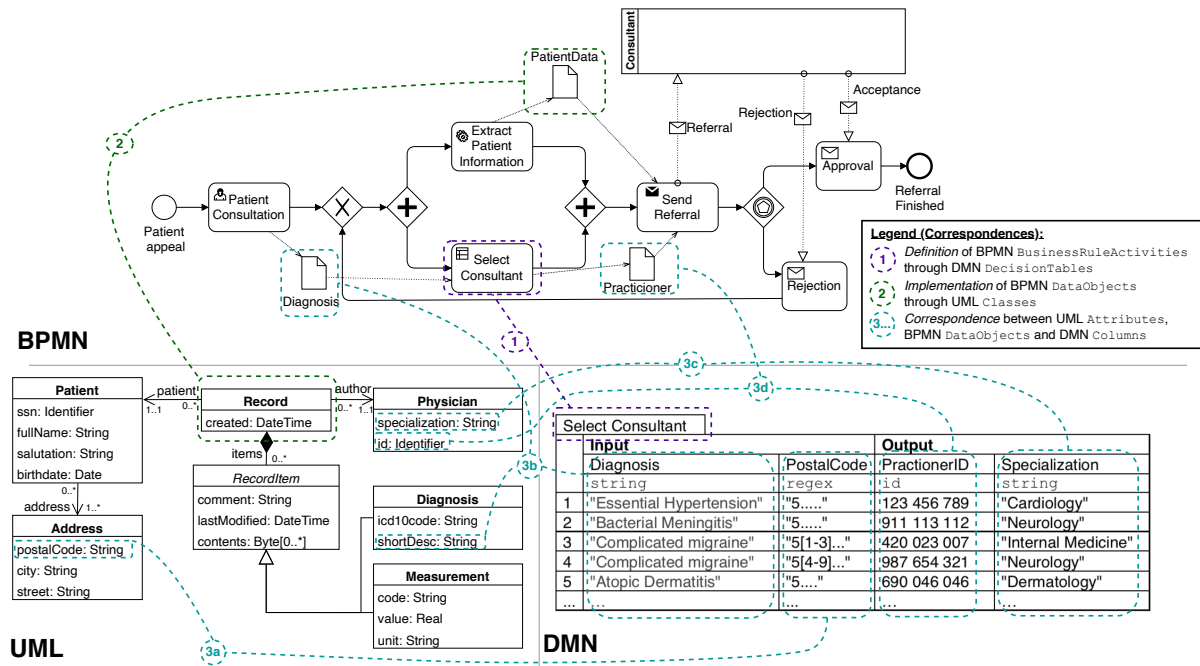


Fig. 1.9: E-Referral Design Stage: Process (BPMN) – Data (UML) – Decision Tables (DMN)

can be many more cases of inter-document relationships subject to more intricate consistency rules.

Let us assume the development process of the eReferral system has proceeded further into the *design* stage. At this point, the UML and BPMN documents have been refined to contain a more detailed specification. Additionally, business rules have been formulated in DMN augmenting the system specification. The result is shown in Fig. 1.9. Compared to Fig. 1.8, the UML data model has been extended with additional entities, attributes, and data types. The process model has also been refined and now shows a simplified version of the referral process in [481] from the viewpoint of the referring physician: The process is triggered by a patient’s appeal beginning with an introductory consultation. Afterwards, information about the patient and its medical history is extracted while in parallel a consultant is selected via a business rule. The patient information is then sent to the consultant. The consultant can either approve the referral or reject it. In the latter case, another consultant has to be found. If a consultant accepts the referral, the process is finished. Moreover, there is a decision table, which embodies the business rules and specifies the behaviour of the business rule activity “*Select Consultant*”.

There are traceability relationships among these three design documents, which are visualised through coloured dashed lines in Fig. 1.9. First, there is a relationship between the *business rule activity* in the BPMN diagram and the *decision table* in the DMN model. The process model focuses solely on behaviour and has no notion of data. Yet, the running process will produce and consume various data elements. To make the required data explicit during the design phase, BPMN provides the graphical element called *data object* (visualised by a file symbol), which refers to elements specified in the UML model. Simultaneously, the columns of the decision table have a *data type*,

which is defined in the UML model and may also be represented via a data object in the BPMN diagram. Let us assume that these traceability relationships are subject to the following consistency rules:

- CR5 Every *Activity* of type *business rule* in BPMN must be defined by a corresponding *decision table* in the DMN model.
- CR6 Every *data object* in BPMN must be implemented by a corresponding *class* or an *attribute* in the UML model.
- CR7 Every *Column* in DMN must be implemented by a corresponding *attribute* in the UML model and the *column type* must correspond to the *data type* of the *attribute*.
- CR8 *Columns* in DMN may be associated with a *data object* in BPMN. In this case, being an a column on the *input side* of the *decision table* means that there must be a corresponding *consumedBy* association between the respective *data object* and *activity* in the BPMN model. Accordingly, a column on the *output side* enforces a *producedBy* association. Moreover, this relationship must be compatible with the implementation-relationship between *data objects* and *attributes* (CR6) such that each triple of *data objects*, *attributes* and *columns* is in a ternary “to-one” relationship.

The traceability relationships in Fig. 1.9 and their consistency rules CR5-CR8 illustrate that there can be involved inter-model relationships of various arities. Furthermore, these relationships may not be as easily identifiable as simple overlaps that are identified over equality of their names. Hence, domain specific expert knowledge and/or means for comparing heterogeneous design documents are required.

The design documents are constantly being modified to account for new requirements or changed regulations. Due to the fragmented nature of the system development process, inconsistencies will arise eventually. Since inconsistencies are expected to have a negative impact of the quality of final development result, it is desirable to resolve them [462], ideally relying on some sort of tool support and automation. A special case of this form of consistency restoration is *synchronisation* [19]: Often an inconsistent update is actually an “incomplete” update, a change to an element has not been applied to all occurrences within the various models. A synchronisation mechanism ensures that a change is applied consistently among all models.

To explicate this further, let us consider the final *implementation* stage of the system development process, which involves the UML model, the database schema and the reference implementation shown in Fig. 1.10. The seasoned software expert will immediately identify the presented situation as an instance of the *object-relational mapping* problem: The UML class diagrams can be more or less directly translated into a corresponding representation in the form of Java classes. Some of these classes are marked as “entities” (note the respective stereotype in UML and the @Entity-annotation in the Java code, see [456]). This means that those classes must be represented as database tables. However, the underlying conceptual models behind UML/Java (object-oriented) and the database schema (relational) do not match exactly. Therefore, special mapping rules have to be defined, e.g. relational database schemas

Introduction

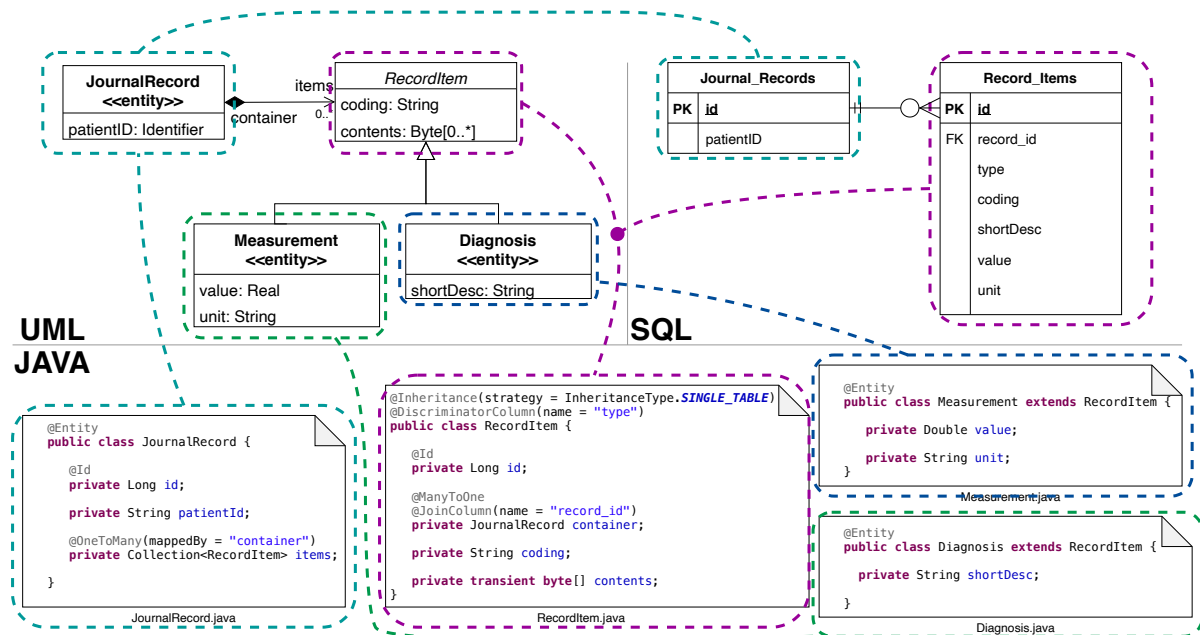


Fig. 1.10: E-Referral Implementation Stage: Data Model (UML) – Database Schema (SQL) – Reference Implementation (Java)

have no corresponding concept for object-oriented inheritance, therefore there are three distinct mapping strategies to address these situations, see [181]. More specifically, the consistency rules for the in this case are as follows:

- CR11 Every UML class must be implemented by a Java class and attributes and references must be implemented by respective fields of the class.
- CR12 Every UML class with the stereotype «entity» must be mapped to a database table where the respective attributes are realised as columns if not the respective field has been marked as `transient` in the Java implementation. Furthermore the corresponding Java class must receive the annotation `@Entity` and there must be a new field `id`.
- CR13 A reference between entity classes must be realised via a mapping table.
- CR14 Abstract classes must be realised according to object-relational mapping strategy, that is declared via an annotation [456].

There is a plethora of object-relational mapping tools available for most programming languages which allow to derive a database schema from an annotated class specification and handle almost all technical details of the database access. Nonetheless, it is often the case that all design documents are considered authoritative at the same time, i.e. the database schema cannot always be automatically replaced when changing the class model because the database administrator may have performed a manual change on the schema which must not be overwritten. This situation is known as *round-trip engineering* [19] and is still a fairly open issue in software engineering research. Consistency issues, in general, have been investigated by SE researchers since the mid-eighties [175, 433, 497] and there is a vast body of results, approaches and

1.4 Research Project: Multi-model Consistency Management

tools, see e.g. [284, 433, 462, 469]. However, solutions are often limited to a certain technology, modeling language (e.g. UML only) or tool. Interoperability between existing solutions is reported as insufficient [352, 462]. Also, support for a comprehensive management of traceability relationship supporting heterogeneous software artefacts is rather limited [87, 352, 462]. As a result, practitioners adhere to ad-hoc solutions and the following tasks generally occur in settings, which are similar to the *eReferral*-case discussed here:

- Detection and representation of overlaps and other types of traceability relationships among multiple heterogeneous design documents,
- definition and execution of domain specific consistency rules among multiple heterogeneous design documents,
- handling of detected inconsistencies, preferably via means of (semi-)automatic consistency verification or synchronisation affecting multiple artefacts simultaneously.

1.4 Research Project: Multi-model Consistency Management

At first sight, the two scenarios described in Sec. 1.3.1 and Sec. 1.3.2 appear to be quite distinct: In Sec. 1.3.1, one is working with several already existing systems, while in Sec. 1.3.2, one is developing a single novel system. Yet, wh both scenarios show structural similarities: In Sec. 1.3.1, database contents, syntactically described by different schemas, are the primary artefacts, while in Sec. 1.3.2, design documents, syntactically described by different specification languages, are the primary artefacts. In Sec. 1.3.1, there is shared information (records) among the databases subject to a condition (CR1), while in Sec. 1.3.2, there are traceability relationships subject to various rules (CR2-CR14). In MDSE it is common to consider “everything as a model” [76], i.e. an abstract representation¹⁹. Hence, the artefacts in both scenarios can uniformly be treated as models. Moreover, the shared information and the traceability relationships can abstractly considered as some sort of inter-model relationship. Note the visual similarity in the presentation of the traceability relationships in Fig. 1.9 and the mappings in Fig. 1.5 and Fig. 1.6. Additionally, the list of open practical issues in Sec. 1.3.1 resembles the one in Sec. 1.3.1.

The scenarios described in Sec. 1.3 motivate the need for what I call “*Multi-model consistency management*”:

¹⁹“One cannot know the *thing-in-itself*” (Kant)

Definition 1.5 Multi-model consistency management

Multi-model consistency management is the common term for all activities centred around

- declaring,
- monitoring, and
- maintaining

the *consistency* within a *multi-model*, where a multi-model [126, 278] is understood as

- a collection of (possibly heterogeneous) models, including
- a collection of inter-model relationships among these models.

The problem introduced before show that multi-model consistency management is an open and relevant issue in software engineering, which I want to address within this PhD project. For this, I formulate the following research questions to guide the investigations:

RESEARCH QUESTIONS

- RQ1** How to represent multi-models together with their consistency rules?
- RQ2** How to detect consistency violations w.r.t. consistency rules within multi-models?
- RQ3** How to resolve consistency violations w.r.t. consistency rules within multi-models?
- RQ4** How to organise these activities into a comprehensive workflow and solution architecture ?

These four questions will be investigated on an abstract conceptual level (Chap. 3), with regard to the state of the art (Chap. 4), on a purely formal level (Chap. 5), and on a technically implementation-oriented level (Chap. 6).

STRUCTURE OF THIS THESIS This thesis is divided into three parts and an appendix. This chapter gave an introduction into the research problem and stated the research questions. Together with the following chapter, which will discuss the philosophical and methodological foundation of this research project, it constitutes the first part of the thesis.

The chapters 3 to 7 constitute the second and main part of the thesis, which contains my scientific contributions. In Chap. 3, I develop a *conceptualisation* of the problem domain, which is based on ideas from related research domains. In Chap. 4, I present the result of a literature study about approaches related to multi-model consistency management in the form of a *feature model*, which augments the conceptualisation with an overview of the state of the art. In Chap. 5, I present a novel *formalism* called

1.4 Research Project: Multi-model Consistency Management

comprehensive system, which forms a formal foundation for multi-model consistency management. In Chap. 6, I present a prototypical implementation of the conceptual and theoretical framework in form of a tool called CORR_{LANG}²⁰, which addresses multi-model consistency management problems on a practical level. In Chap. 7, I conduct a *validation* of the aforementioned research artefacts.

Finally, Chap. 8 represents the third part of the thesis, which concludes this project presentation with a summarising discussion about the research contributions and an outlook towards future research directions.

The appendix is completely *optional* and may be viewed by the interested reader on demand. It is divided into three sections: Appendix A contains technical details about the literature study in Chap. 4, Appendix B contains complete proofs of the central propositions and theorems in Chap. 5, and Appendix C contains background material about the formalism (category theory) employed in Chap. 5 in order to make this thesis self-contained.

THESIS RELATED PUBLICATIONS This thesis is *not* organised as a paper collection but as a manuscript. Still, major parts of my scientific contributions have been presented in various publications [302, 445–448, 450, 451]. The relationship between these publications and the content of this thesis is explicated in the list below:

- [445] The article *Multimodel correspondence through inter-model constraints*, joint work with all my supervisors presented at *Seventh International Workshop on Bidirectional Transformations (Bx 2018)* addresses RQ1 and RQ2 (on a formal level). The content of this publication is featured in Chap. 5.
- [446] The article *Towards Multiple Model Synchronization with Comprehensive Systems*, joint work all with my supervisors and presented at the *23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020)* addresses all four research questions (on a formal level). The content of this publication is featured in Chap. 5.
- [302] The article *Single Pushout Rewriting in Comprehensive Systems*, joint work with Harald König, and presented at the *13th International Conference on Graph Transformation (ICGT 2021)* covers an important aspect (rule rewriting) of the formal framework and thus addresses RQ3 on a formal level. It is featured in Chap. 5.
- [447] The article *GraphQL Federation: A Model-Based Approach*, joint work with two of my supervisors and Ole van Bargaen, presented at the *16th European Conference on Modelling Foundations and Applications (ECFMA 2020)* presents the predecessor tool of CORR_{LANG} and addresses RQ1 and RQ4. It is featured in Chap. 6.
- [451] The article *Multi-Model Evolution through Model Repair*, joint work with all my supervisors and published in a special issue of the *Journal of Object Technology on Models and Evolution* addresses RQ1–RQ3 on a conceptual level and investigates the state of the art w.r.t. to these questions. It is featured in Chap. 3 and Chap. 4.

²⁰<https://github.com/webminz/corr-lang>

Introduction

- [450] The article *Single pushout rewriting in comprehensive systems of graph-like structures*, joint work with Harald König, published in a special issue of the journal “*Theoretical Computer Science*” represents an extended version of [302]. Thus it is partly featured in Chap. 5.
- [448] The article *Comprehensive Systems: A formal foundation for Multi-Model Consistency Management*, joint work with all my supervisors, published in a special issue of the journal “*Formal Aspects of Computing*” represents an extended version of [446], which adds more material concerning the conceptual aspects and addresses RQ3 even more. It is mostly featured in Chap. 5 and partly in Chap. 3.

“What we know is a drop, what we don’t know is an ocean.”

—Isaac Newton

CHAPTER 2

METHOD

2.1 Philosophy of Science

A *Doctor of Philosophy*, abbreviated Ph.D. (*philosophiae doctor*), is the highest obtainable academic degree. Its etymology (“*docere*” (Latin): to teach, and “*philosophia*” (Greek): love of wisdom) traces back to the universities of medieval Europe, which were traditionally organised into the four faculties *theology*, *medicine*, *arts* and *law*. Anything else was considered *philosophy*. Indeed, the history of modern day *science* and philosophy – more precisely the western tradition of philosophy¹ – is tightly intertwined. The early philosophical inquiries known as *natural philosophy*² can be seen as the predecessor of what is today called *natural science*. The study of objects and laws of the physical reality became *physics*, which, with growing knowledge and the inherent growing complexity, was further divided into specialised fields that focus on a specific part of reality, e.g. *chemistry* (molecules), *biology* (organisms), *sociology* (human behaviour), *economy* (human organisation).

Scrutinizing the own methodology is a trait of scientific work practice. The central question “What qualifies as *science*?”, i.e. what delineates scientific investigation from other non-conclusive investigations, is subject to a discipline called *philosophy of science*. Hence, discussing the methodological background of this thesis requires to partly touch upon *philosophy*. Let me thus recall the relevant philosophical ideas and positions.

2.1.1 *Philosophy: A (short) historical account*

Western philosophy is traced back to ancient Greece. *Thales of Miletos* (~ 624–545 BC) and *Pythagoras of Samos* (~ 570—495 BC) are considered to be the first philosophers who started to reason about the world around them and attempted to explain it. Pythagoras³ and his followers believed in a deep relationship between the physical world and *numbers*, more concretely, that the former can be completely described by the latter. Their studies can be seen as the precursor to modern day mathematics.

¹The intention is not to understate the epistemological, social and cultural value of Eastern, Middle Eastern, African or Indigenous philosophy traditions. However, our contemporary scientific world is based on the western tradition.

²Isaac Newtons theory about gravitation is actually titled *Mathematical Principles of Natural Philosophy*.

³Pythagoras is credited for the geometric theorem bearing his name. However, the validity of the former statement had allegedly already been known by Babylonians and Indians long before him.

Method

Following the Pythagoreans, there are *Socrates*⁴ (~ 470—399 BC) and his student *Plato* (~ 423—348 BC), who founded the *School of Athens*, the first institution of higher education in the modern sense. Plato had a major influence in several areas of philosophy such as politics, ethics, as well as *epistemology* (i.e. the philosophical inquiry about the nature of “knowledge”) and *metaphysics* (the philosophical inquiry about the relation between “knowledge” and “reality”). His view on the nature of human knowledge is captured by the *Allegory of the Cave*: Physical objects are just “shadows” of so-called ideal *forms*, which can only be discovered by thought. *Aristotle* (~ 384—322 BC), Platos most famous student, is credited for the invention of the *syllogism* (logical deduction a.k.a. inference), which is the foundation for mathematical study of logic (“modus ponens”). In his epistemological philosophy, Aristotle questions the absolute role of thought alone and argues that insights are born by experiences *and* observation. Thus, establishing the empirical position within the metaphysical debate. Simultaneously, Aristotle admitted the existence of abstract concepts, called *categories*, which exist independently of experience.

The following times of the Roman Empire, the Migration Period, and the Middle Ages mark an almost 2000 years long period of “stagnation” w.r.t. new philosophical ideas, with a few notable exceptions such as the philosophical texts of Cicero or the *Scholastic*, which was a return to the classical Greek philosophy from the angle of Christianity. Eventually, the late 16th century saw new activity in the philosophical debate in the form of the controversy between *Empiricism* and *Rationalism*. *Francis Bacon* (1561–1626) is sometimes referred to as the father of the Empiricism, which postulates that all human knowledge is the result of experience and observation. Two other notable proponents of Empiricism are *John Locke* (1632–1704) and *David Hume* (1711–1776). Hume, nonetheless, is also credited for discovering an important problem of the empiricist standpoint, called the *induction problem* [242]. The latter acknowledges the fallibility of knowledge generation by experience: To generalise the experience of a range of subsequent events (e.g. “seeing white swans”) into a general rule of causality (e.g. “all swans are white”) may be incorrect due limited capabilities of the human perception⁵. Bacons contemporary *René Descartes* (1596–1650), also known for his contributions to mathematics (the cartesian coordinate system bridging geometry and algebra) and physics (reflection and refraction of light), took the opposing rationalist standpoint where thought and reason is the *only* valid source of truth.

Arguably, one of the most important texts in philosophy is the *Kritik der Reinen Vernunft* [266] (German: Critique of Pure Reason), written by the German philosopher *Immanuel Kant* (1724-1804). This highly influential work reconciles Empiricism with Rationalism: In Kant’s view, knowledge arises both from perception and reason. He argues that experience without terms is “blind”, and terms without experience are “empty”. The world consists of intractable things (noumena), which have properties that can be observed (phenomena). Knowledge embodying sentences are classified along two dimensions: *A priori* (before experience) vs. *a posteriori* (after experience) as well as *analytical* (i.e. *not* enlarging the existing knowledge base) vs. *synthetical* (i.e. enlarging the knowledge base). Kant claims that synthetical sentences a posteriori

⁴Socrates never wrote anything down and therefore his philosophy only exists in transcribed conversations with him and his students.

⁵Think of the well-known anecdote associated with the sentence: “All swans are white”.

are problematic since they might be subject to misperception and therefore seeks for synthetical sentences a priori. The latter is possible by applying a so-called *transcendental schema* to perceptions resulting in abstract terms, which represent *true* knowledge. The transcendental schema consists of terms, whose existence has to be taken as given and he resorts to the Aristotle's categories for this.

2.1.2 Science: The Demarcation Problem

At the beginning of the 20th century a group of scientists and philosophers around the physicist *Moritz Schlick* (1882–1936) gathered regularly at the University of Vienna to discuss philosophical issues related to scientific knowledge. They became known as the *Vienna Circle* and their position is known as *logical positivism*. It was motivated⁶ by the work of *Gottlob Frege* (1848–1925) [188], *Bertrand Russell* (1872–1970) [485], and *Ludwig Wittgenstein* (1889–1951) [491], whose texts on the foundation of logic provided a framework for the formulation of all areas of mathematics providing precise rules for the inference of true mathematical propositions from given axioms. The goal of the Vienna Circle was to create a similar foundation for the natural sciences and to leave the non-conclusive metaphysical discussion of Kant's idealism, i.e. demarcating science from philosophy [82]. The fundamental principle of their logical positivism approach is the *verification principle*: Just as true propositions in logic must be traceable to the axioms of the theory, valid statements in natural science must be traceable to empirical observations or logical implications of the former. This standpoint has been criticised by several thinkers with a reference to Hume's Induction Problem. In his book, the *Logic of Scientific Discovery* [389], *Karl Popper* (1902–1994) analyses the problems of logical positivism and proposes the *falsification* principle instead: Empirical observation can only refute and never confirm a theory because perception is fallible. Thus, the only way to gain knowledge is to propose a hypothesis (possibly motivated by former experience), derive its consequences and test these against reality (experiment). If the experiment does not refute the hypothesis, it can be accepted as "valid" until a counter-example has been found. In this case, the hypothesis has to be replaced by a refined version, which accommodates for the troubling example. This process is known as the *hypothetico-deductive* method and is widely accepted as *the* scientific approach in natural sciences.

As a response to Popper's standpoint, a group of historians claimed that science, as a human activity, is at its core a social phenomenon. In his book, *The Structure of Scientific Revolutions* [300], *Tomas Kuhn* (1922–1996) claims that the way how science is conducted in a certain field is always based on a *paradigm*, i.e. there is a given process or framework that defines how issues are to be addressed or as he calls it: "how puzzles are solved". When the paradigm fails to guide the solution for an issue, a new paradigm may arise. If the new paradigm gathers enough consensus, a *revolution* occurs and the old paradigm is replaced by the new one. Hence, the truth of scientific knowledge is not chiefly based on the congruence between theory and reality but based on *consensus* in the scientific community. The social-relativist position has been put to its extreme by *Paul Feyerabend* [170]. In his view, there is no difference between scientific knowledge

⁶Reportedly, *Sigmund Freud's* emerging *Psychoanalysis*, which was in turn heavily inspired by Nietzsche, formed another motivation for the Vienna Circle: They wanted to refute it as "unscientific".

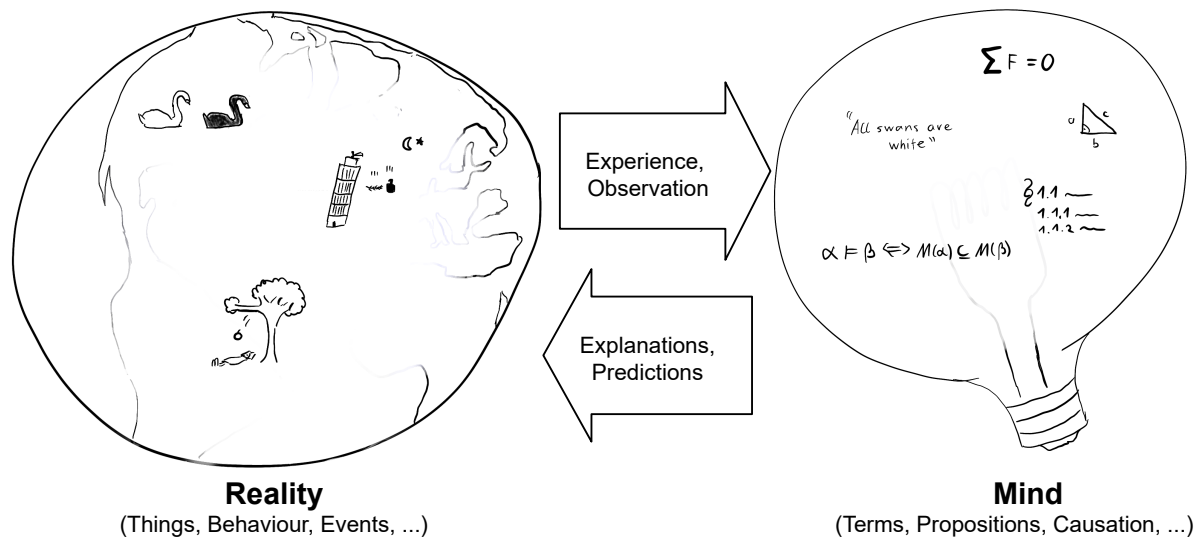


Fig. 2.1: Model: Philosophy of Science

and other forms of knowledge: “Anything goes” – a position seen critical by many researchers.

Philosophy of Science is thus characterised by a conflict between *realism* and *non-realism*. To its extremes, there are *naive realism* (“things are exactly as we seen them”) and postmodernist *relativism* (“reality is an illusion”). Popper and Kuhn can be seen as moderate representatives of either position and the *constructive empiricism* [471] of *Bas van Fraassen* resembles a compromise between realism and non-realism, which is agnostic about the true nature of reality but acknowledges the utility of scientific theories to explain phenomena of the real world. Indecently of where one positions himself in this dispute, it is important to be aware of the fact that the question “What is Science?” cannot be answered scientifically but only philosophically.

To summarize the central ideas of this section consider Fig. 2.1: There is a physical reality to the left comprising events, things, their behaviour, etc. which is translated into the artificial world of the mind to the right comprising propositions, terms and laws. The latter makes predictions about reality. A *theory*, as considered in the mathematical sense, is a set of propositions (axioms) together with all the propositions that can be derived from the former [491]. Science is all about true propositions over reality and thus a theory is admissible if all propositions in the theory correspond to a respective observable state of affairs in reality (Popper). This meta-paradigm is also widely accepted in the scientific community (Kuhn). Thus, I take these two aspects as the indicators for scientific work.

2.1.3 Research Methodology

Each research domain has its set of *scientific methods* (paradigm). Scientific methods are the means for creating theories. Standard textbooks [101] generally distinguish between *qualitative* methods, *quantitative* methods or a combination of both.

Qualitative research methods are mainly used in social sciences [97, 195] and are based on collecting non-numerical data, i.e. texts, interviews, audio recordings etc. It is often used in less-understood contexts to inductively generate hypotheses.

Quantitative research methods are mainly used in natural sciences but are also used in social sciences (e.g. hypothesis testing). They are based on mathematics, i.e. based on theories that are formalised in the language of mathematics. Consequences of the theory axioms are derived using the framework of logic and tested against reality by gathering numerical data in a controlled environment. The usage of mathematics for the formulation of natural theories goes back to *Isaac Newton* (1642–1726), who proposed to use differential equations to describe the behaviour of physical objects [359]. Phenomena in social sciences and many other empirical sciences are commonly described using the mathematical language of statistics and probability theory.

Mathematics itself, i.e. the rigorous application of logic to confirm or refute a proposition w.r.t. given axioms, is purely analytical and thus does not qualify as science in Popper's sense. Still, mathematics is an important *supporting* discipline for (empirical) science.

Finally, there is another – *secondary* – type of scientific inquiry. While qualitative and quantitative inquiries are seen as *primary* studies, i.e. they immediately generate new hypotheses or support existing ones, literature study represents a secondary study, which may provide new insights. Formalised examples of such literature studies are *systematic literature reviews (SLRs)* and *systematic mapping studies*. This secondary research method originated in medicine but has now found broad acceptance in other disciplines such as software engineering (SE) [74, 139, 277].

2.2 Research Methodology in Constructive Sciences

Young software engineering researchers sometimes “struggle” to defend their work against methodologists. This is due to the fact that the traditional notion of quantitative and qualitative research methods does not exactly match the scientific work practice in Computer Science, Software Engineering and other Engineering disciplines such as Electrical Engineering or Mechanical Engineering because these disciplines are *constructive*. Engines, Integrated Circuits, Computers, the internet, these are examples of *artificial* things. None of these things existed 2000 years ago and one cannot consider them as a “natural” phenomenon. They are, however, built via intelligent combination of existing theories. Taking *the computer* as an example: Two concrete problems, namely “counting” and “measuring land”, motivated the inception of “arithmetic” and “geometry”, respectively, which later became mathematics. As the analytical discovery in this discipline progressed, the idea of a “machine doing arbitrary computations” (i.e. mechanical symbol manipulations) was born. Human ingenuity combined this idea with vacuum tubes and later semi-conductors to create the first computers. Semi-conductors, in turn, were made possible by theory about electricity and atoms developed by physicists before. Yet, the final product was something that did not exist before and its advent changed the physical reality, i.e. way people work and communicate. Also, it created novel phenomena that had not been seen before. Thus, theory can also be used to change reality and not only explain it. In the picture shown in Fig. 2.1, the arrow going from right to left can be extended to capture this notion: Empirical research is *descriptive* (reality is explained by means of theories), formal research is *analytical* (the theory is analysed), and constructive research is *prescriptive* (theories change reality). The reader interested in a comprehensive account on the

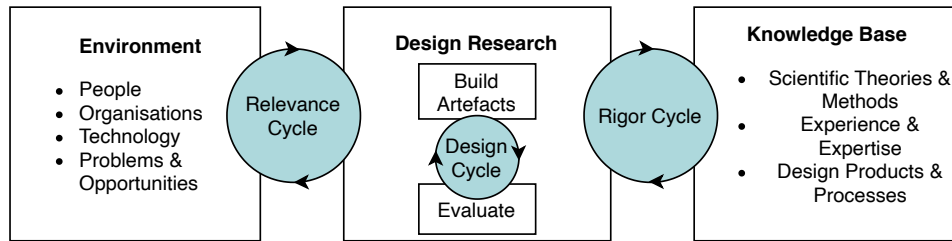


Fig. 2.2: Design Science Framework [231]

science of the artificial is referred to the seminal book by Simon [429].

Design Science is a (comparably new) research methodology [374], which originated in the information systems research discipline [360, 477]. It provides a general framework (paradigm) for constructive research such as Engineering [140] and Computer Science [248] but has been applied to other domains as well, see [9, 374]. The underlying paradigm behind Design Science is *problem solving* through the creation of artefacts. The respective type of knowledge, called *design theories*, has been discussed by several researchers [208, 232, 248, 299, 335, 336]. A design theory prescribes how an artefact is created and should be grounded in existing kernel theories [208], i.e. the theories created by the empirical sciences.

The Design Science framework is best explained along the three-cycle concept from [231], which is shown in Fig. 2.2. There are three areas: *Environment*, *Design Research*, and *Knowledge Base*. The Environment bears the motivation to perform design research, i.e. it poses a concrete problem. Design Research is the central element in the framework and comprises two activities: Building artefacts and evaluating them. Both activities are iterated in a design cycle [336]. Here, evaluation means to demonstrate the utility of the artefact w.r.t. the problem posed by the environment, e.g. by field testing. This constitutes the relevance cycle. Simultaneously, the creation of artefacts is grounded by an existing knowledge base. The knowledge base contains kernel theories, which had been developed by (empirical) science, and design artefacts, which had been produced by other instantiations of the Design Science framework. An important aspect of Design Science is that the design process always contributes back to the knowledge base, which distinguishes it from a routine design activity. It constitutes the rigour cycle.

Finally, Hevner et al. [232] identified seven guidelines that characterise effective Design Science research:

1. Design as an Artefact: Design Science must always produce a purposeful artefact.
2. Problem Relevance: The goal of Design Science is to solve an important and relevant problem.
3. Design Evaluation: The utility, quality, and efficacy of a design artefact must be demonstrated via evaluation methods.
4. Research Contributions: Design Science research must provide verifiable contributions in the areas of the design artefact, design foundations, and/or design methodologies.

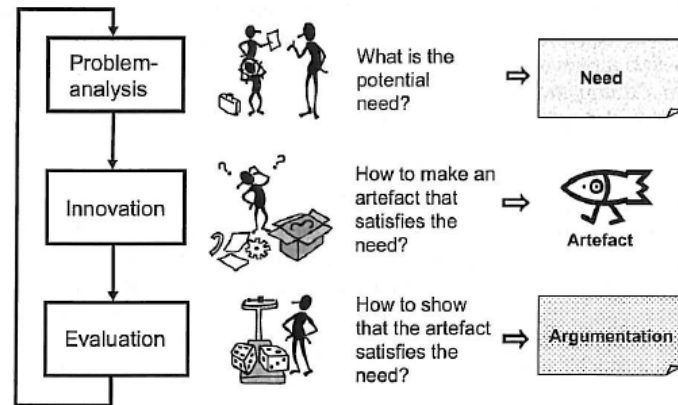


Fig. 2.3: Technology Research [430]

5. Research Rigour: Design Science research must apply rigorous methods in both the construction and evaluation of the artefact.
6. Design as a Search Process: The search for an effective artefact requires utilizing all available means of the possible solution space.
7. Communication of Research: The research must be effectively communicated.

2.3 Research Methodology in Software Engineering

To connect the methodological discussion above with the remainder of this thesis, the research methodology in Software Engineering must be assessed. According to [232], there are two, equally important, kinds of research in Software Engineering: *Behavioural Science* and *Design Science*.

Behavioural Science is effectively social science and its focus are people and their relation to software technology. This branch uses the same methods (quantitative and qualitative) for theory building and validation as found in other empirical sciences. The behavioural approach takes technology as given and investigates its socio-organisational consequences by using data collection and respective analysis techniques. The results are theories and explanations (description-driven), e.g. “How does the application of object-oriented programming influence the productivity of developers” [29]? This branch of SE research is also known as *Empirical Software Engineering* [139, 260].

On the other hand, there is the Design Science branch of software engineering research. Here, technology is the goal (prescription-driven). The focus lies on creation and evaluation of software artefacts intended to solve a given problem. Solheim and Stølen describe the conceptual process, depicted in Fig. 2.3, for this particular way of doing research [430]. This process starts with a concrete problem for which there are no existing solutions or for which no existing solutions are adequate, see the Relevance Cycle in Fig. 2.2. After a thorough analysis of the problem, a novel *artefact* is created. This “innovation” activity is expected to be based on existing theories, see the Rigour Cycle in Fig. 2.2. Finally, the artefact must be evaluated in terms of how it addresses the initial problem and to what extent it overcomes the limitations of existing artefacts, i.e. whether it represents a “better” artefact. In general, Computational and

mathematical methods are used to evaluate the quality and effectiveness of the artefacts [335]. The latter distinguishes SE research from “normal” SE work practice (building software systems). March and Smith [335] distinguish four types of artefacts that are produced in technology research: *constructs*, *models*, *methods* and *instantiations*. Constructs provide a language to formulate and communicate problems and solutions. Models are abstract representations of real world situations, using constructs, which aid problem understanding and eventually solution conception. Methods provide guidance on how to solve a problem. Instantiations are prototypes or final products that demonstrate the solution feasibility.

Hevner et al. [232] emphasize that the Behavioural and Design Science part are equally important. Too strong a focus on Behavioural Science would result in theories based on outdated or ineffective technologies, while too strong a focus on Design Science would result in a surplus of artefacts that are useless in practical scenarios due to a lack of theoretical foundations.

2.4 Research Methodology in this PhD project

In the light of the above discussions, the work presented in this thesis clearly falls into the Design Science category. Hence, I will position myself around the process described in [430] and depicted in Fig. 2.3.

The problem that I am addressing (multi-model consistency management) is presented in Sec. 1.4 and motivated by two concrete problems from software engineering work practice presented in Sec. 1.3. The problem scenarios will not be elucidated by means of Behavioural Science as this would largely exceed the scope of this thesis. I assume that these two problem are relevant issues that both software engineers and SE researchers will relate to.

During the course of the PhD project, three artefacts were created that are present in part II of this thesis:

- A1** A conceptual model of multi-model consistency management (Chap. 3),
- A2** a formalism, called *comprehensive systems*, suitable for representing problem and solution domain (Chap. 5), and
- A3** prototype tool implementations, called CORR_{LANG} (Chap. 6).

These artefacts can be associated with the three activities, *understanding* the problem, *representing* it in an abstract way, and *creating* a concrete solution. Using the characterisation of artefacts given in [335], **A1** would be classified as a construct, **A2** as a model, and **A3** as an instantiation. The creation of these artefacts was preceded by an in depth-study of the state of the art, which is presented in Chap. 4. The scientific value of the created artefacts, i.e. whether they represent a new or improved solution w.r.t. the initial problem, is assured by a detailed validation, which is presented in Chap. 7. Simultaneously, I adhered to the Design Science guidelines formulated in Sec. 2.2.

Part II

CONTRIBUTIONS

“Wovon man nicht sprechen kann, darüber muss man schweigen. (germ: Whereof one cannot speak, thereof one must be silent.)”

—Ludwig Wittgenstein [491]

CHAPTER 3

CONCEPTUALISATION

The contribution part of this thesis starts with a conceptualisation of the problem domain. The resulting conceptual framework facilitates a better understanding of the concepts, activities and challenges within multi-model consistency management. Further, it will allow to compare and integrate results from related research domains. Another useful outcome is a unification of the diverse terminology. This chapter thus paves the way for a comprehensive literature, presented in the following chapter, and forms the foundation for the formalisation and tool development presented later.

In Sec. 3.1, I will begin with a short historical account of central concepts and ideas that are related to multi-model consistency management. Sec. 3.2 presents these concepts within a generic *model management* framework, which has originally been developed by Diskin, Maibaum and their collaborators in the course of an industrial collaboration project on model-based engineering in the automotive domain and communicated in a series of publications: [119, 125, 128, 130]. Finally, Sec. 3.3 adapts the generic model management framework to the more concrete setting of multi-model consistency management. The latter marks my main contribution of this chapter, which was *partly* presented in [451].

3.1 Existing Concepts & Ideas

Multi-model consistency management is closely related to the following research domains: View-based Software Development [175], (In)consistency management [433], Traceability Management [352], Multi-view Modeling [87], Metamodeling [23], Model Transformation [264], Model Management [45], Megamodeling [167], Model Repair [331], Model Synchronisation/bidirectional transformations (BX) [103], and Coupled Evolution [307]. These topics will be investigated in greater detail in Chap. 4. In this chapter, I am mainly focusing on the most fundamental ideas.

3.1.1 View-based Software Development

The issue of maintaining the consistency among distributed parts of a system specification is older than the MDSE discipline. It was first thoroughly investigated in the field of *Requirements Engineering* during the nineties [175, 497]. One well-known approach of that time is the *ViewPoints* framework [175, 362]. It considers a system specification as a collection of loosely coupled viewpoints.

Conceptualisation

A viewpoint, see Fig. 3.1, captures the perspective and knowledge of a stakeholder on a particular part of the system specification, see the aspects in Fig. 1.7. It comprises five so-called *slots* [175]: The *representation style* denotes the description language; The *domain* describes the area of concern captured by this viewpoint; The *specification* slot contains the current state of the partial system specification; The *work plan* describes the actions that can be used for creation and verification of the specification together with a guiding process that states in what order these actions should be invoked. The *work plan* contains the complete development history. The number of viewpoints is not fixed and new viewpoints may be instantiated during later stages of the development process using a pre-defined *template*. The *ViewPoints* framework emphasizes loose coupling between viewpoints and heterogeneity of representation styles to delineate itself from earlier approaches such as [480, 497], which were based on common data models, uniform notation and centralised databases. Those approaches are considered more complex and hard to extend in the long run, see [174].

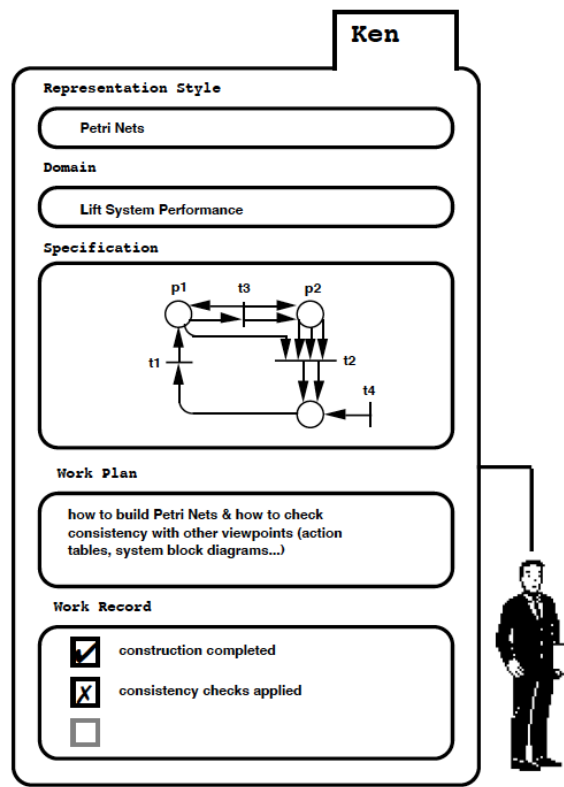


Fig. 3.1: Viewpoint [175]

3.1.2 (In)consistency Management

An important activity concerning viewpoints is *consistency verification*, which, in case of the *ViewPoints* framework, is performed by invoking so-called *check actions*. There are two types of check actions [174]:

in-viewpoint checks verify the consistency of the specification within the respective viewpoint.

inter-viewpoint checks verify the consistency of the specification w.r.t. to a specification developed by another viewpoint.

For implementing these checks, Finkelstein et al. propose a *logic-based* approach [174]: A viewpoint specification can be interpreted as a set of sentences in a suitable logic such as first order logic (FOL). Verifying consistency *internally to a viewpoint* means to check whether the set of sentences, which results from translating the viewpoint specification into a respective logical representation, is free of contradictions. Verifying consistency *between two viewpoints* means to translate both viewpoints into a logical representation and checking for the absence of contradictions. This includes the necessity to detect *overlaps* between the two specifications, i.e. both viewpoints may refer to

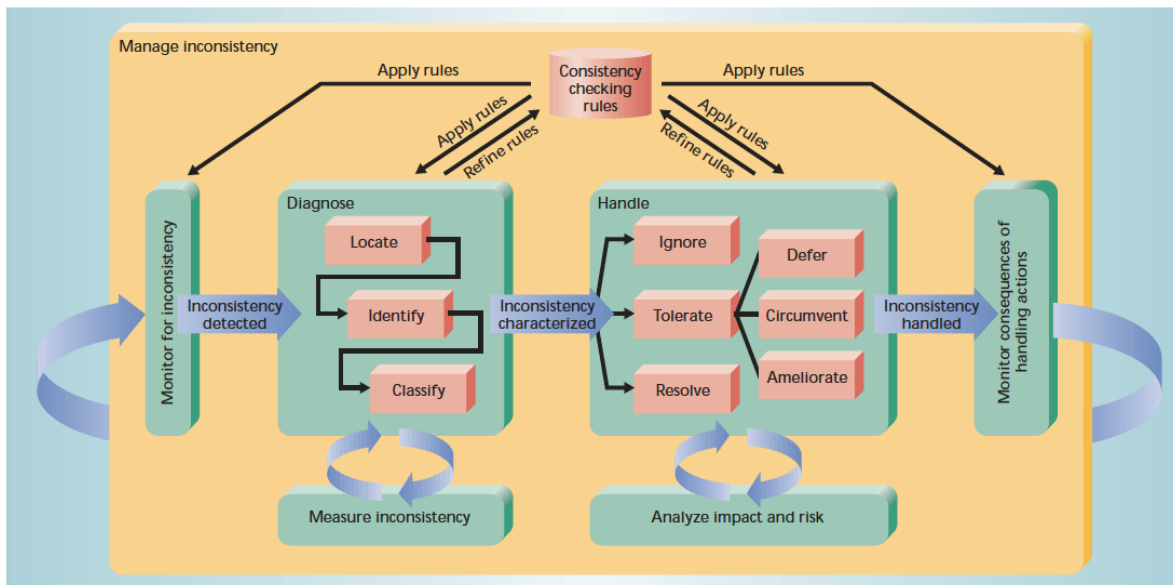


Fig. 3.2: Inconsistency Management Framework [361]

the same elements in the underlying system [176]. Spanoudakis et al. [432] identified different types of overlaps that may exist in distributed system specifications:

null There are no overlaps between the specifications

partial Both specifications have a “shared part” but also contain elements unknown to the respective other,

inclusive One specification is completely contained in the other (part-of relationship)

total Both specifications are isomorphic.

According to Finkelstein et al. [176], overlaps imply *consistency rules*, compare Sec. 1.3.1 (CR1). The violation of a consistency rule is called an *inconsistency*. Handling inconsistencies is a major issue within software engineering. Nuseibeh et al. [361] developed a comprehensive framework for *inconsistency management*, which is depicted in Fig. 3.2.

At the centre of this framework, there is a set of global consistency rules. It is important to note that the set of these rules is not immutable, i.e. new rules may be introduced due to changed requirements while others may become obsolete over time. The distributed system specification is constantly being monitored w.r.t. these rules. When an inconsistency is *detected*, it is first *diagnosed*, i.e. additional metadata (location, type of violation, affected elements, and impact) is collected. Afterwards, the inconsistency is *handled*. Inconsistencies generally have a negative impact on the quality of the final system, and thus it is desirable to *resolve* them. However, it may not always be possible yet reasonable to try to resolve inconsistencies immediately [361], e.g. due to fundamental conflicts concerning the set of consistency rules. Thus, “tolerating inconsistency” [30] becomes a viable alternative and the stakeholder may decide to [361]

defer the decision to later,

Conceptualisation

circumvent the inconsistency by changing a consistency rule,

ameliorate the inconsistency (i.e. only resolving it *partially*), or

ignore the inconsistency.

The outcome of this decision may be codified in the form of a *policy* [176]. Policies reside on top of consistency rules and stipulate how an inconsistency w.r.t. specific consistency rules should be handled. Finkelstein et al. [176] distinguish the following types of policies:

Preventive Policies prohibit any action that would cause an inconsistency,

Toleration Policies define when an inconsistency should be tolerated, and

Remedial Policies always trigger inconsistency resolution. They may also provide extra information to the resolution about *how* the inconsistencies shall be resolved.

The early literature on view-based software development and (in)consistency management has been surveyed by Spanoudakis and Zisman in their seminal paper [433]. In this paper, the “inconsistency monitoring” step from [361] is refined further into the activities *overlap identification* and *inconsistency detection*. These activities are related to the notion of *pre-traceability* and *post-traceability* mentioned in [211], i.e. the phases before and after traceability relationships have been established among the various artefacts of the system specification.

3.1.3 Traceability Management

Overlaps are a special case of *traceability* relationships. Traceability is the common term for any kind of relationship among software artefacts, see Def. 1.3 and the examples in Sec. 1.3.2. Traceability is generally expressed through relationships between elements of the same model (vertical) or through relationships between elements of disparate models (horizontal) [316]. Spanoudakis and Zisman [434] identified eight different types of traceability relationships:

Dependency One element depends on the existence of another element, e.g. because it accesses a feature of the other element.

Generalisation/Refinement One element is broken down into smaller and more detailed parts. Elements related by generalisation/refinement relationships reside on different abstraction levels.

Evolution An element has evolved throughout the development process, think of a version history.

Satisfiability One element meets the expectations imposed by another element.

Overlap Two elements share a common part or are identical.

Conflict Two elements are in conflict with each other, i.e. they make contradictory assertions.

Rationalisation One element explains the rationale behind the creation, deletion or evolution of another element.

Contribution An element shows which stakeholder has contributed to its current state.

3.1.4 Consistency Management in UML

Since its inception, the Unified Modelling Language [405] has become the de-facto standard for formulating software specifications. UML is organised as a “family” [96] of 14 sub-languages (diagram types), which are divided into two categories: *structural* and *behavioural* [37]. Each diagram type focuses on a specific system aspect. For instance, *class diagrams* (structural category) define the system’s *classes*, i.e. building blocks of object-oriented software systems representing data types or domain entities. *Sequence diagrams* (behavioural category) define the interaction between two or more class instances. UML-based system specification can thus be considered as an instance of the view-based paradigm [175]: Every diagram type represents a viewpoint of the same underlying system. Hence, there is a possibility of these views becoming inconsistent [245]. For instance, deleting a method in a class might render a sequence diagram, which refers to this method, as inconsistent.

Consistency management of UML models has been addressed by many researchers. One notable mention is the pioneering work of Egyed [142] and his *VIEWINTEGRA* approach. Consequentially, this research domain has been surveyed in several publications [37, 245, 284, 320, 460, 469], which developed further classifications of the various approaches in this area. Aside from the obvious technical distinctions such as UML-version or diagram type, Engels et al. [155] distinguish four types of consistency problems along two orthogonal dimensions. On the first dimension, they distinguish between *vertical*, i.e. consistency of one UML diagram w.r.t. its metamodel and *horizontal* consistency, i.e. consistency among multiple diagrams. This distinction corresponds closely to the notions of in-viewpoint and inter-viewpoint checks in [174]. On the second dimension, Engels et al. distinguish between *syntactical* and *semantic* consistency. Syntactic consistency verifies structural well-formedness, i.e. that the UML model conforms to the abstract syntax of the UML metamodel. Semantic consistency additionally concerns the model’s behaviour. In the early days of UML, the semantic behaviour of UML models was not formally specified, which opened up for various domain specific interpretations. Nowadays, UML models are often augmented with OCL-invariants [479] to express domain specific behaviour.

3.1.5 Multi-View Modeling

Views do not only play an important role in UML, but also for DSLs [209]. The research domain that investigates the topic of distributed system specification using arbitrary (general-purpose or domain specific) modeling languages is called *multi-view modeling* [68, 87]. Studies in this domain often refer to the *IEEE 1471 | ISO/IEC 42010* standard [246] as a conceptual basis. That standard defines a conceptual framework for software architectures which comprises the concepts *models*, *views* and *viewpoints*: The model is an abstract description of the system under development; The views represent

specific parts of this model; and a viewpoint is a selection of views. Furthermore [246] distinguishes between two approaches: *projective* and *synthetic*. In a projective approach, the system model exists from the very beginning and every view is a projection of it. UML is a representative of a projective approach. In a synthetic approach, views are a priori independent. Thus, the system model exists only theoretically in the beginning. Eventually, it has to be created by *composing* the views. This synthesis generally necessitates the detection of overlaps among the views. The *ViewPoints* framework [175] is a representative of a synthetic approach. Moreover, there is a proposal (*Orthographic software modeling*) that seeks to combine projective and synthetic approaches [25], which is based on the idea of a *single underlying model (SUM)*. The SUM must be synthetically created from independent artefacts in the beginning and can subsequently be used for deriving projective views.

3.1.6 Metamodeling

Models are denoted in a *modeling language* (representation style in Fig. 3.1. Types of modeling languages can be classified along two dimensions: “style” and “domain”. On the “style”-dimension, one generally¹ distinguishes between *graphical* and *textual*² languages. On the “domain”-dimension, one distinguishes between *general-purpose* and *domain specific* languages. UML is an example of a graphical general-purpose modeling language while the “state machine language” from Fowler’s DSL-book [180] is an example of a textual domain specific language.

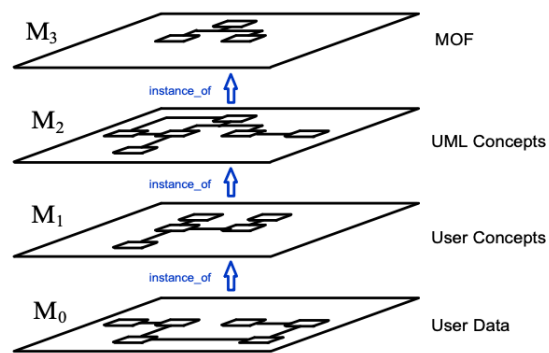


Fig. 3.3: Four-level Meta-Hierarchy [23]

A modeling language is described by a *metamodel*, i.e. an abstract description of the modeling language itself. A metamodel can be seen as a generalisation of an (E)BNF-grammar [28], which are used to define the syntaxes of textual languages. A metamodel contains a definition of the *abstract syntax* of the language, i.e. the *concepts* of the language, their *relationships* and *well-formedness rules* define over them. The conceptual idea of a metamodel has been thoroughly scrutinized during the early development stages of OMG’s MDA [23, 76, 162, 163], which resulted in the *four-level meta-hierarchy* shown in Fig. 3.3. The bottommost level M0 contains abstract representations of “things” in the real world. The types of representable objects are defined by software models (e.g. UML diagrams) which reside on level M1. The language for denoting these models is represented by a metamodel, which resides on level M2. Finally, metamodels must themselves be denoted in a meta-modeling language. In MDA, this meta-modeling language is called by MOF [368] (actually a

¹There are also table-based and tree-based representations. Both can be considered as special cases of a graphical notation and Sec. 5.1.1 will introduce a way to treat all kinds of styles uniformly.

²Therefore, the line between source code and model becomes blurry, which further corroborates the stance that “everything is a model”

subset of the UML class diagram language), which resides on level M3. The relation between a model and its metamodel is called *instanceOf*.

The four-level meta-hierarchy has been criticised from several researchers. The main criticisms are that the *instanceOf*-relation mixes *linguistic* and *ontological* aspects [23, 303] and that the restriction to exactly four sequential levels poses as an unnecessary limitation. Therefore, Atkinson and Kühne [24] coin the concept of *multi-level modeling*, which allows an arbitrary number of metalevels in addition to different dimensions, which represent ontological and linguistic concepts separately.

3.1.7 Model Transformation

A key characteristic of MDSE is that models are used *actively* instead of being merely a passive means of documentation. This feature is facilitated by the concept of *model transformations*, which have been considered to be “the heart and soul of model-driven engineering” [426]. In the most general sense, a model transformation is a program that takes one or more models as input and produces one or more models as output [61]. The idea originated from the MDA-vision [192], which envisaged a three-step process starting with *computation-independent models (CIMs)*, which are first transformed into *platform-independent models (PIMs)* and then finally into *platform-specific models (PSMs)*, where each transformation step refines the system specification by adding the necessary technical details. However, the implementation of these model translations was not explicated further. Thus, in 2002 the OMG released a *request for proposals* entitled *Queries / Views / Transformations (QVT)* to address this issue. Several proposals from academia and industry were made and eventually the OMG adopted a standard known *QVT* [367], but which still has no official and complete functioning implementation due to inherent semantic problems [439]. Neither MDA nor QVT gained a broad adoption by software practitioners. Albeit, they influenced the plethora of model transformation tools that exists today, e.g. the well-known *ATLAS Transformation Language (ATL)*³ [263]. Kahani et al. [264] contains a comprehensive overview over existing model transformation tools and approaches. A classification scheme for model transformation approaches is given in [104] and [342]:

endogeneous vs. exogenous [342] In an endogenous model transformation, input and output models are denoted in the same language (instances of the same meta-model) while in an exogenous model transformation in- and output models are denoted in different languages.

horizontal vs. vertical [342] In a horizontal model transformation, in- and output models are on the same abstraction level (e.g. refactoring), while in a vertical model transformation, abstraction levels differ (e.g. refinement).

unidirectional vs. bidirectional [342] In a unidirectional model transformation, there is a strict distinction between input (source) and output (target) models. A bidirectional model transformation allows to perform the transformation in the other direction as well. The latter is associated with its own research domain, which is explained further below.

³<https://www.eclipse.org/atl/>

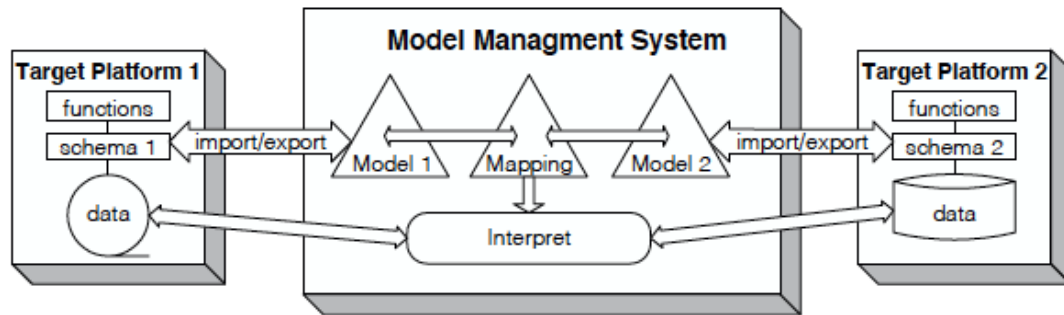


Fig. 3.4: Model Management [46]

in-place vs. out-place [104] A model transformation may either *update*, i.e. in- and output model coincide, a model (in-place) or *create* a new model (out-place).

imperative vs. declarative [104] The definition of the model transformation may follow different paradigms: An imperative model transformation definition is given in terms of a procedural or object-oriented program [291]. Declarative approaches are generally based on relations [8] or graph-transformation rules [17]. They provide a high level of abstraction but may be less-expressive than their imperative counterpart. Finally, there are hybrid approaches, which aim to combine benefits from both imperative and declarative approaches.

Before a model transformation can be executed it must be *defined*, see the conceptual picture in Fig. 3.5b. This definition is “located” on the level of metamodels, i.e. it contains references to the source metamodels (which are queried) and the target metamodels (which are instantiated). The execution of a model transformation creates a so-called *trace*, i.e. a set of links that relate elements of the source models to elements of the target models. The alignment of multiple models together with traces has been referred to as *model weaving* [114]. The later is also closely related to model traceability, see Sec. 3.1.3.

3.1.8 Model Management

Model transformations is one instance of a *model management* operation. Model management was first introduced in the database domain [46]: With the growing importance of web-related technology, developers had to manage increasingly more heterogeneous artefact, i.e. database schemas and XML document type definitions. Thus, Bernstein et al. [45, 46] proposed the concept of model management. The conceptual idea is sketched in Fig. 3.4: Schemas are understood as *models*, which are related by *mappings* and processed by *operations*. The paradigmatic idea behind model management is a transition from “element-at-a-time”-processing to “model-at-a-time”-processing. The analogy in the database world is “record-at-a-time”-processing versus “table-at-a-time”-processing, which was made possible with the advent of SQL, a declarative language. Bernstein proposed a *catalogue* of essential model management operations in [45]. This catalogue was introduced to the MDSE community and extended by Brunet, Chechik and their collaborators in [70]. The composite list of model management operations is given below:

match [45, 70] takes two models as input and produces a mapping between them, e.g. by detecting overlaps.

merge [45, 70] takes two models and a mapping among them as input and produces a single model, which provides an integrated view on both models wherein mapped elements are identified.

diff [45, 70] takes two models as input and identifies the *edit actions* (i.e. syntactic descriptions of model modifications), that were applied to transform one model into the other containing only elements that satisfy the criterion.

slice [70] takes a model and criterion as input to produce a model, which is a projection of the input model.

split [70] takes a model and a criterion as input to produce a partition. The partition is given as a pair of models related by a mapping.

check_property [70] takes a model and a property (e.g. a consistency rule) as input and verifies whether the model satisfies the property.

is_consistent [70] takes two models and a mapping between them as input and verifies whether they are consistent concerning overlaps.

patch [70] takes a model and a sequence of edit actions as input to produce an updated model.

propagate [70] takes a source model, a target model and a sequence of edit actions on the source model as input to produce a sequence of corresponding edit actions for the target model.

model_gen [45] takes one model as input and produces a new model as output (see out-place model transformations).

compose [45] takes two compatible mappings as input to produce a new composed mapping.

Rondo[340] and *Clio*[216] are examples of the first fully-functional model management tools, which have been applied in industrial contexts and inspired commercial *Extract-Transformation-Load (ETL)* tools, which are heavily used in data warehousing today. In the MDSE community, *Epsilon* [371] and the ecosystem around the *ATL* transformation tool [47] are popular examples of academic model management tools.

3.1.9 Megamodeling

Researchers have made several attempts to combine the concepts from metamodeling, model transformation and model management into one conceptual framework known as *megamodel*. A megamodel is a model whose elements are models themselves [79]. Arguably, the most well-known attempt is the proposal by Favre [164–167], which is depicted in Fig. 3.5. The building block of this framework are *systems* (Fig. 3.5a), which are either real physical systems, digital systems (i.e. software systems) or abstract

Conceptualisation

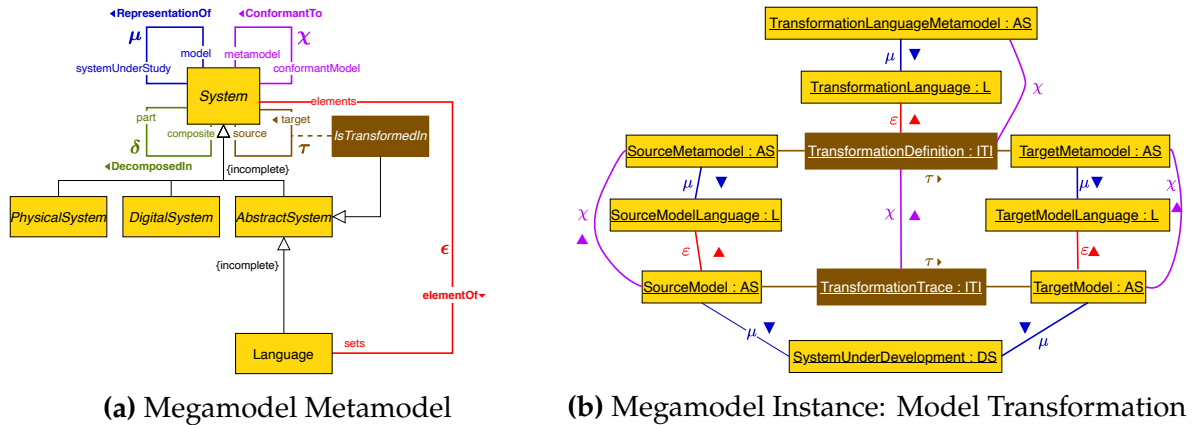


Fig. 3.5: Favre's megamodel, adapted from [164–167]

systems (models). Systems are related by various types of relationships abbreviated by Greek letters, see the associations in Fig. 3.5a. The *representationOf* relationship (μ) expresses that one system is an abstract representation (model in the classical sense) of the other. The *elementOf* relationship (ϵ) relates two systems where one system is a (modeling) *language*. The *conformsTo*-relationship (χ) arises from $\mu\epsilon$ -sequences and captures the (purely) linguistic relation between a model and its metamodel. Finally there are *decomposedInto* (δ) and *transformedInto* (τ) relationships, which express refinement and evolution, respectively. In the spirit of MDA, the development process is represented through a sequence of τ relationships. It is important to note that a transformation is considered to a system as well which represents the transformation trace or the transformation definition respectively. Fig. 3.5b contains an example instance of the megamodel, which depicts a model transformation. This megamodel was an influential idea in the MDSE community but possesses, with few exceptions (*ATLAS MegaModel Management (AM₃)* [11]), almost no tool implementations.

3.1.10 Model Repair

The activity of (semi)-automatically resolving inconsistencies has several synonyms such as *restoration* [442], *fixing* [143], or *repairing* [288]. In the following, I will mostly stick to the terminology in [331] and call this activity *model repair*. One of the first mentions of model repair was within the *ViewPoints* framework [174], where consistency rules are augmented with *repair actions* that are being executed when the respective rule becomes violated. This idea of repair actions was continued in *Xlinkit* [357], which applies a similar idea to collections of XML documents inter-linked via *XLinks* [113]. In the context of UML, Egyed's *Model/Analyzer* [395, 396] marks a major contribution. This tool automatically calculates several possible consistency-restoring edit actions such that the user can choose one of them. There is a whole research domain centred around this topic. Hence, there are many more proposed solutions based on various strategies, such as *searching* [281, 488], *model finding* [328], *logic programming* [156], *description logics* [341], *graph grammars* [194], or *machine learning* [35]. An up-to-date survey over model repair approaches is the one by Macedo et al. [331].

3.1.11 Bidirectional Transformations

Model synchronisation [19] is a special case of model repair, which involves multiple models, see Sec. 1.3.2. It is known as *update propagation* [105] investigated by the cross-disciplinary research domain called *bidirectional transformations (BX)* [103]. The theoretical origin of BX is traced back to the *view-update problem* in databases, which was thoroughly investigated in [31]. A database *view* is a reduced instance, which is derived from a *source* database. An update on the source can easily be translated to an equivalent update on the view via projection. The translation from a view update to a source update is more complicated. A general approach to this problem is the so-called “*constant complement*”-approach [31]. This means that the source can clearly be divided into the view-part and a complement-part. The latter remains unchanged under a view-update.

In general, the problem of keeping two representations of the same data synchronised appears in many domains, e.g. *user-interface development*, *(functional) programming languages*, and *software engineering*. In the programming language community, it was investigated by Foster, Pierce and collaborators [32, 53, 54, 178, 238, 239]. The most important outcome of their investigation is the notion of a *lens* [178]. The simplest form of a lens is given by a *asymmetric set-based lens (as-lens)*:

Definition 3.1 (Asymmetric set-based) lens [178]

A lens is a quadruple $(S, V, \text{get}, \text{put})$ comprising a set of *sources* S , a set of *views* V , a unary function $\text{get} : S \rightarrow V$, and a binary function $\text{put} : V \times S \rightarrow S$. The lens is called *well-behaved* if and only if the following two conditions hold:

$$\forall s \in S, v \in V : \text{get}(\text{put}(v, s)) = v \quad (\text{GETPUT})$$

$$\forall s \in S : \text{put}(\text{get}(s), s) = s \quad (\text{PUTGET})$$

The lens is called *very well-behaved*, if and only if additionally the following holds:

$$\forall s \in S, v, v' \in V : \text{put}(v', s) = \text{put}(v', \text{put}(v, s)) \quad (\text{PUTPUT})$$

Asymmetric set-based lenses are closely related to the view-update problem. The *round-tripping-laws* (GETPUT) and (PUTGET) guarantee that the source and target remain *consistent*. A view v is considered consistent with a source s if and only if $v = \text{get}(s)$. When a lens is very-well behaved, it means that the lens implements the constant-complement approach, see [253, 256]. Lenses provide the formal foundation for several functional programming libraries offering synchronisation primitives and have been generalised into several directions, the most fundamental concepts are:

Delta-lenses [134] treat updates as a structural entity (= *delta*) of its own right, i.e. *get* and *put* propagate deltas instead of states.

Edit-lenses [239] are similar to delta-lenses but instead of considering changes *structurally*, they are treated *operationally*.

Symmetric-lenses [238] dissolve the division in source and view. There are two different domains, where none is derived from the other. Thus, there are

put-functions in both directions.

Johnson and Rosebrugh [255, 258] showed that, under certain conditions, the different types of lenses can be translated into each other, e.g. set-based lenses are a special (codiscrete) case of delta-lenses and symmetric-lenses can be expressed as pairs of asymmetric lenses. Apart from that, there are proposals for even more general lens-definitions regarding *uncertainty* [122], *learning* [120], and *multi-ary situations* [126].

In MDSE domain, model transformation, see Sec. 3.1.7, is seldom a “one-way” process [123]. For example, code that is being generated from a more abstract description will eventually contain modifications by the developers. When the abstract description is now changed, re-generating the code must not overwrite the changes that happened in the meantime. This situation is known as *round-trip engineering* [19], see Sec. 1.3.2, and necessitates *bidirectional* [438] and *incremental* [194] model transformations. A conceptual description for such model transformations is given by Steven’s *consistency maintainers*, which were developed during her study of (the flaws of) the QVT standard [439]:

Definition 3.2 Consistency Maintainers [439]

Let M, N be two sets, called *model spaces*. *Consistency Maintainers* between M and N are given by a triple $(R, \vec{R}, \overleftarrow{R})$ comprising a *consistency relation* $R \subseteq M \times N$, a *forward restorer* function $\vec{R} : M \times N \rightarrow N$ and a *backward restorer* function $\overleftarrow{R} : M \times N \rightarrow M$.

The consistency maintainers are called *correct* if the following laws hold:

$$\begin{aligned} \forall m' \in M, n \in N : (m', \vec{R}(m', n)) \in R \\ \forall m \in M, n' \in N : (\overleftarrow{R}(m, n'), n') \in R \end{aligned}$$

They are *hippocratic* (“do no harm”) if the following holds:

$$\forall m \in M, n \in N : (m, n) \in R \implies \vec{R}(m, n) = n \wedge \overleftarrow{R}(m, n) = m$$

Finally, they are *history ignorant* if the following holds:

$$\begin{aligned} \forall m, m' \in M, n \in N : \vec{R}(m, \vec{R}(m', n)) = \vec{R}(m', n) \\ \forall m \in M, n, n' \in N : \overleftarrow{R}(\overleftarrow{R}(m, n'), n) = \overleftarrow{R}(m, n') \end{aligned}$$

Note the similarity between the conditions in Def. 3.2 and Def. 3.1. This is due to the fact that consistency maintainers can be interpreted as symmetric lenses [439].

Triple graph grammars (TGGs) introduced by Schürr in 1994 [422] are means for synchronising two structures that are presented as graphs, which are a widely recognised formalisation of model transformation and synchronisation [144, 194, 230]. Abstractly, this approach is considered to be an instance of a *symmetric delta-based lens* [135], which can be depicted by a *tile pattern*⁴ [121], see Fig. 3.6a (Dashed lines indicate elements

⁴Motivated by the concept of double categories and historically going back to [71]

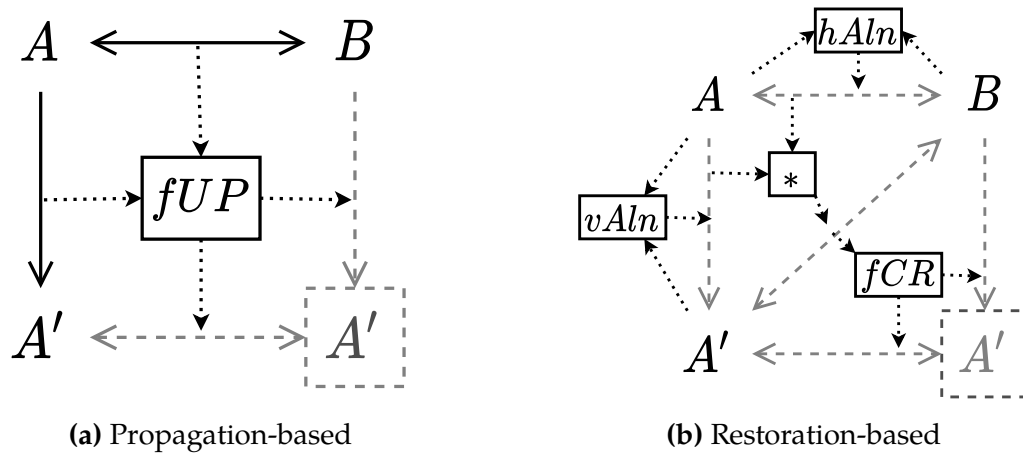


Fig. 3.6: Tile Notation, adapted from [18]

that are created during the execution of the operation). This figure shows a *forward update propagation* (*fUP*) operation that takes a *correspondence relation* (horizontal arrow) and an *update* (vertical arrow) as input to produce an update (the propagation) and a new correspondence relation to the resulting model. This approach is called *propagation-based*.

In their study of BX approaches, Anjorin et al. [18] identified how the propagation operation can alternatively be conceived as special case of model repair, see Fig. 3.6b. If the correspondence relation and update are not explicitly represented, they can be created via *horizontal* and *vertical alignment* (*hAln* & *vAln*) operations. Vertical and horizontal arrows can be combined via a *compose-operation* (***) resulting in a *diagonal* relation. The latter may be inconsistent, which is repaired by a *forward Consistency Restoration* (*fCR*) operation. This approach is called *restoration-based*.

In 2008, researchers from the aforementioned domains acknowledged their commonalities and started the cross-disciplinary BX research domain. The results of their first meeting are reported in [103]. This event was followed by a Dagstuhl seminar [241] in 2011 and a summer school in 2016 [1].

3.1.12 Coupled Evolution

Yet another special case of model repair, which is similar to BX is called *coupled transformation/evolution* [89, 307], short *co-evolution* [162, 224]. The conceptual difference to BX is that a modifications happens to an artefact on another meta-level and the co-evolution approach has to find a way to automatically adapt the artefacts on the meta-level below. Well-known examples of this scenario are *schema migration* in databases [402] or *metamodel-instance adaptation* in MDSE [89, 213, 334], i.e. a metamodel (modeling language) is changed and the existing models and dependent artefacts (model transformation definitions, editors) have to be migrated to the new version of the modeling language. An important concept in this context are *migration rules* [224, 373], i.e. updates are accompanied by a description how the dependent artefacts must be adjusted to restore consistency.

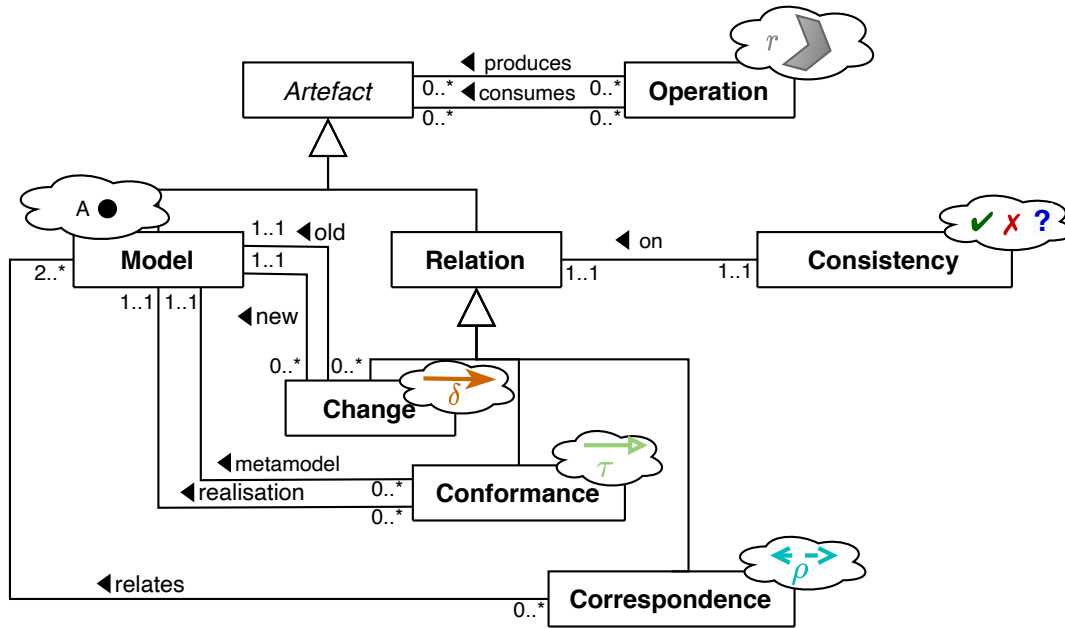


Fig. 3.7: Model-Management: Entities

3.2 Generic Model Management Framework

To put the various concepts introduced in Sec. 3.1 together, a comprehensive framework is needed. A suitable starting point is the *model management* framework presented by Diskin, Maibaum and others [119, 125, 128, 130], which is partly influenced by Favre’s megamodel [163–165]. I will adopt their framework here and apply some minor simplifications⁵. A personal contribution w.r.t. this framework is a *visual language*, which allows to depict model management operations intuitively.

3.2.1 Artefacts

Fig. 3.7 depicts an overview of the main entities of model management and their relationships. Due to its abstract nature, associations in this model are undirected the classes contain no further attributes nor methods.

Models are the essential building blocks of the framework. In the spirit of [76], almost every artefact related to design, development or operation of a software system is considered to be a model. There is no distinction between models and views as in Sec. 3.1.5. The notion of “a model being a view of another” is expressed through a *correspondence* relation. There are three different types of inter-model relationships: *change*, *conformance*, and *correspondence*.

Change is a binary relation between two models A and A' , which tells that model A (old version) has been modified, resulting in the model A' (new version), see the “evolution” traceability type (Sec. 3.1.3) and the vertical arrows in Fig. 3.6.

⁵E.g. The distinction between *viewOf*, *replicaOf*, and *refinementOf* is combined into the notion of *correspondence*

Conformance is a binary relation between two models A and M at different meta-levels, i.e. the second model M is an abstract representation of the language in which the former model A is expressed, see Sec. 3.1.9.

Correspondence is a relation among multiple models (A, B, C, \dots), which share a common part (overlap) and/or are related by other kinds of traceability relationships, see Def. 3.2 and the horizontal arrows in Fig. 3.6.

Consistency is a *property*, which is attached to an inter-model relation. This property express whether the relation is currently deemed to be *consistent* (\checkmark), *inconsistent* (\times) or *unknown* (?).

Operations process models and relations to create new models and relations or update existing ones. The most common model management operations are discussed further below.

The small cloud symbols in Fig. 3.7 introduce a concrete syntax, which can be used to depict model management scenarios. Models are denoted by black dots and their names are capital latin letters. The three relations are denoted by arrows, each highlighted by a different colour and arrow different arrow tip. In the spirit of Favre [167], they are associated with a Greek letter: δ stands for changes, τ stands for conformance, and ρ stands for correspondence.

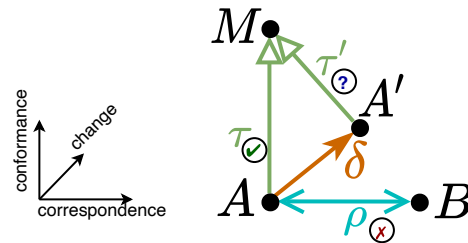


Fig. 3.8: 3D Model Management Space

The three relations can simultaneously be associated with a *dimension*, of 3D space, which is sketched in Fig. 3.8. In related works [15, 162], these dimensions have been referred to as *time* (change), *space/variability* (correspondence) and *language* (conformance). Consistency is, if relevant, visualised by small check-,cross- or question marks. Operations are depicted by filled double arrows or chevrons.

Compared to Favre’s megamodel [164–167], only the conformance-relationship has been inherited. The *representationOf*-relationship is not considered since all members of the model management frameworks are abstractions. Similarly, *elementOf* is not considered and will be expressed via *conformance*. The *decomposedIn*-relationship can be expressed via *correspondence* but the latter is able to represent many more inter-model relationships that are not directly expressible in Favre’s megamodel. Also, the *isTransformedIn*-relationship does not exactly coincide with the *change*-relation since the latter is not necessarily bound to model transformation and can also express manual modifications through the user.

3.2.2 Operations

A model management tool has to offer means for representing and storing models (editor functionality) and it must offer *operations* to work with these artefacts [45]. The execution of model management operation can be fully automatic, completely manual or something in between. To discuss the most central operations, I will utilise

Conceptualisation

the concrete syntax described above. Each operation, called *design patterns* in [125], comprises various types of inter-model relations. I distinguish elementary operations that only involve one relation at a time and complex operations that involve multiple relations simultaneously.

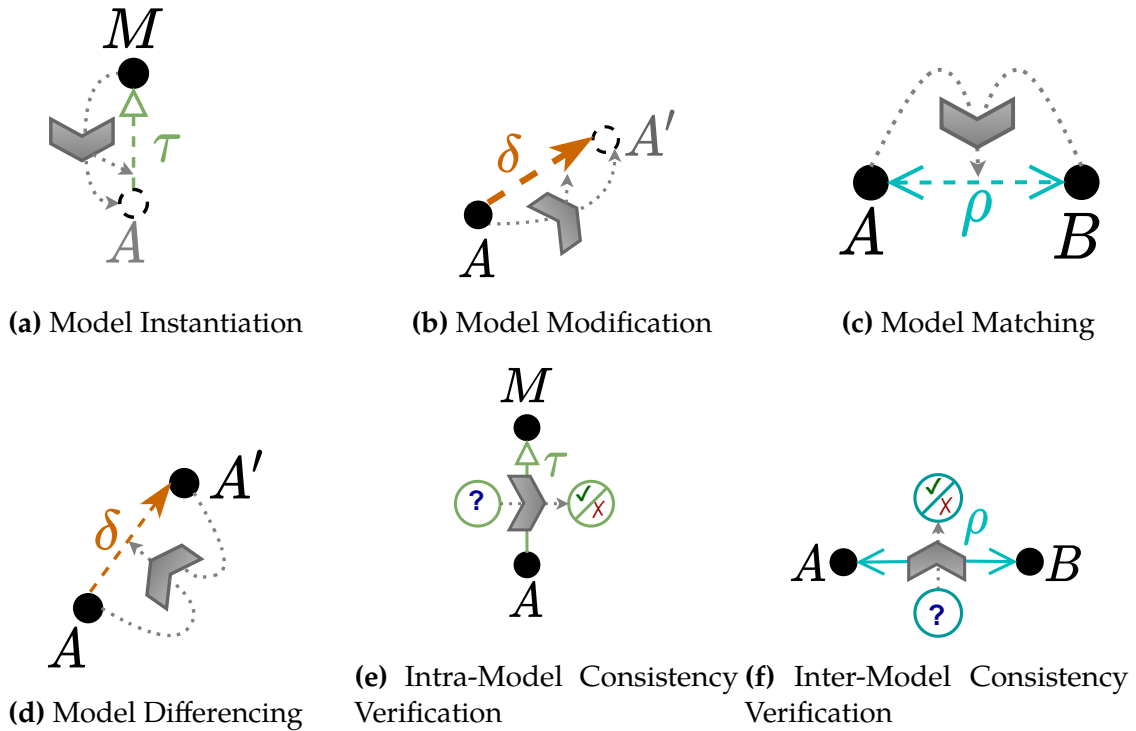


Fig. 3.9: Elementary Model Management Operations

ELEMENTARY OPERATIONS

Model Instantiation (Fig. 3.9a) describes the creation of a new model. The models considered in this thesis have a specific format, i.e. they are denoted in some modeling language. Thus, the newly created model A implicitly comes with a conformance relation τ towards the metamodel M , which represents this language.

Model Modification (Fig. 3.9b) allows updating an existing model A , which will be represented by a change δ . This operation can be performed fully manually or fully automatically, see “endogenous in-place model transformation” in Sec. 3.1.7.

Model Matching (Fig. 3.9c) is the central activity in model and data integration. It takes two or more models as input and produces a *correspondence* relation between them. This means finding overlaps and other traceability relations among the input models. This operation, called horizontal alignment in Fig. 3.6b, is also contained in the catalogues in [45] and [70].

Model Differencing (Fig. 3.9d) is closely related to *model matching*. Indeed, implementation strategies for both operations turn out to be similar [290]: *Model differencing* can be implemented via *model matching* by first identifying common

elements and then considering the remaining unmatched elements to be the update. The conceptual difference is that this operation produces a *change* and not a *correspondence*.

Intr-Model Consistency Verification (Fig. 3.9e) describes the activity of checking the consistency of a single model (in-viewpoint check in Sec. 3.1.1). It is also known as *vertical* consistency verification. This operation checks the model against structural rules imposed by the metamodel. It is common, to augment the syntactical rules with user-defined *semantic rules*, which assert a custom system behaviour [155].

Inter-Model Consistency Verification (Fig. 3.9f) describes the activity of checking the consistency among multiple models (inter-viewpoint check in Sec. 3.1.1). It is also known as *horizontal* consistency verification [155]. It is a more complex variant of *internal verification* since it additionally has to consider *correspondence* relations between models.

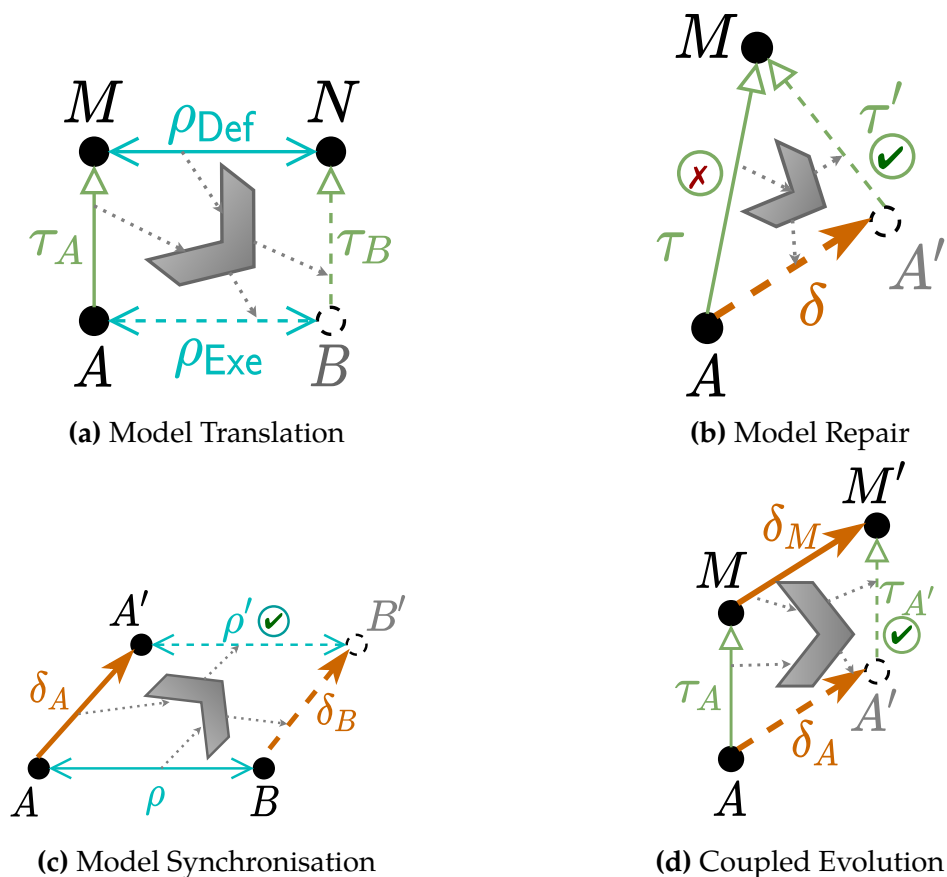


Fig. 3.10: Complex Model Management Operations

COMPLEX OPERATIONS

Model Translation (Fig. 3.10a) is better known under the name (out-place) model transformation. I intentionally avoid calling this operation model transformation to exclude in-place model transformations. In this framework, the latter are

Conceptualisation

considered as a special case of *model modification*. Model translation takes an instance of a source metamodel (represented by τ_A) and produces a corresponding instance of a target metamodel (represented by τ_B). The translation (represented by ρ_{Def}) is *defined* between the respective metamodels M_A and M_B and produces a *trace* (represented by ρ_{Exe}). Note the similarity between Fig. 3.10a and Fig. 3.5b. One of the most common examples of this operation is *code generation*. Also, the *model slicing* operator mentioned in [70] can be considered as special case of this operation, where the translation means projection

Model Repair (Fig. 3.10b) is another name for the restoration of consistency in the aftermath of a detected inconsistency, see Sec. 3.1.10. An inconsistent model (represented by τ) is repaired by a “fix” (represented by δ) resulting in a consistent model (represented by τ').

Model Synchronisation (Fig. 3.10c) is the topic of BX (Sec. 3.1.11), i.e. keeping two related models (represented by ρ_0) consistent with each other when they change. This means that in the event of a modification in one of the models (represented by δ_A), consistency is restored by propagating an update (represented by δ_B) to the other model. Compare the visualisation in Fig. 3.6a to Fig. 3.6a.

Coupled Evolution (Fig. 3.10d) describes the problem of adapting a model A when its language changes, i.e. its metamodel M is updated (represented by δ_M). The result of this operation is a “migration” (δ_A) such that the resulting model A' conforms (represented by $\tau_{A'}$) to the new language version, see Sec. 3.1.12.

3.3 Multi-Model Consistency Management Framework

The generic model management framework in Sec. 3.2 covers the whole breadth of topics in MDSE and beyond, which is too broad for the scope of this thesis. The focus of this thesis is multi-model consistency management. Thus, the scope has to be narrowed down to this particular domain and a more concrete conceptual framework has to be created, which will also mark the main contribution in this chapter. I begin by stating five *principles*, which characterise multi-model consistency management. Based on these principles, a *domain model* (structure) and a *process model* (behaviour) is developed, see Sec. 3.3.6.

3.3.1 Model Spaces

First, I adopt the notion of *model spaces* from BX:

Definition 3.3 Model Space

A *model space* is a homogeneous collection of *models* together with the network of possible *changes* among them. A change between two models inside a model space is also called an *update*.

Remark 3.1 *State-Based Model Spaces*

Some researchers conceive model spaces in a *state-based* manner, i.e. without updates [133]. Def. 3.3 can account for this notion by considering updates to be given by the cartesian product of all member models with itself, i.e. every model pair is a possible update.

The attribute *homogenous* implies that all members of the model space conform to the *same* metamodel. Fig. 3.11 provides an intuitive depiction of a model space. Often, a metamodel becomes tantamount to a model space because it gives rise to a default model space (= all instance models). The model space generated by a metamodel M will be referred to as $\text{Mod}(M)$.

The network of *updates* inside a model space may exhibit the notion of *update composition* and *idle updates*. A composed update, i.e. a sequence of atomic updates, can be interpreted as a *path* in the update network. An idle update, i.e. not changing anything, can be interpreted by remaining at the same model (state).

Furthermore, the notion of model spaces induces the *type-instance pattern*. Thus, I will be working with two meta-levels, see Fig. 3.3. The lower level contains all members of models spaces. The upper level contains the metamodels behind these model spaces. This also applies for the example scenarios in Sec. 1.3. For instance, the model spaces in the system integration scenario in Sec. 1.3.1 are given by the possible database states of the various information systems and the metamodels are their respective schemas. In the system design scenario in Sec. 1.3.2, the model spaces are given by the possible revisions of the design documents and the metamodels are the languages used for denoting these documents. Multi-level modeling, which was shortly mentioned in Sec. 3.1.6, will remain outside the scope of this thesis. Thus, there are exactly two linguistic meta-levels, which, throughout this thesis, will be referred to as *type* or *metamodel* level and *instance* level.

The concrete nature of model spaces is investigated by **RQ1**.

3.3.2 Commonalities

The first principle stated how *models, change, conformance* are organised within multi-model consistency management. The second principle concerns the remaining *correspondence* relation, which induces multi-models. A multi-model is defined as collection of inter-related models, see Def. 1.5. In the generic model management framework, a multi-model is represented as tuple of the correspondence relation. However, it turns out that a multi-model is more than just a tuple comprised of related models stemming from disparate model spaces, i.e. there is additional data attached to a multi-model. This data is represented by overlaps and other traceability links. In the literature, there many names for this phenomenon, e.g. traces [77], correspondences [439], corrs [422], commonalities [279], morphisms [340], mappings [45], cross-reference links [110]. In

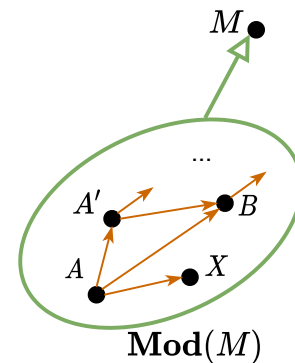


Fig. 3.11: Model Space

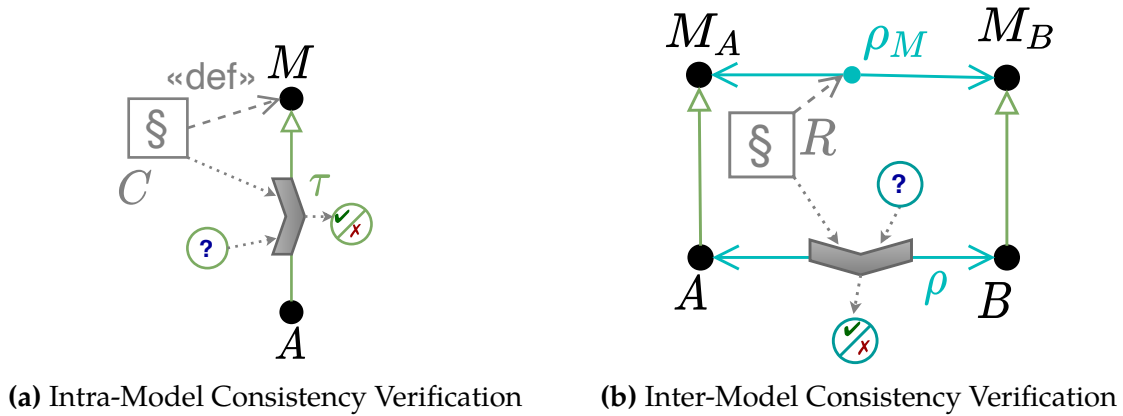


Fig. 3.12: Consistency Rules

this thesis, I will choose the term *commonality*, since *correspondence* is already in use.

Definition 3.4 Commonality

A structural relationship among *elements* spanning disparate models is called a *commonality*

The concrete interpretation of a commonality and concrete representation is kept abstract at this point and will be investigated in later chapters. Note that the commonalities within a multi-model may not be known from the start. In this case, *model matching* (Fig. 3.9c) has to be applied. All aspects related to the discovery and representation of commonalities are investigated by **RQ1**.

3.3.3 Consistency Rules

Sec. 3.2.2 describes two types of consistency verification: intra-model and inter-model verification. This third principle states that both operations are executed w.r.t. to *consistency rules*.

A consistency rule provides the rationale for deeming the relation to be consistent or inconsistent. Thus, the reasoning process becomes reproduceable and therefore also automateable. Without explicit consistency rules, internal and external verification would be completely manual processes.

Definition 3.5 Consistency Rule

A *consistency rule* is a syntactic definition that is defined over a metamodel or corresponding metamodels (inter-model consistency rule), which is used by a *verification* operation to decide whether a model or multi-model is consistent.

Hence, the presentation in Fig. 3.9e and Fig. 3.9f has to be updated. Fig. 3.12 shows the respective visualisation, which accounts for consistency rules. Note, the bullet in the middle of the correspondence ρ_M , which highlights the *commonality* data, see Sec. 3.3.2.

One can distinguish two phases of a rule: *definition* and *execution*. The definition happens at design-time and is based on metamodels, i.e. the rule definition refers to

element types. Take as an example the following OCL-invariant, which requires that the name of every `Class` in a UML class diagram must start with a capital letter. This definition mentions a type (`Class`) and a feature (`name`), which are both elements of the UML-metamodel.

```
context Class inv: self.name->substring(0,1) = self.name->substring(1,0)->toUpper()
```

In case of consistency rules, which are used for inter-model verification, a rule refers to elements from multiple metamodels including their commonalities.

The execution happens at run-time and is based on models (instances). In practice, this can be implemented by a *check* function telling whether the model is valid or not. To allow for further analysis, this function, instead of just saying `true` or `false`, may mention all the elements violating the rule. I call the resulting artefact an *inconsistency report*, which is empty when there are no inconsistencies. Hence, execution of a consistency rule can be interpreted as a special kind of model transformation.

The classification in [155] distinguishes between syntactical and behavioural rules. The former safeguard well-definedness w.r.t. the language definition, e.g. that the inheritance relation in class diagrams must be free of cycles and so on. This type of rule also marks the most commonly investigated case, see [460]. Semantic rules enforce a specific system behaviour that should be embodied in the models. These rules are domain-dependent and therefore commonly specified by a user while syntactical rules are usually given a priori. At this point it is important to clarify that the main focus of this thesis are *structural* models. Thus, behavioural semantics associated with models such as *state machines* or *activity diagrams* are outside of the scope of this thesis. When talking about semantic consistency rules, I refer to user-defineable rules that make assertions about static properties of models.

All aspects of consistency rules and their verification are investigated by [RQ2](#).

3.3.4 Model Repair

The fourth principle asserts the existence of *primitives* for performing *model repair*. The concrete implementation is kept abstract, i.e. it can utilise searching, repair actions, user interaction, learning and more. Also, means for model synchronisation, e.g. lenses and consistency maintainers, are part of this category, see the discussion in Sec. 3.1.11. The signature of a repair primitive takes as input an inconsistency report including references to all relevant consistency rules and (multi-)model to produce one or more updates (the fixes or propagations).

The means for model repair are investigated by [RQ3](#).

3.3.5 Architectures

The fifth and final principle concerns the architecture of multi-model consistency management solutions. This is especially relevant for situations where one is dealing with more than just two model spaces, like the examples in Sec. 1.3.

I claim that there are, in principal, two approaches for orchestrating the existing means for consistency verification and restoration. They arise from the two main approaches to network design: *client-server* and *peer-to-peer*. In the context of multi-



	Global View	Maintainer Network
Visualisation		
Advantages	+ Control + Scalability + Multi-ary correspondences	+ Little Overhead + Resilience + Privacy
Disadvantages	- Big Overhead - Single Point of Failure - Privacy	- Control - Scalability - Multi-ary correspondences

Table 3.1: Global Views vs. Consistency Networks

model consistency management, I call them the *global view* approach and the *maintainer network* approach.

In the former approach, there is an auxiliary artefact: the *global view*. It provides an integrated presentation of all models, correspondence relations and consistency rules. Verification and restoration are performed based on the global view. This implies a centralised and restoration-based approach, see Fig. 3.6b, and there is no need for special propagation primitives such as lenses. An example is the concept of a SUM in multi-view modeling, see Sec. 3.1.5.

The *maintainer network* approach realises verification and restoration in a decentralised way. Every correspondence relation requires a respective inter-model verification procedure and synchronisation primitive. This is usually facilitated in pairwise manner. An example of this approach is described in [442].

Tab. 3.1 gives a summary of the advantages and disadvantages of each approach. The global view approach offers the benefit of a central point of *control*, which allows to resolve conflicts and to detect repair actions that would cause cyclic invocations, see [197, 442]. Also, it is easy to *scale* up to more component models since adding a new component simply means incorporating one more model in the global view (grows linearly). This is different in case of the maintainer network, whose complexity grows quadratically. On the other hand, the maintainer network approach avoids the drawbacks of the global view approach, i.e. *privacy* issues (the global view exposes data from all components), *overhead* (the global view artefact may become unwieldy), and single-point-of-failure (when the global view cannot be constructed).

The main advantage of the global view approach is its ability to deal with general *multi-ary* correspondence relations in a straightforward way. In general, not every n -ary relation ($n > 2$) can be factorised into pairs of binary relations. This has been investigated by the database community, see e.g. [458]. For an illustrative example⁶ in the MDSE domain, consider Fig. 3.13: The figure comprises three UML class diagram models A_1 , A_2 and A_3 , which model the same system and therefore Classes with identical names are considered to be same. The models are consistent by themselves and also when considering them pairwise they appear consistent. Only when considering all three models at the same time, it is possible to spot that there is an

⁶Credit goes to Zinovy Diskin for the idea behind this example.

3.3 Multi-Model Consistency Management Framework

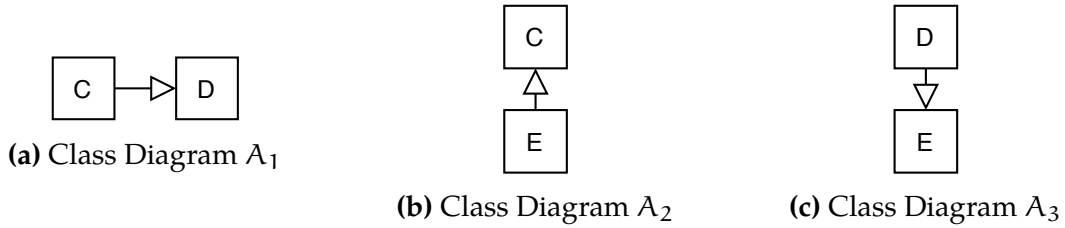


Fig. 3.13: Inconsistent Class Diagrams

acyclic inheritance hierarchy! Therefore, binary means synchronisation are generally not sufficient. Adding n-ary synchronisation means in a maintainer network may increase the complexity significantly and eventually result in a setting, which resembles the global view approach again, see the discussion in [91].

This principle is associated with RQ4.

3.3.6 Summary: Conceptual Model

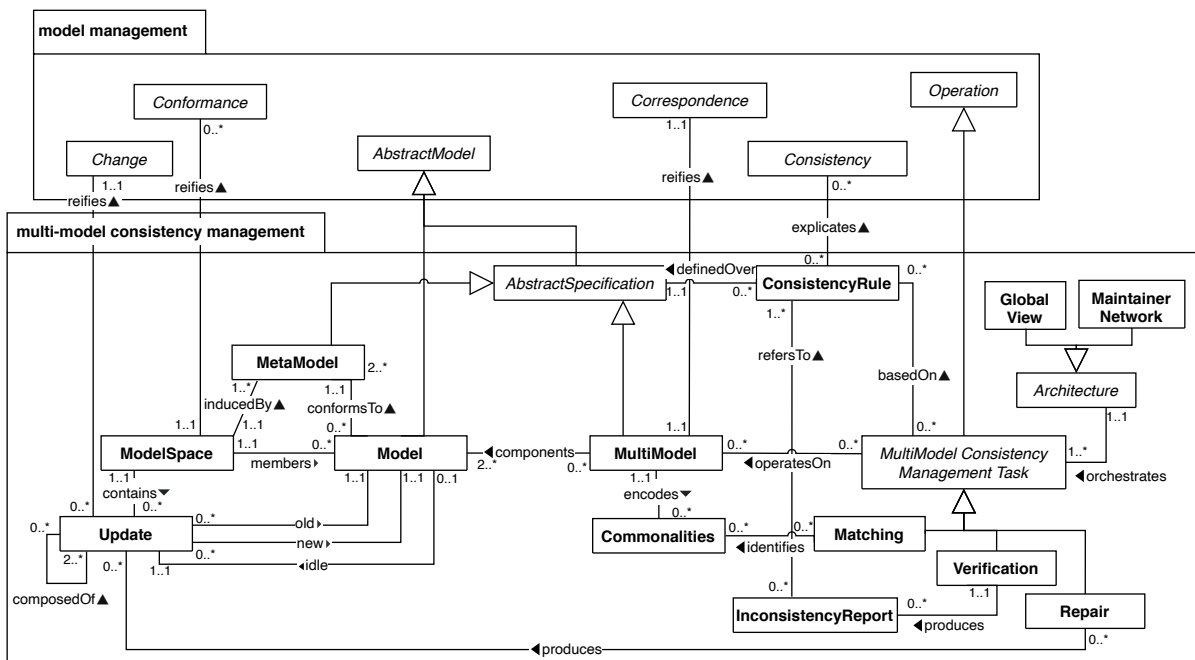


Fig. 3.14: Multi-Model Consistency Management: Data

Fig. 3.14 presents the domain model emerging from the aforementioned principles. This model is related to the model in Fig. 3.7 via reification or specialization (The term *model* in Fig. 3.7 has been renamed to *AbstractModel* here). A *multi-model* comprises (at least two) *components* and has associated *commonalities* data. Components are *models*, which are contained in a *model space*. The latter is induced by a *metamodel* and comprises a network of updates. Model spaces are induced by a metamodel.

Multi-model consistency management comprises three activities: *matching*, *verification* and *repair*, which are orchestrated in an *architecture* and based on *consistency rules*. The latter are defined over *metamodels* or a multi-model that reifies a correspondence among metamodels. The coordination of these activities is depicted in BPMN model in

Conceptualisation

Fig. 3.15. The process can be triggered in regular intervals, in an event-based manner (e.g. when an observed model gets modified), or on a manual invocation. Depending on whether *commonalities* are up to date or not, *model matching* (Fig. 3.9c) has to be performed. Afterwards, *verification* is performed, which produces an *inconsistency report*. The process may end here with reporting the discovered inconsistency or, depending on project policies, a *model repair* phase may follow to resolve the discovered inconsistencies by producing updates (= fixes/propagations). The data objects in Fig. 3.15 depict the relationship between operations and artefacts, and the non-standard visualisation of the activity types in Fig. 3.14 shall indicate the various possible strategies for implementing the respective operation. In the following chapter, I will analyse how artefacts (model spaces, multi-models, commonalities, and consistency rules) and operations (matching, verification, and repair) are actually implemented.

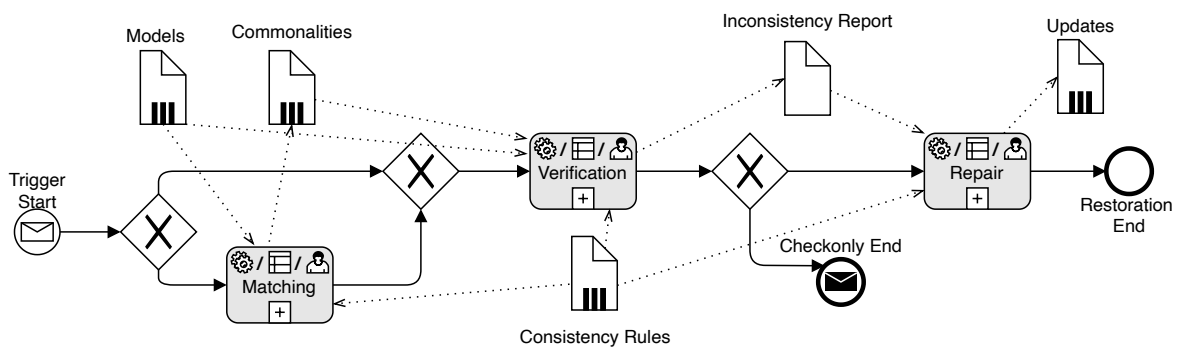


Fig. 3.15: Multi-Model Consistency Management: Process

“Those who do not remember the past are condemned to repeat it.”

—George Santayana

CHAPTER 4

STATE OF THE ART

In this chapter, I will investigate the body of existing knowledge and identify how the concepts of multi-model consistency management identified in the previous chapter have been realised in manifold scientific approaches. The main contribution of this chapter is the introduction of a *feature model* [265], which allows to categorise and compare the various existing approaches. An earlier version of this feature model has been published in [451]. The updated version presented in this thesis includes some minor changes and extensions, a detailed list of differences is found in Appendix A.

This chapter begins with an explanation of the *method* in Sec. 4.1 explaining how the literature study was conducted. Sec. 4.2 contains the main contribution of this chapter, a feature model for multi-model consistency management. Sec. 4.3 presents a list with general observations that were made during the literature study leading over into Sec. 4.4, which presents three concrete tools that support multi-model consistency management. Finally, Sec. 4.5 concludes this chapter with a summary of current limitations with existing tools and approaches.

4.1 Method

The “gold standard” for literature studies in software engineering are *systematic literature reviews* or *systematic mapping studies* [139, 277], short SLRs and SMSs. Both define a rigorous process [276, 379] for querying literature databases with selected keywords and analysing the results w.r.t. a set of initially stated research questions. These types of studies are called *secondary* studies since they aggregate the results of *primary* studies. In general, systematic literature reviews and mapping studies require a lot of effort and ideally involve a big group of researchers from different institutions to assure an unbiased selection and analysis of primary studies. Luckily, there already are several literature reviews and survey papers in the domains related to multi-model consistency management see Sec. 3.1. Therefore, me and my co-authors chose a slightly different approach in [451] and used these literature reviews and survey papers as the basis for our investigations. Hence, the work in [451] can be considered to be a *tertiary* study since it builds upon secondary studies. The secondary studies that represent the starting point of our investigation were identified by the following search string that was fed into the *Google Scholar*¹ and *DBLP*²:

¹<https://scholar.google.com.tw/>

²<https://dblp.org/>

Secondary Study	Citation	Domain	# of Prim. Studies
Multi-view Consistency in UML: A Survey	[284]	Consistency Management	86
Multi-view approaches for software and system modelling: a systematic literature review	[87]	Multi-View Modeling	40
A Feature-based Classification of Model Repair Approaches	[331]	Model Repair	54
Feature-based classification of bidirectional transformation approaches	[234]	bidirectional transformations	13
Approaches to Co-Evolution of Metamodels and Models: A Survey	[224]	Coupled Evolution	56
Model Management Tools for Models of Different Domains: A Systematic Literature Review	[461]	Model Management	56
Survey of Traceability Approaches in Model-Driven Engineering	[190]	Traceability Management	12
Systematic Review of Software Behavioral Model Consistency Checking	[350]	Consistency Management	96
Model-Driven Engineering as a new landscape for traceability management: A systematic literature review	[416]	Traceability Management	29
Total	9		+ 474 = 483

Table 4.1: Data base of multi-model consistency management-related studies

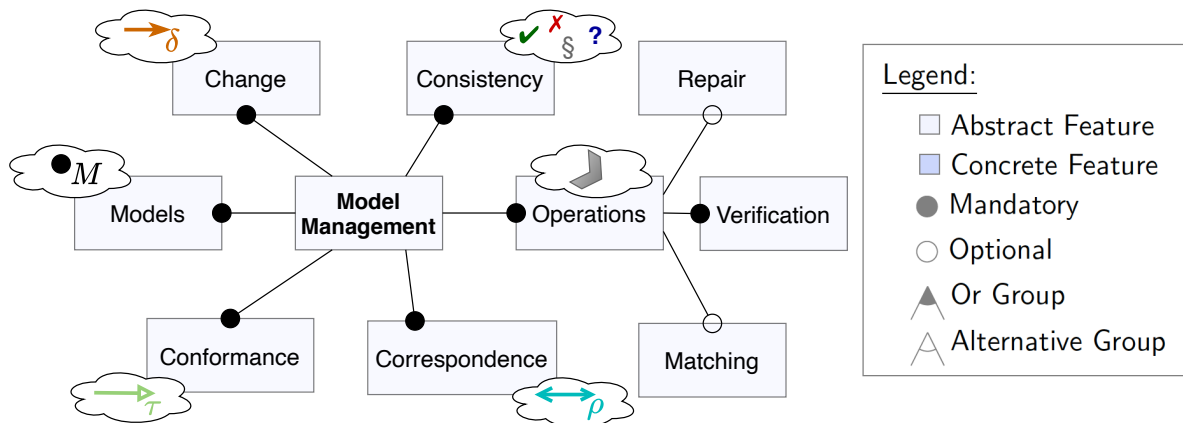


Fig. 4.1: Feature Model Overview

(“model repair” OR “consistency verification” OR “(in)consistency restoration” OR
 “(in)consistency management” OR “bidirectional transformation” OR “model
 synchronisation” OR “multi-view” OR “model management” OR “megamodel” OR
 “model traceability” OR “traceability management”)
 AND
 (“survey” OR “literature review” OR “feature-based” OR “overview” OR
 “taxonomy”)

This search was based on occurrences of these terms in the title only. The results returned by *Google Scholar* and *DBLP* had to be filtered manually w.r.t. to the criterion that a study must be categorised as a work within software engineering or Computer Science, thus explicitly excluding studies within *machine learning* (sharing the terms “multi-view” and “model”), *CAD* and *3D-modeling* (sharing the terms “consistency” and “model”), *UI development* (sharing the term “user model”), and *decision support systems* (sharing the term “model management”). The final result of identified secondary studies is shown in Tab. 4.1. Earlier survey studies [6, 37, 153, 245, 320, 433, 460, 469, 489] in the respective areas are subsumed by the papers listed in 4.1 and therefore not contained in this table.

I continued my literature study by reading each of those nine papers in detail. Afterwards, I skimmed and categorised the contained primary studies and did a “snowballing”-search w.r.t. the papers mentioned in the related work sections. This way, I identified some more related survey and classification papers ([18, 68, 123, 157, 438]) that constitute the data base for my literature study.

4.2 Feature Model

Feature models [265] have become a kind of a de-facto standard for presenting the result of literature studies, see e.g. [68, 87, 331]. It was pioneered for this purpose by Czarnecki and Helsen [104] in their survey of model transformation approaches. A feature model provides a classification scheme in terms of hierarchically organised

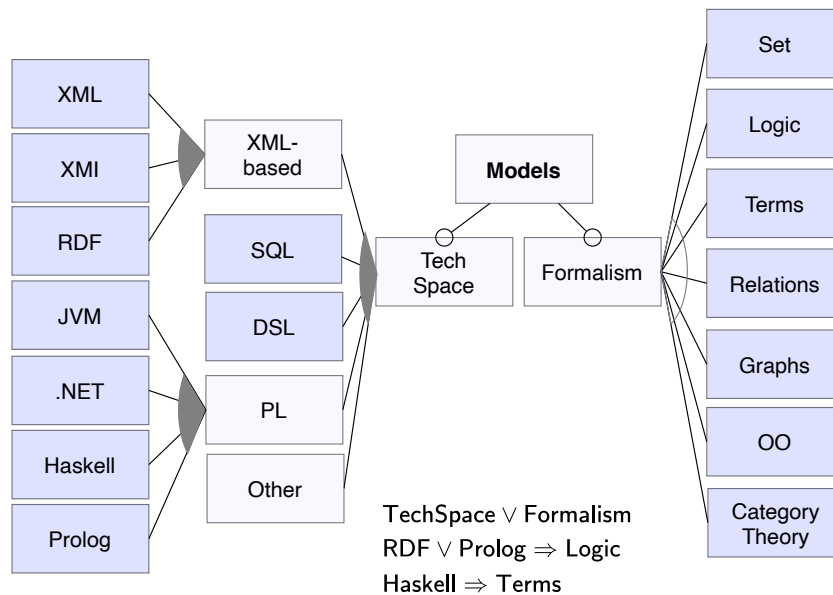


Fig. 4.2: Model Feature

features. The root layer of my feature model is shown in Fig. 4.1. The “clouds” allude to the concrete syntax introduced in Chap. 3.2, see Fig. 3.7.

The legend in Fig. 4.1 explains the visual syntax of feature models. A feature can be *abstract* (light-blue background colour), which means that it is composed of other (abstract or concrete) features. A *concrete* feature (dark-blue background colour) represents a boolean variable, i.e. whether a study considers this feature or not. The hierarchical relationship between two features indicates whether certain sub-features are always present (*mandatory*) or only sometimes (*optional*). An abstract feature can also represent a group of concrete features. Their enumeration is either inclusive (*or-group*) or exclusive (*xor-group*).

To highlight features in the text, a sans-serif font is used for referring to them directly. The figures in this chapter highlight show specific parts of the overarching feature model and some of them contain expressions in propositional logic. These expressions denote constellations of features that have been discovered to always appear together. When referring to features across different figures, it is sometimes necessary to fully qualify a feature because their names are not always globally unique. The parent-child relationship between features is denoted as “::”, e.g. “Top Feature::Sub Feature”.

This literature study is concerned with approaches to multi-model consistency management, which represents a special setting for model management. Thus, the root feature is called Model Management. Below are the top-level features (“dimensions”) Models, Change, Conformance, Correspondence, Consistency, as well as the three Operations Matching, Verification, and Repair. The sub-tree of each of those features is presented in the following sections.

4.2.1 Models

Models are the elementary artefacts in multi-model consistency management. This feature distinguishes between Tech Space [78] (i.e. how models are concretely implemented) and Formalism (i.e. how models are conceived in theory), see Fig. 4.2. An

approach to multi-model consistency management must at least address one of these features (see the formula “TechSpace \vee Formalism” in Fig. 4.2).

TECH SPACE Model management in the database domain is based on SQL [31, 402]. Approaches are generally implemented in a specific programming language (PL). The most common choices are Java Virtual Machine (JVM) languages [73, 313], the .NET framework [235], Haskell [286] or Prolog [384]. A majority of approaches works with models, whose representation is XML-based. XML Metadata Interchange (XMI) [366] is an XML-dialect used for storing and exchanging UML-models [365]. It is also used for model serialisation within the Eclipse Modeling Framework (EMF) [436], which is the technical foundation of numerous MDSE tools. RDF is an XML-dialect for knowledge representation mainly used in the context of semantic web [435]. Some approaches work with general XML documents [357]. Alternatively, models are represented in a textual DSL [283]. There are also mentions of other technologies, e.g. [233], but they constitute only a small share and are therefore grouped as Other.

FORMALISM Arguably, the most simple modeling formalism is to consider a model as a Set of its elements [439]. Research in the database domain [31] naturally utilises Relations [92] as a formal underpinning. A formally equivalent approach is to consider a model as a set of sentences in a Logic. This is the standard Formalism for approaches based on solvers or theorem provers [88, 328]. Note that there are various kinds of “logics”, e.g. *description logic* [470] or *relational logic* [249]. In functional programming approaches [286], models are essentially Terms, i.e. “generalised trees”. Strictly more general are Graphs [233], which allow for cyclic relationships between elements and are considered a standard means for depicting structured information in Computer Science. The Object Oriented (OO) formalism [288, 395] combines aspects of both graph and relational formalisms and adds additional features such as inheritance and constraints. The most abstract formalism [59, 235] represents models via concepts borrowed from Category Theory [34], which generalise all of the above, see e.g. [132, 408]. Some of the technologies mentioned above immediately a specific Formalism, e.g. RDF and Prolog imply a respective formalisation through a Logic-based representation.

4.2.2 Change

Change is one of the three relationships types between models in the model management framework from Sec. 3.2. This abstract feature is further analysed w.r.t. the three aspects Representation, Allowed Updates and Meta Information, see Fig. 4.3.

REPRESENTATION There are two ways for representing change: State-based [31, 178] or Delta-based [44, 133]. The latter is further classified into Structural- and Operational-Deltas. In State-based representation there are only (old and new) model states. Structural Deltas [133, 230] represent change as an entity of its own right, which is concretely represented by links that witness which elements are added, deleted, or remain unchanged. Operational Deltas [239] represent actual method invocations.

Representation happens through Recording. This can happen Offline or Online [224]. The latter is either implemented in an Intrusive [287], i.e. the user has to use a special

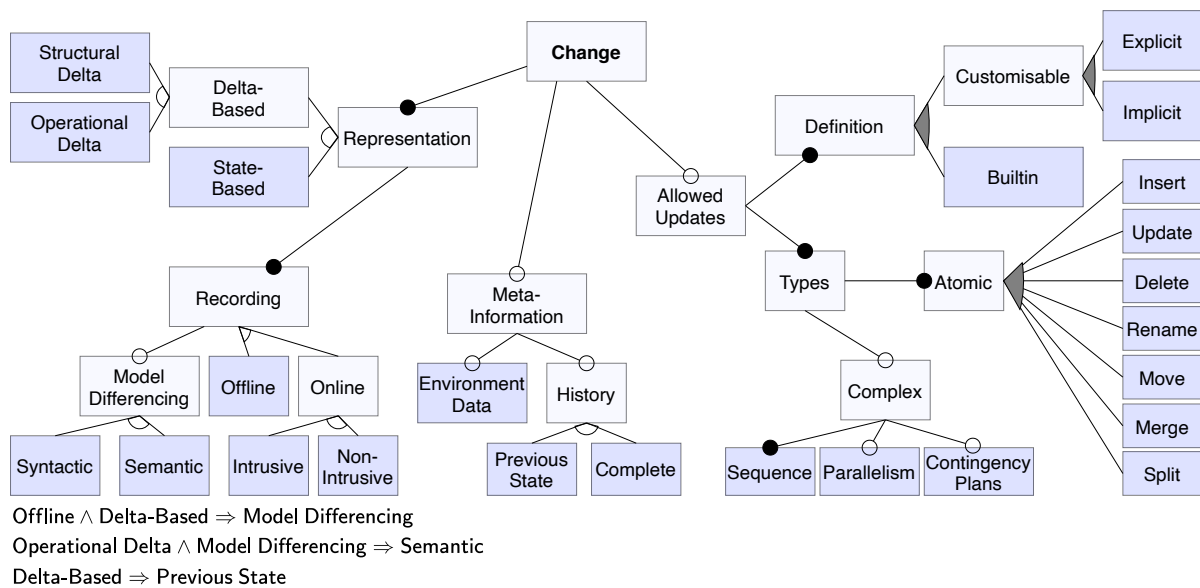


Fig. 4.3: Change Feature

interface, or Non-Intrusive [112] manner, i.e. the user remains unaware of the recording. Offline recording in combination with a Delta-based representation necessitates Model Differencing [10, 290, 399], which is further distinguished between Syntactic [67] and Semantic [267] model differencing. The latter is required to Operational-Deltas [268].

ALLOWED UPDATES Another aspect that can be analysed concerning the Change-relation is the set of Allowed Updates for a certain model. This Definition may be Builtin [90] for a concrete approach or Customisable. The latter can happen in an Explicit way (i.e. the user defines a set of available operations [328]) or in an Implicit way (i.e. the allowed updates are derived from another description, e.g. a metamodel [271]).

Furthermore, one can distinguish the Types of allowed updates. I further distinguish between Atomic and Complex changes. Common examples of atomic changes are Insert, Update and Delete operations. It can be worthwhile to consider Rename separately and also consider more elaborate atomic changes such as Move, Split and Merge, see [224]. Most Complex changes are defined in terms of a Sequence of atomic changes. However, there are approaches that also consider changes happening in Parallel [501] or even Contingency Plans [385], which account for different possible outcomes.

META-INFORMATION A common type of Meta-Information is a History over the changes happening to a model. This history may only contain the Previous State [470, 500] or even the Complete [141, 385] history. Another type is Environment Data [292] such as user credentials, the current, or a reference to external documents.

4.2.3 Conformance

The majority of software models is denoted in a modeling language, which is represented by a special model called metamodel. The relationship between a model and its metamodel is called Conformance, which expresses that a model is well-formed. Hence, there are two aspects to this feature: Well-Formedness and Metamodels, see Fig. 4.4.

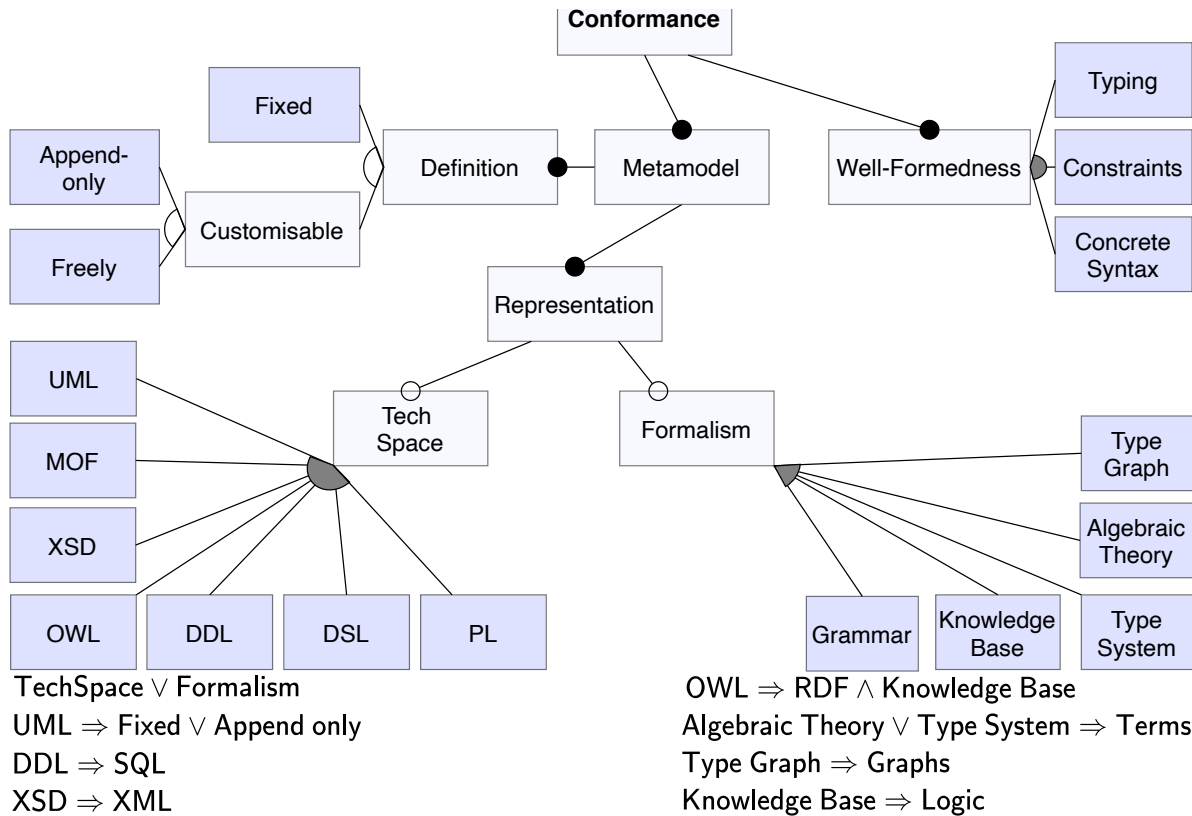


Fig. 4.4: Conformance Feature

WELL-FORMEDNESS Well-Formedness is generally expressed via rules [373], called Constraints. In addition to this, there is the notion of Typing, i.e. every element is correctly assigned to an abstract syntax element (language concept) [87]. Furthermore, a metamodel may be equipped with a Concrete Syntax such that it can be verified whether the concrete presentation of the model adheres to this definition [401].

METAMODEL Under the Metamodel aspect, I distinguish between Definition and Representation. An approach may work with a Fixed set of metamodels [142] or allow it may open for a Customisable [372] set of metamodels. The latter may be limited to configuring the set of metamodels in an Append-only [395] fashion or it may allow configuring the set of metamodels Freely [371]. The Representation aspect is further analysed w.r.t. TechSpace and Formalism, as in the Models feature, see Sec. 4.2.1. The UML metamodel is fixed [365] but it allows extensions via *profiles* and OCL annotations [479]. MOF [368] is the OMG standard for defining metamodels. A prevalent variant of this language is *Ecore*, which is the metamodeling language in EMF [436]. In the context of XML, XSD is the common language for the definition of metamodels [340]. In the context of the semantic web (i.e. RDF models), OWL definitions can be interpreted as metamodels [435]. In the context of databases, metamodels are expressed via expressions written in the data definition language (DDL) [31]. Apart from that, there are various DSLs for defining metamodels, e.g. [47, 372, 482]. Moreover, the definition of data types in a programming language (PL), e.g. Java classes with annotations, can be interpreted as a metamodel [225]. On the formal side, metamodels are interpreted via Grammars [422], Knowledge Bases [385], Algebraic Theories [57], Type Systems [286] or Type Graphs

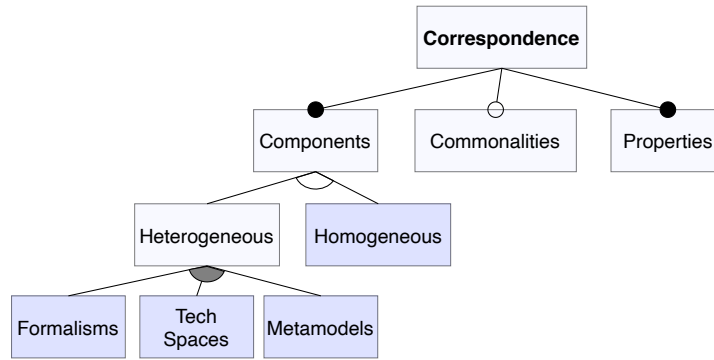


Fig. 4.5: Correspondence Feature - Overview

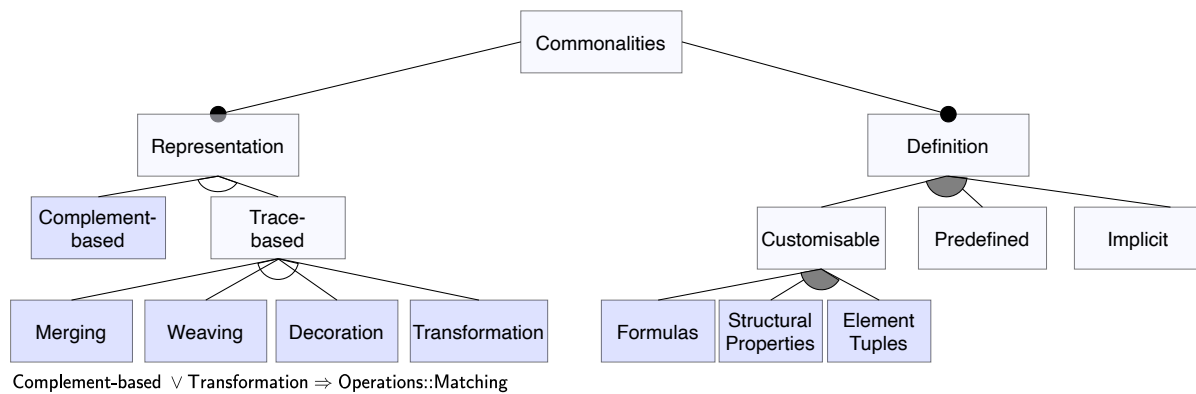


Fig. 4.6: Correspondence::Commonalities Feature

[48]. Some technologies are directly associated with a specific formalism (e.g. RDF/OWL \Rightarrow Logic) and some formalisms only work together with a specific formal model representation, see Fig. 4.4.

4.2.4 Correspondence

Multi-models (Def. 1.5) are based on the notion of Correspondence, see Sec. 3.2 Hence, this feature takes a prominent role and I distinguish three aspects of it: Components, Commonalities and Properties, see Fig. 4.5. The detailed structure of the sub-features Commonalities and Properties is shown in Fig. 4.6 and Fig. 4.8, respectively.

COMPONENTS The models that are related by a correspondence are called Components (of a multi-model). Early model management approaches required a Homogeneous setting [411], i.e. all models share the same metamodel, formalism, and technical representation. In general, multi-model consistency management shall allow for Heterogeneous settings, i.e. Components may differ with regard to their underlying Metamodels [134], Tech Spaces [78], and Formalisms [347].

COMMONALITIES (DEF. 3.4) are what distinguishes a multi-model from being a mere tuple of Components. I distinguish two aspects of this feature: Definition (design-time) and Representation (run-time). In the multi-model consistency management process, Definition comes first. It can be Implicit, explicitly Predefined, and/or explicitly

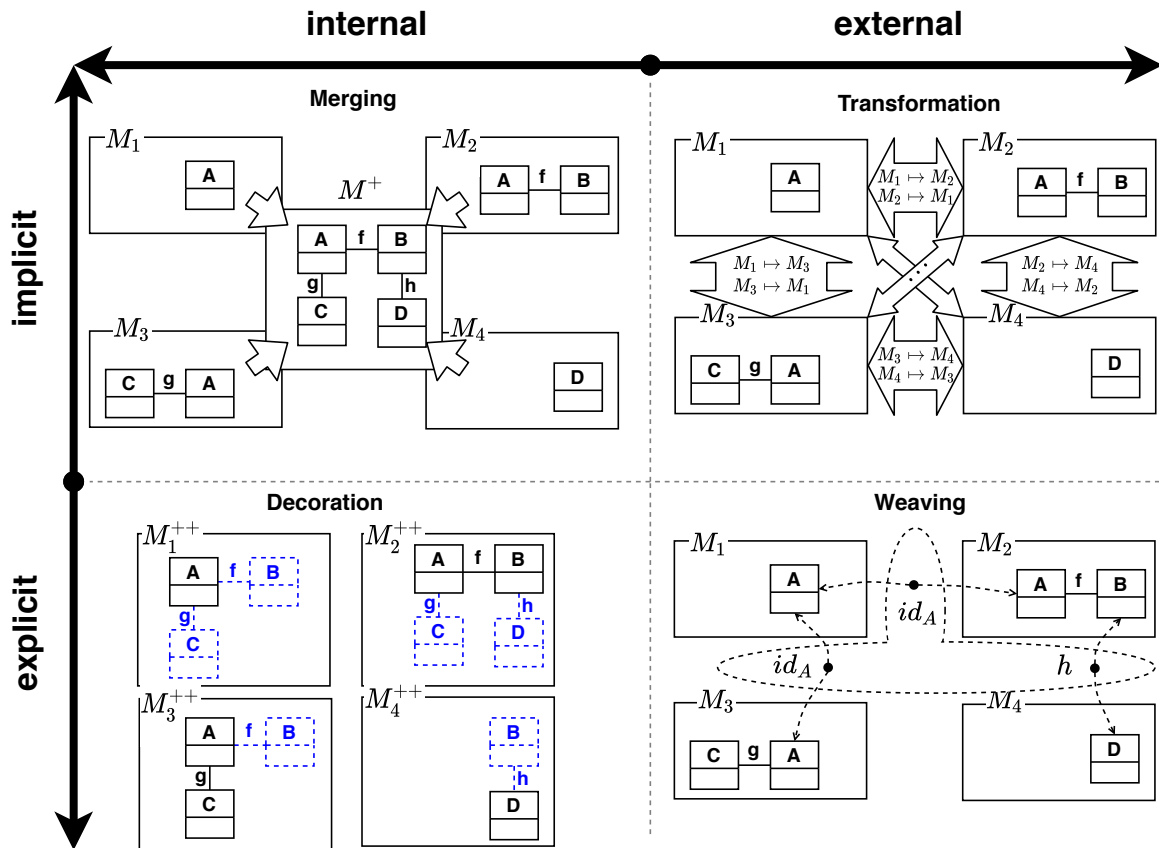


Fig. 4.7: Trace-based Commonality Representation

Customisable. Implicit means that the notion of commonalities is opaque to the respective tools [382]. The UML metamodel [365] provides many examples of Predefined definitions, e.g. every occurrence of a Class in a UML model with the same name represents the same Class. Concerning the variants of customisability, I distinguish between Formulas, Structural Properties, and Element Tuples, see also [290]. These three features can be ordered along decreasing *abstraction levels*. Element Tuples must be provided by the user and are on the lowest abstraction level. Formulas are on the highest abstraction level and also the most expressive variant. QVTr [328, 367, 439] is an example of this, which allows matching elements based on (almost) arbitrary expressions. Structural Properties [290], i.e. relating two or more elements based on their values for a specific property, are located somewhere in between.

Commonalities can be represented either in a Trace-based or Complement-based way, see also [234]. The latter are more predominant in programming-based BX approaches [179] and always require an (at least implicit) Matching phase because complements only mention the elements that are not related (negative) but never what elements are actually related [32]. A Trace-based [70, 77, 124] presentation, on the other, hand tells what elements are related (positive). I distinguish four primary ways of expressing correspondences called Merging, Weaving, Decoration, and Transformation, which arise from the combination possibilities of the dimensions *implicit vs. explicit* and *internal vs. external*, see [234]. These four approaches are sketched in Fig. 4.7:

Merging [70, 84, 275] asserts the existence of an underlying comprehensive model.

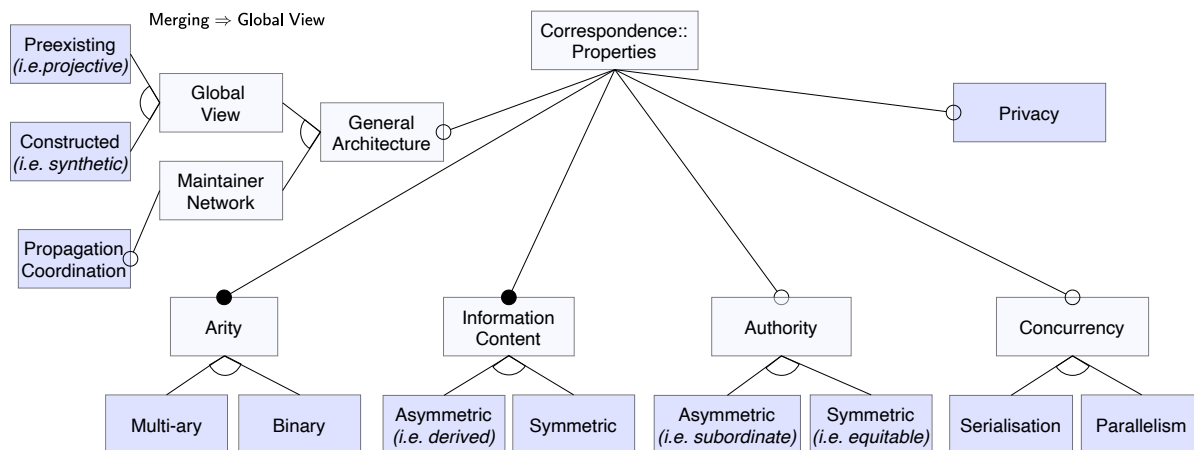


Fig. 4.8: Correspondence::Properties Feature

All models are projections of this model. Commonalities arise when an element appears in multiple of these projections. A well-known representative of this approach is the architecture of UML and also every projective approach to multi-view modeling [246] is an instance of this approach. It is important to note that underlying comprehensive model does not always exist and thus has to be constructed beforehand [25, 412],

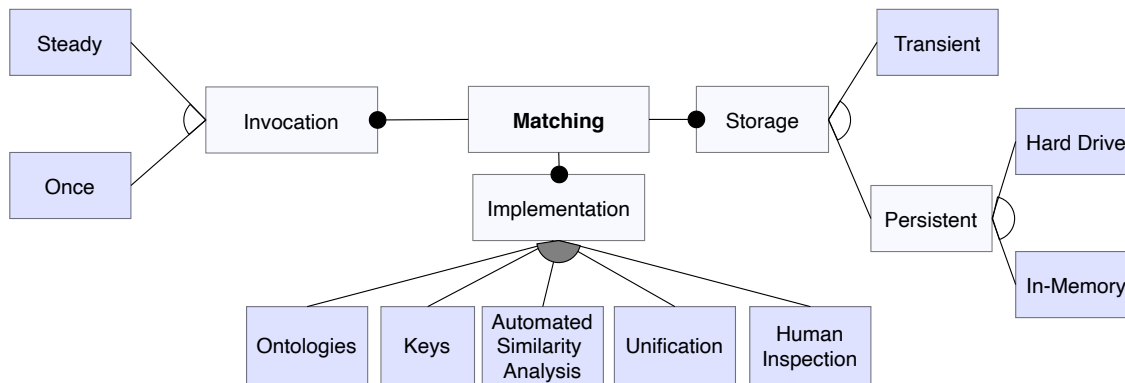
Weaving [77, 124, 138] asserts the existence of an externally stored collection of *links*. These links represent relationships between elements from disparate models, i.e. named tuples. The term “weaving” was first introduced in [77], where such links are called traces and appear as a side-product of the execution of a model transformation.

Decoration [110, 154] augments existing models with the additional information embodied in the Commonalities. This idea was pioneered in [154], where it was called *dynamic meta-modeling*: More recently, multi-level modeling techniques such as *facets* [110] have allowed implementing this approach in a lightweight and less-invasive fashion comparable to aspect-oriented programming.

Transformation [142] considers only the abstract Definition of Commonalities among the models, usually in terms of “transformation”-functions that translate the elements from one element into another. Thus, the concrete Representation remains implicit such that this approach always implies an internal Matching phase. The pioneering approach is found in [142].

PROPERTIES One of the most important Properties of the Correspondence relation is its Arity, i.e. the number of Components in the multi-model. The majority of approaches is limited to Binary situations [91, 440]. There are only a few mentions of approaches dealing with Multi-ary correspondences, e.g. [126, 279].

Note that binary multi-models not automatically mean that the respective approach cannot be applied to use cases comprising more than two models: If the respective approach considers a General Architecture for orchestrating the constituents of a multi-model consistency management process. There are two design strategies for such an architecture, which I termed Global View and Maintainer Network, Sec. 3.3.5. A well-known example of Global Views are merged models [25, 275, 412]. The Global View may either be Projective [69] (i.e. pre-existing) or Synthetic [412] (i.e. is constructed by the designers/user). In the Maintainer Network [442] approach, Coordination Propagation



Transient \Rightarrow Steady

Unification \Rightarrow Models::Formalism::Logic

Commonalities::Definition::Implicit $\wedge \Rightarrow$ Human Inspection

Keys \Rightarrow Commonalities::Definition::Predefined \vee Customisable::Structural Features

Fig. 4.9: Matching Feature

[197, 442] may be necessary, for instance, when Model Repair is applied within one multi-model this can affect other multi-models sharing the same components. In this case, central coordination is needed.

Information Content and Authority are features that have originally been introduced in [123] (where they were called information and organisation “symmetries”). The terminology in their sub-distinction, i.e. Asymmetric vs. Symmetric, stems from *lenses* [255], see Def. 3.1. A correspondence classified as Asymmetric w.r.t. Information Content describes a situation where all information in one (view) component is completely derived from another component (source). In the Symmetric case, each component contains information that is unique to it. Asymmetric w.r.t. Authority describes a similar but different notion where some components are predominant compared to others, e.g. a specification over its implementation. In the Symmetric case, all components are equitable. Note that this notion is different from Information Content because specification and implementation may contain data that is not stored in the respective other.

Authority is related to Concurrency, an important and challenging aspect, i.e. changes can happen to multiple components. Most approaches require them to happen in Serialisation [126] (on at a time). Only recently, researchers investigated how Parallel changes and thus conflicting updates can be supported [370]. Authority can play an important role in coordinating conflicts, which, however, can lead to overwritten changes and information loss.

The importance of Privacy [259] has been identified in BX recently and it was pointed out that it so far has received less attention in the literature. In practice, it plays a major role in many scenarios with strict legal privacy requirements, e.g. the health care sector. Concretely, this means that not all elements contained in a component model are accessible. It necessitates filtering/obfuscation and increases the degree of manual activities due to approval processes.

4.2.5 Matching

Matching [32, 70, 145] can be described as the procedure that turns a Commonalities::Definition into a Commonalities::Representation. It is one of the three essential Operations of multi-model consistency management and is depicted in Fig. 4.9.

In [433], Spanoudakis and Zisman identify for primary strategies for the Implementation of model matching. I adopt this classification and added Keys as to their list. An implementation following that strategy utilises a circumstance where one can evaluate a unique value for a given model element (e.g. the element name). If two or more elements from separate models evaluate to the same key, they are matched [290]. This can be implemented very efficiently utilising hash-based lookup data structures [338].

Unification [285] is used in combination with logic-based approaches: Two elements are matched if their logical representation can be rewritten to the same *term*.

Ontologies require to semantically annotate model elements. Two or more elements are matched if they are annotated with the same semantic concept. A special case are thesauri (e.g. *WordNet* [169]) which contain synonyms and hypernyms of terms.

Automated Similarity Analysis is the common name for all implementations that compare tuples of suitable elements with each other and establish a commonality if they show a similarity, i.e. their distance w.r.t. a given metric is under a specified threshold. This may involve *optimisation* algorithms [312]. In general, this is an NP-complete problem [404] as it can be reduced to weighted bipartite graph matching.

Thus, Human Inspection may be required to identify commonalities among elements [42]. This is necessary when the notion of commonalities is informal but executing Matching this way is time consuming [433].

The Invocation of the Matching procedure happens either Once [211] or Steadily [311]. The Storage of the Commonalities is Transient or Persistent. If it is Transient, this means that Matching is essentially a step in a superordinate operation (i.e. Verification or Repair) and the results of this operation are immediately passed on. Transient automatically implies Steady. Persistent storage means that the results of the Matching can be reused and Persistent in combination with Transient can be interpreted as incremental matching [311]. In-Memory [292] storage means that these results are available only during the execution time of the respective model management tool, while Hard Drive storage means that the results are stored in a distinct file, which can be re-used by other external tools.

4.2.6 Consistency

Global Consistency is the eventual goal of multi-model consistency management. For this feature, I distinguish the two aspects Rules and Inconsistency Report, which is shown in Fig. 4.10.

The classic characterisation [155, 284, 469] distinguishes between Structural and Behavioural consistency rules (Nature) as well as Intra-Model and Inter-Model consistency rules (Scope).

One may further consider different Severity Level of a consistency rule. An example is a distinction between proper *constraints* and *critiques* [288]. These levels may be Fixed by the tool or Customisable.

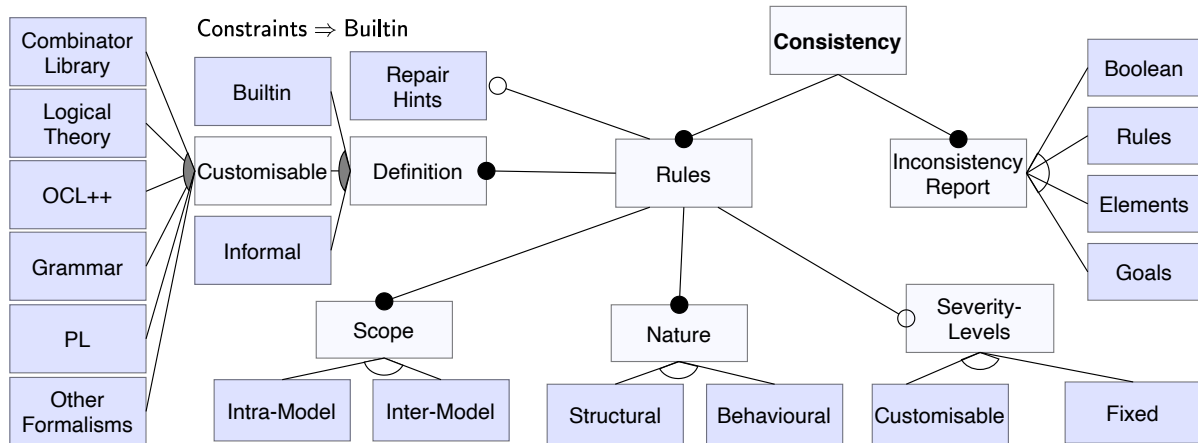


Fig. 4.10: Consistency Feature Dimension

Consistency rule can be augmented with Repair Hints [235, 286, 413], i.e. indicate a strategy how to resolve an inconsistency related to the rule. Repair Hints generally imply a Rule-based repair approach, see Sec. 4.2.8.

The set of defined consistency rules is Builtin [396], Customisable [288] or Informal [385]. There are various formalisms for defining the consistency rules. Combinator Libraries [178, 235, 286] are primarily utilised in programming based synchronisation tools, where the notion of consistency is implicitly derived from the semantics of a set of predicates and composition operators. In Grammar-based consistency rule languages, consistency arises as the set of (multi-)models that can be produced from production rules [403, 422]. The implementation of the consistency check is thus based on pattern matching [331]. A common way of defining consistency rules is by means of a Logical Theory, e.g. FOL [244]. In MDSE, OCL [363] together with its variants [293, 367] are widely used formal languages to define domain specific consistency rules. Another way is to let users write a functioning in some PL [288] returning true or false, which defines the semantics of a consistency rule and allows harnessing the features of the respective language. Other Formalisms such as Petri-Nets or state machines are generally used for defining the semantics of behavioural consistency rules [155].

The form of the Inconsistency Report differs from approach to approach. The plainest type is a Boolean [286] only saying if the model is consistent or not. More instructive is to tell which Rules are violated. Even more insightful is to additionally report the Elements [396] that are violating it. Another option is to directly point to the Goal for the subsequent repair process [385].

4.2.7 Verification

Given a model or multi-model, Verification turns Consistency Rules into a (hopefully empty) Inconsistency Report. This feature is one of the three essential Operations in multi-model consistency management and is visualised in Fig. 4.11.

The Invocation of the Verification procedure may be Manual [156, 313] (i.e. the user explicitly invokes it), Event-Based [235, 328] (e.g. the tool monitors the model files and it is triggered upon a modification), or Policy-Based [173]. The Execution of the Verification is either Interactive (involving a human) or Automatic.

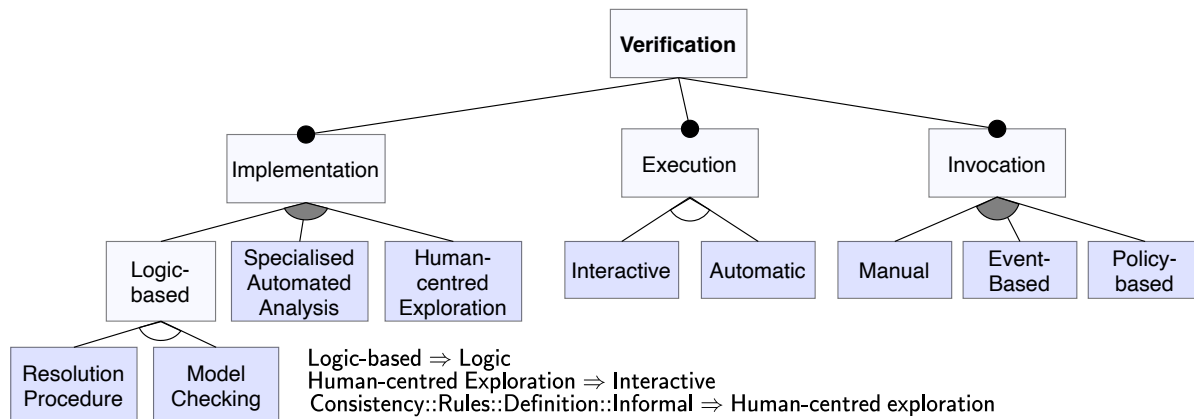


Fig. 4.11: Operations::Verification Feature

Concerning Implementation, Spanoudakis and Zisman [433] identified four approaches whereof two can be grouped as Logic-based. The first logic-based variant encodes all models and consistency rules as logical axioms in a knowledge base and utilises a Resolution Procedure (e.g. a theorem prover) to check that the knowledge base is free of contradictions. Alternatively to this syntax-based approach, there is Model Checking, i.e. enumerating all possible instances of a theory and looking for an instance that satisfies it. The logic-based approaches have the advantage that they are very generic and can be applied to a big range of domains as long as they admit the necessary representation. The main issues are the possibility of non-decidability (depending on the respective logic) and complexity (state explosion problem). Thus researchers have developed Specialized Automated Analysis tools, which are based on a specific formalism [502] or technology [357]. These solutions are more effective for certain problems but less universal. Finally, a Human-centred exploration approach requires the most effort, however, it is the only way to discover inconsistencies for informally given models and consistency rules [433].

4.2.8 Repair

The restoration of detected inconsistencies is known as Model Repair, the third essential Operation of the multi-model consistency management framework. The topmost level of this feature dimension is shown in Fig. 4.12. The complex sub-features Implementation, Human Interaction and Formal Guarantees are depicted in greater detail in Fig. 4.13, Fig. 4.14, and Fig. 4.14, respectively.

The Invocation of the Repair procedure can be Manual, Event-Based, and/or Policy-based. Eventually, Repair produces a Result. Here, one can distinguish between Presentation and Cardinality, see [331]. The Presentation is either given as a Change or as a State. Note that this may be different compared to the presentation of Change used otherwise. The Presentation can be abstract in a sense that it contains placeholders (e.g. for attribute values of newly created elements). In the database jargon they are termed Labelled Nulls [21] and indicate for the user to take further actions.

Secondly, one can analyse the Cardinality of the result set. Multiple results imply user interaction in the form of Result Selection. The tool can leverage this activity via an Ordered [331] presentation. This requires an underlying metric to compare results, e.g.

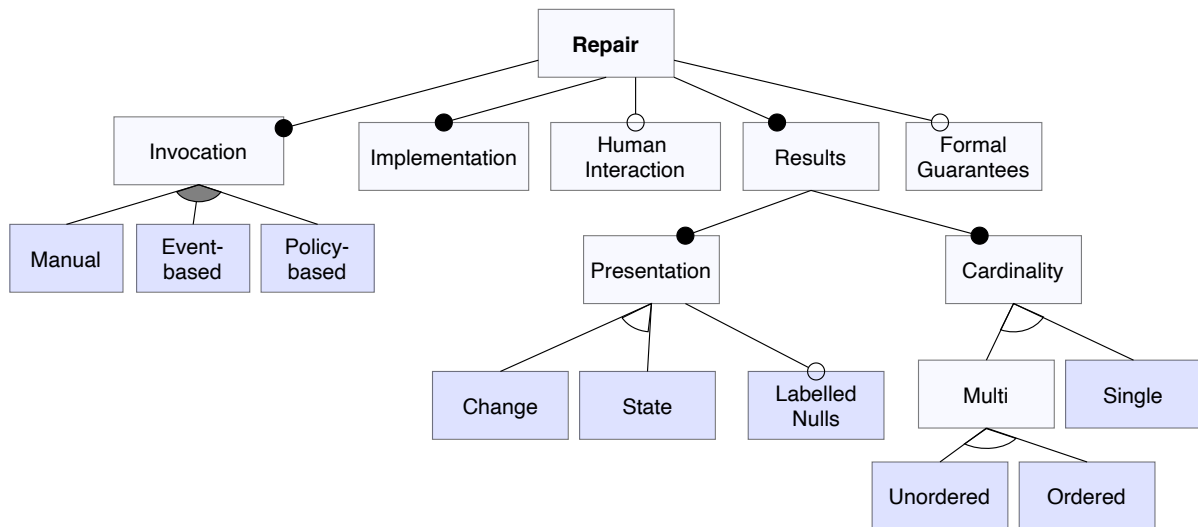


Fig. 4.12: Repair Feature - Overview

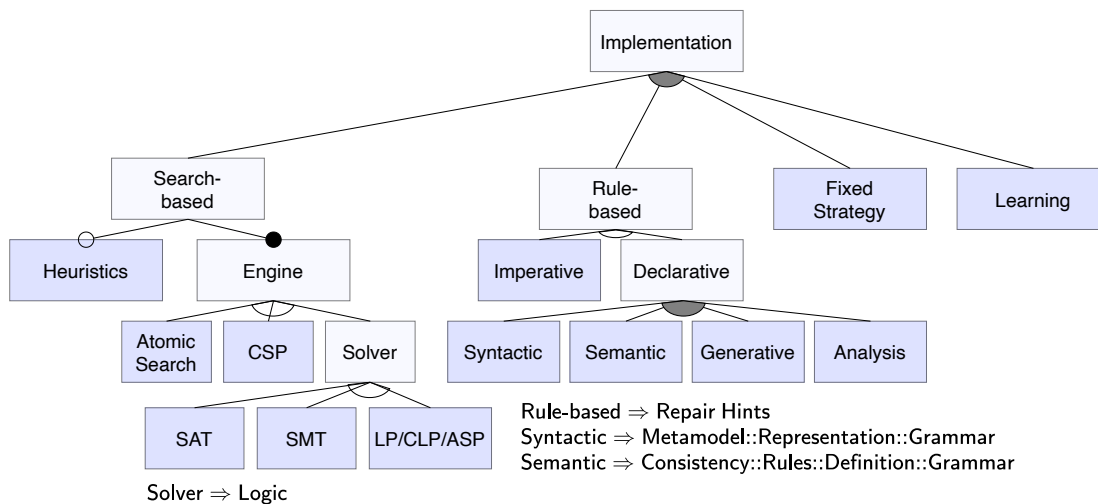


Fig. 4.13: Repair::Implementation Feature

induced by costs, policies, or inconsistency levels. An interesting approach is taken in JTL [88], which works with multiple results simultaneously over time.

IMPLEMENTATION In general, there are two strategies for implementing model repair: Search-based and Rule-based, which can be supplemented by a third minor supportive approach (Fixed Strategy) and/or Learning, see Fig. 4.13.

SEARCH-BASED This generic approach can intuitively be described as letting the procedure figure out a solution by “trial-and-error” via considering model repair as a *search problem*. It comprises a *state space* given by the set of all (multi-)models. The *transitions* are given by the set of all possible updates. The *start state* is the current (inconsistent) model and *goal states* are all those models which are considered consistent.

The strength of this approach is its domain independence and it can easily be adapted to new or changed scenarios. The weakness and thus the biggest challenge of this approach is its computational complexity. Already for a “small” state space,

the naive Atomic Search [274, 428] quickly runs out of time or memory. It is well known, that so called Heuristics can improve the efficiency of search algorithms to make complex problems tractable [377]. However, finding a suitable heuristic function remains tricky. One may think of generic heuristic function such as the number of violations, however domain specific knowledge tends to provide even more effective heuristics [281].

Thus, it is common to consider the problem as a constraint satisfaction problem (CSP) [308]: In CSP search space states are not atomic but have an internal structure. This structure is given by a set of *variables*, where each variable can take one value from a given *domain* and every variable is subject to one or more *constraints*. This factored presentation allows for a much more effective search because searching means to vary the value of only those variables, which are affected by a constraint violation.

Yet another approach is satisfiability (SAT) solving [108]. It actually represents a special case of a CSP where all variables are boolean and all consistency rules must be formulated in propositional logic. SAT solvers represent its own research domain, which has produced remarkable results and performance improvements that can be exploited for implementing model repair [391]. The translation of a whole problem domain into this formalism quickly leads to a proliferation of variables and propositions. Thus, a more convenient and high-level way is to use a satisfiability modulo theories (SMT) solver [346], which offers a more abstract interface by providing a set of built-in theories, e.g. arithmetics on integers, manipulations of character-strings, etc. These theories have their own highly optimized translation into the underlying SAT-presentation. An example of an SMT-solver that is often used in the context of object-oriented modeling is *Alloy* [249], which offers a built-in relational theory that in turn resembles object-oriented design and notation. SMT-solvers are a popular choice for model repair and used in a wide number of approaches, e.g. [281, 328, 443].

The technique used in SAT and SMT solving is called *model checking*, see Sec. 4.2.7. Instead of using model checking, one can alternatively use syntactical reasoning: The dynamics (i.e. allowed updates) of the domain are encoded in logical statements and the repair is formulated as a query asking whether a consistent model state can be reached by a sequence of changes. When the query can be fulfilled in the present knowledge base, a repair is found. Implementations of this approach are Logic Programming (LP), Constraint Logic Programming (CLP) and Answer Set Programming (ASP) [88, 156, 385]. Due to its nature there are certain restrictions on the type of logical sentences that can be used, e.g. they must be quantifier free, do not contain negation (in case of LP), etc.

RULE-BASED Rule-based solutions explicitly tell the program *how* to fix a certain inconsistency. These instructions are given as *rules* in the form:

IF *condition* THEN *action*.

A *condition* represents a specific consistency rule violation and *action* is a sequence of edit operations fixing this inconsistency. Thus, in rule-based approaches the definition of consistency and repair rules is often tightly connected. It heavily depends on the domain expert to define the *right* set of rules. Thus, rule-based solutions are not universal and cannot easily adapt to different scenarios. The strength of this approach,

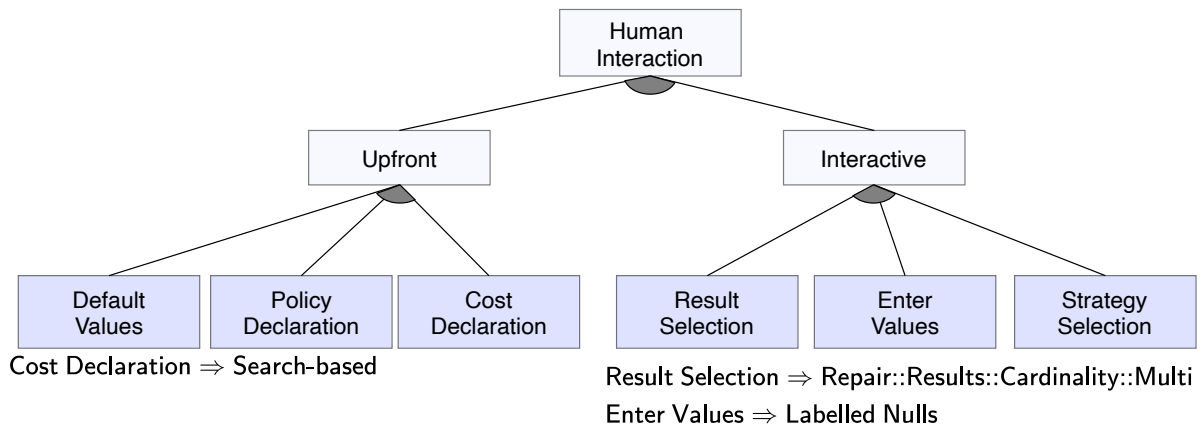


Fig. 4.14: Repair::Human Interaction Feature

however, is its efficiency: After having identified the rule, which must be applied to a given violation, repair is performed in constant time. This lookup may drastically affect the efficiency of the whole procedure (think of it as a search-problem of its own).

Rule-based solutions can be classified into Operational [357, 413, 499] and Declarative [230, 396]. Operational rules are procedures written in a programming language. In general, operational rules provide no guarantee that they actually lead to the desired result and it is up to the user to define the rule correctly. Thus, researchers came up with Declarative rules [341], which can be statically analysed. Arguably, the most popular declarative rule-based framework is given by the *(Algebraic) graph transformation (GT)* framework [146], which offers powerful means to statically Analyse concurrency, confluence, conflicting and termination properties of a set of rules. An example of a rule-based approach combining graph transformation and user interaction is found in [353, 354]. Another feature of the declarative approach is the ability to Generate [144, 396] repair rules automatically, which significantly reduces the manual effort. Such approaches are usually based on grammars, which can be classified into Syntactic [396] or Semantic [230, 422]. The syntactic category exploits the fact that (modeling) languages are generally defined in terms of a grammar, which can be used to derive potential changes. The semantic category requires the consistency rules to be defined in terms of a grammar as well.

FIXED STRATEGY & LEARNING One may consider a third category called Fixed Strategy. It is generally combined with one of the above strategies, usually rule-based ones [235, 286]. It means that the approach considers a built-in procedure for model repair. A prominent example of fixed strategies are *constant complements* [31], see Sec. 3.1.11

Learning [406] is the ability of a program to improve its performance on a given task over time. It appears to be a promising approach to improve the performance of search-based approaches [36] and can help to identify hidden policies and user preferences.

HUMAN INTERACTION Some researchers [396] have argued that the user must play a leading role in the model repair process. I distinguish several ways of human interaction,

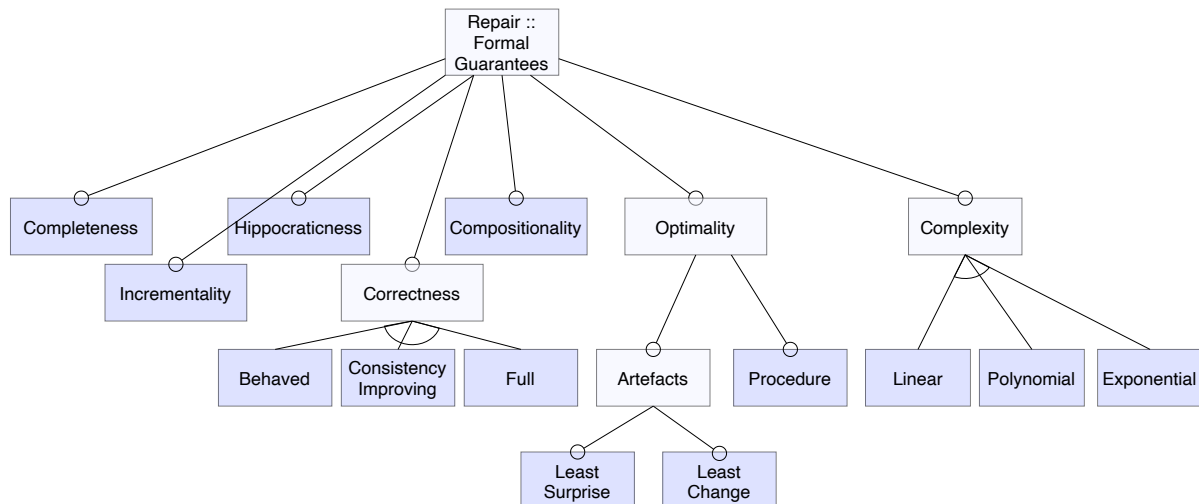


Fig. 4.15: Repair::Formal Guarantees Feature

grouped into Upfront (performed before the repair invocation) and Interactive (performed during the repair) measures. A typical example of an upfront measure is the definition of Default Values, which will be used when new elements are created during the repair. An upfront measure with less obvious implications is Cost Definition [328], which associate a cost with each update and thus influencing the repair results because the tool will try to minimize these costs. Another sophisticated tool is given by Update Policies [122], which prescribe the rationale for picking a certain solution among several choices. The predestined example for an interactive measure is Result Selection, i.e. the tool calculates all possibilities resulting in a consistent result and the user picks his preferred choice. The interactive equivalent of default values is requesting the user to Enter Values for missing attribute values on the way. Strategy Selection [385] is similar to result selection with the difference that the outcome is unclear, i.e. the tool presents the user with a choice of “abstract” edit sequences.

FORMAL GUARANTEES The analysis of the Formal Guarantees of a model repair approach has been popularised by the BX research domain. Correctness is the fundamental property safeguarding that the repair produces a consistent result. The distinction between three stages of this feature: Behaved (the results do not add new inconsistencies) vs. Improving (the results are less inconsistent than the input) vs. Full (the results are completely consistent), was introduced in [331].

The repair procedure is called Complete [230] when it produces at least one result for every input, i.e. there are no “unresolvable” inconsistencies.

Hippocraticness expresses the idea that the repair operation applied to a consistent (multi-model) should not do anything (“does no harm”). Meertens [339] formulated accentuation of this concept: A repair should not change more than necessary.

Incrementality [194] describes the ability of the repair procedure to reuse information produced by previous invocations. This also means that, the complexity is must be relative to the “size” of the update and not relative to the “size” of the model.

Compositionality plays a central role in many theoretical approaches, e.g. the lens framework [133, 178, 258], and describes the ability of composing elemental repair

“primitives” into bigger ones whilst preserving their (formal) properties.

Furthermore, one can analyse the Complexity (concerning the implementation) and Optimality of the repair procedure. The latter can further be distinguished w.r.t. the produced Artefacts or the Procedure itself. Concerning artefacts, there are two concepts: Least Change is based on a metric on the model space and states that the repair always prefers results with a minimal “distance” from the original model [328]. Others [86] argued that the “least changed” repair option may not always be the preferred choice and thus proposed the concept of Least Surprise, which is based on the notion of continuity [86]. Analysing the optimality of the repair procedure was proposed in [442], which considers a repair optimal if and only if it requires as few invocations of elementary repair “primitives” as possible.

4.3 Observations

During the literature study and the development of the feature model, several observations were made that are collected in the list below:

Tool Quality A noteworthy share of tools, for instance, most of the ones mentioned in [234], is no longer retrievable because of broken URLs, non existing source code repositories, or dependencies on outdated Eclipse versions, which are no longer supported by current operating systems. Thus, there is a need for stable tool repositories [264] and benchmarks[18] to make tool evaluations reproducible.

Standards EMF has become a de-facto standard in the MDSE community. Thus, XMI and Ecore are the common file formats for denoting models and metamodels. The situation is more diverse when it comes to representing changes, consistency rules and commonalities. The XMI standard [366] comprises a concept for a change-based representation of models but its tool support seems limited concerning the often mentioned necessity for model differencing [10, 270, 290]. In an MDSE context, consistency rules are often encoded in OCL (or its variations) [80]. However, its tool support is not as widespread compared to Ecore and XMI [199]. Regarding the representation of commonalities, there is no consensus and researchers utilise diverse approaches, see Sec. 4.2.4. Therefore interoperability among tools is often limited [489].

Languages for Global Consistency Rules Related to the last statement of the previous bullet point, is the fact that there is no real consensus on a “language” for denoting consistency rules over multi-models. The QVT standard [367] can be seen as an attempt in this direction, but suffers from semantic ambiguities and lacks practical implementations [439]. Hence, researchers proposed heterogeneous approaches, e.g. constraints over trace models [413] or grammars [422]. In my feature model, I treat the definition of commonalities (i.e. *match*-rules) and consistency rules (i.e. *check*-rules) separately while other approaches (e.g. QVTr and grammars) mix both aspects. I conclude that there is a need for a better understanding about the types and nature of global consistency rules and commonalities.

Binary vs. Multi-ary The majority of inter-model consistency verification and model synchronisation approaches only considers *binary* correspondence relations, see e.g. [91, 126, 279, 441].

Repair Strategies There are clusters of features that are used in combination, which is induced by the respective implementation strategies: Rule-based repair approaches are associated with a definition of consistency rules that is based on a grammar or combinator libraries, e.g. [230, 235]. Approaches, which utilise a more abstract logic-based formulation of consistency rules are more likely to implement model repair in a search-based way, e.g. [156, 328]. Yet, it becomes apparent that both strategies are not competing with each other but could be combined in addition to supporting human interaction and learning into a comprehensive framework. In general, the repair problem remains NP-complete (= the complexity of abstract search problems) and solving it once and for all with a fully-automatic repair program is unfeasible. Hence, it may be worthwhile to focus more on specific application domains where one can harness domain dependent expert knowledge, which can help to find “the best” solution.

4.4 Demonstration of Selected Approaches

I will now have a closer look at three concrete MDSE tools and their ability to address the multi-model consistency management problems scenarios in Sec. 1.3 in order to uncover the conceptual limitations of existing solutions. This opinionated selection of tools features three very distinct tools, which shall illustrate the broad spectrum of approaches. The choice of these particular tools was based on (1) relative occurrences in the literature and (2) availability and quality of the associated binaries, i.e. public accessible download page or source code repository and the possibility to run the tool on my machine. A classification of these tools w.r.t. the above feature model is given in Tab. 4.2 and explained in greater detail below.

4.4.1 Echo

*Echo*³ [327, 328, 330] is an academic prototype developed by Macedo and collaborators at the *University of Minho*. The tool was originally designed as an implementation of QVTr [327] but afterwards it has been extended to offer general model verification and repair operations supporting both regular (local) models and multi-models [328]. Thus, it addresses all aspects of multi-model consistency management. The tool is available as a plugin for the *Eclipse* integrated development environment (IDE). The plugin dependencies are somewhat outdated and the last commit on the source code repository happened in 2018 but it is still possible to install it in the current version of the Eclipse IDE. Echo has been chosen as a representative for the class of tools and approaches, which follow a logic-based approach, e.g. *USE*⁴ [198, 200], *JTL*⁵ [88, 156] or *Badger* [385]. It has been featured in multiple model repair tool comparisons [331, 383].

³<http://haslab.github.io/echo/>

⁴<https://sourceforge.net/projects/useocl/>

⁵<https://jtl.univaq.it/>

4.4 Demonstration of Selected Approaches

	Echo	Epsilon	Emoflon
Models			
Tech Space	XMI	XMI, XML, Other	DSL, (XMI)
Formalism	Logic	OO	Graphs
Change			
Representation	State-based	State-based	Structural Deltas
Recording	-	-	Offline, Model Differencing::Syntactic
Allowed Updates			
Definition	Customisable::Explicit	Customisable::Explicit	Customisable::Implicit
Types::Atomic	Insert, Update, Delete, Move	Insert, Update, Delete, Move	Insert, Update, Delete, Move
Types::Complex	Sequence, Parallelism	Sequence	Sequence, Parallelism
Meta-Information	-	Environment Data	Previous State
Conformance			
Well-Formedness	Typing, Constraints	Typing, Constraints	Typing
Metamodels			
Definition	Customisable::Freely	Customisable::Freely	Customisable::Freely
Tech Space	MOF	MOF	DSL, (MOF)
Formalism	Knowledge Base	-	Type Graph
Correspondence			
Components	Heterogeneous::Metamodels	Heterogeneous::(Metamodels, Tech Spaces)	Heterogeneous::Metamodels
Commonalities			
Representation	Transformation	Merging	Weaving
Definition	Customisable::Formulas	Customisable::Formulas	Customisable::Formulas
Properties	-	-	-
Arity	Binary	Binary	Binary
General Architecture	Maintainer Network	Global View::Constructed	Maintainer Network
Information Content	Symmetric	Symmetric	Symmetric
Authority	-	-	-
Concurrency	Serialisation	Serialisation	Serialisation
Privacy	-	-	-
Matching			
Invocation	Steady	Once	Steady
Storage	Transient	In-Memory	Hard Drive
Implementation	Unification	Automated Similarity Analysis	Automated Similarity Analysis
Consistency			
Inconsistency Report	Rules	Elements	Elements
Rules			
Definition	Builtin, Customisable: OCL++	Builtin, Customisable: OCL++	Customisable: Grammar
Nature	Structural	Structural	Structural
Scope	Inter-Model	Inter-Model	Inter-Model
Severity Levels	-	Customisable	-
Repair Hints	-	yes	yes (implicit)
Verification			
Execution	Automatic	Automatic	Automatic
Invocation	Event-based	Manual	Manual
Implementation	Model Checking	Specialised Autom. Analysis	Specialised Autom. Analysis
Repair			
Invocation	Manual	Manual	Manual
Implementation	SMT	Imperative	Declarative (Semantic, Generative)
Human Interaction			
Upfront	Policy declaration	-	-
Interactive	Result Selection	Strategy Selection	-
Results::Presentation	State	State	Change
Results::Cardinality	Multi::Ordered	Single	Single
Invocation	Manual	Manual	Manual
Formal Guarantees	Correctness::Full, Least Change	-	Correctness::Full, Hippocraticness, Completeness, Incrementality

Table 4.2: Feature Model Classification of selected Tools

State of the Art

Echo builds on top of EMF. Hence, metamodels are defined using Ecore and models are stored as XMI files. Local consistency rules are formulated as OCL invariants, which are attached to the Ecore metamodels. Global consistency rules are defined using QVTr. However, the semantic interpretation of this language differs from the official OMG standard [329]. List. 4.1 shows how CR₅ (“BusinessRuleActivities” have associated “DecisionTables”) from the software modeling example in Sec. 1.3.2 is denoted.

```
transformation bpmn2dmn (bpmn : BPMN, dmn : DMN) {  
  
  top relation businessRule2decisionTable {  
    n : String  
    domain bpmn act : Activity {  
      type = ActivityType::BUSINESS_RULE  
      name = n  
    };  
    domain dmn tab : DecisionTable {  
      name = n  
    };  
  }  
}
```

Listing 4.1: CR₅ in QVTr

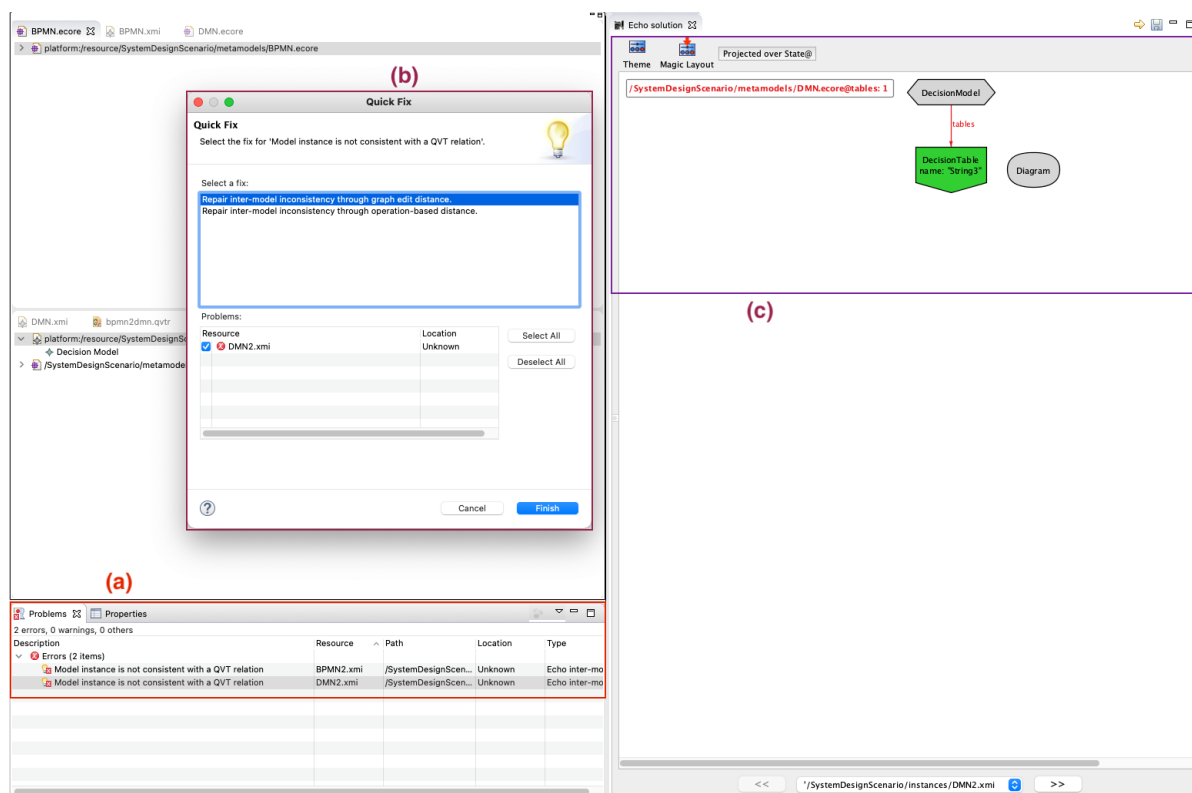


Fig. 4.16: Screenshot the Eclipse Echo plugin: (a) detected inconsistencies in the (b) metric selection for repair (c) repair result presentation using Alloy

Internally, Echo translates (Ecore) metamodels, (XMI) models and (OCL and QVTr) consistency rules into a logical representation. More concretely, it utilises the Alloy⁶ [249] model finder. Alloy is an SMT solver with a *relational logic* interface and facilitates both consistency verification and model repair via (bounded) model finding, i.e.

⁶<https://alloytools.org/>

searching for an instance that complies with the logical theory. A special feature of the Echo approach is the built-in notion of *least change* [328], i.e. during model repair Echo first looks for valid instances having a minimal *distance* to the original model. There are two possible distance metrics: A graph-edit distance, or an operational edit distance, which is configured by writing OCL pre- and post-conditions for the operations in the metamodel.

Fig. 4.16 contains a screenshot of the Eclipse editor with the Echo plugin installed. Echo actively monitors the consistency of single models w.r.t. their metamodel and pairs of models w.r.t. a QVTr-transformation definition. Consistency is checked after every modification to an observed resource and detected inconsistencies are reported to the user through the IDE (a). The user can choose to fix the inconsistency and trigger the model repair procedure they have to pick the preferred distance metric (b). The tool then proposes possible fixes to the user utilising the Alloy visualisation mechanism (c). The complete feature classification of the Echo approach is shown in the first column of Tab. 4.2.

4.4.2 Epsilon

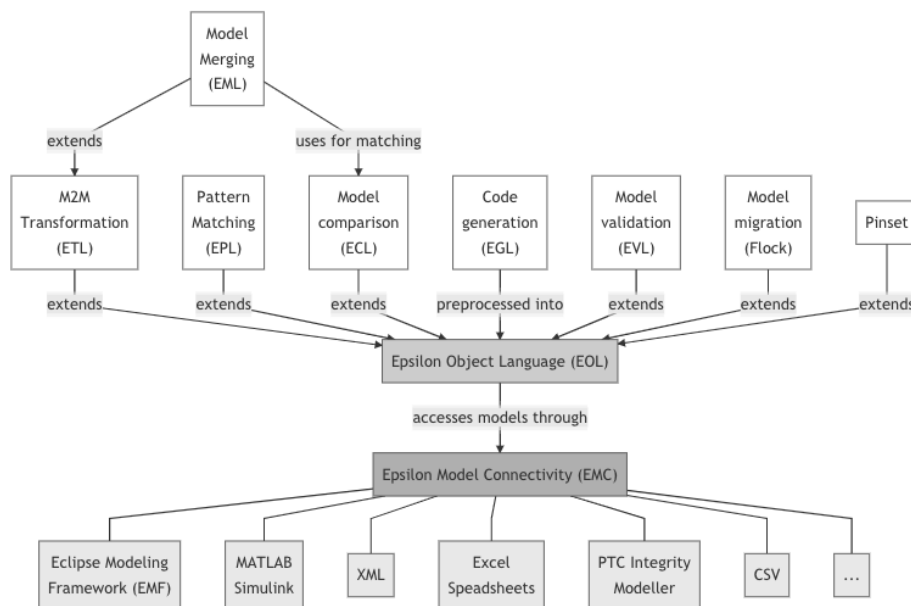


Fig. 4.17: Epsilon tool architecture

*Epsilon*⁷ [371] is, alongside *ATL*⁸ [47], one of the most mature and actively developed academic MDSE tool ecosystems, which is also used in the industry. It is being developed at the *University of York* and started with an imperative model transformation language called *Epsilon Object Language* [371]. The framework has subsequently been extended by various DSLs, one for each model management task. The user writes a program in one of these languages to configure the respective model management operation. This program is executed on the Epsilon runtime, which is available both as

⁷<https://www.eclipse.org/epsilon/>

⁸<https://www.eclipse.org/atl/>

a plugin for the Eclipse IDE and as a standalone Java application. Moreover, it offers an abstraction over several popular model serialisation formats (including Ecore/XML, free form XML documents, Microsoft Excel sheets, Simulink models and more). Thus, Epsilon programs can be executed on models stemming from various technological spaces. Fig. 4.17 gives an overview of the Epsilon architecture.

CONSISTENCY CHECKING VIA MERGING Epsilon is a generic model management framework, which allows to implement different multi-model consistency management approaches on top of it. However, it comes with an “intended” strategy for implementing global verification among multi-models via merging [412] utilising the *Epsilon Merging Language (EML)* [292] and the *Epsilon Verification Language (EVL)* [288]. The latter is an OCL-like language in which consistency rules – called constraints – are defined.

```

1 context BPMN!Activity {
2   constraint CR5 {
3     check : self.type = BPMN!ActivityType:BUSINESS_RULE implies
4           DMN!DecisionTable.allInstances().select(t|t.name = self.name).count() = 1
5     message : "The business rule activity " + self.name + " has no corresponding decision table"
6     fix {
7       title: "Add decision table"
8       do {
9         var tab : new DMN!DecisionTable
10        tab.name = self.name
11      }
12    }
13  }
14 }

```

Listing 4.2: Example of an EVL constraint

For example, consider List. 4.2, which exemplifies how **CR₅** is implemented in Epsilon. The constraint (line 2) is attached to a model element type (line 1) and must implement a check function (lines 3-4). Here, the user writes an EOL-statement that implements the verification procedure. Similar to OCL, EVL offers an `allInstances()`-function returning a collection of all model elements with a particular type and it is possible to query elements from multiple disparate models simultaneously (note the **BPMN!** and **DMN!** prefixes in List. 4.2) such that inter- and intra-model consistency rules can be implemented. Moreover, the user can configure the text of the error message (line 5) that is shown if the constraint is violated, and can also define an imperative EOL program to restore the inconsistency (lines 6-12). The verification is triggered manually by executing the EVL program on a collection of models. The repair can be triggered by applying it as a “hot fix” to a discovered inconsistency (requires Eclipse IDE).

The implementation of **CR₅** in List. 4.2 relied on `BusinessRulesActivities` and corresponding `DecisionTables` having the same name. However, in practice it is not always feasible to rely on unique identifiers such as names to relate elements from disparate models. Thus, model matching (Fig. 3.9c) is required, which is offered via the *Epsilon Comparison Language (ECL)* [289].

```

1 rule MatchBRActivityAndDecisionTable match l:BPMN!Activity with r:DMN!DecisionTable {
2   guard : l.type = ActivityType:BUSINESS_RULE
3   compare : l.name = r.name
4 }
5 rule MatchDataObjectAndColumn match l:BPMN!DataObject with r:DMN!Column {
6   guard: l.consumers.matches().inputSideColumns.includes(r) or
7         l.producers.matches().outputSideColumns.includes(r)
8   compare: l.name.isAlike(r.name)

```

```

9 }
10 operation String isAlike(other: String): Boolean {
11     ... // Custom implementation (utilising Ontologies, etc.)
12 }

```

Listing 4.3: ECL matching rules

List. 4.3 gives an example comprising two ECL rules that match `BusinessRuleActivities` with `DecisionTables` and `DataObjects` with `Columns`. A rule defines *what* type of model element should be compared with each other (match and with), *when* they should be matched (compare), and optionally a filter-criterion (guard). Both, guard and compare must return a boolean value and can contain arbitrary EOL statements. This allows the user to apply arbitrary matching techniques, see the function `isAlike` in line 10 in List. 4.3. Epsilon makes sure that there are no cyclic invocations of match rules and that each element is matched *at most once*, i.e. it picks first possible match (greedy). The execution of the ECL program produces a `MatchTrace`, which has to be processed further to create a *merged model*. The construction of the latter is configured via EML rules, see List. 4.4.

```

1 /* Merge rules */
2 rule MergeBRActivityAndDecisionTable merge l: BPMN!Activity with r: DMN!DecisionTable
3     into t : Merge!DecisionTableDef {
4         t.name = l.name;
5         t.isMatched = true;
6     }
7 rule MergeDataObjectAndColumn merge l : BPMN!DataObject with r : DMN!Column
8     into t : Merge!DataObjectCorrespondence {
9         t.name = l.name;
10    }
11 /* Copy rules */
12 rule CopyUnmatchedBusinessRuleActivity transform s : BPMN!BusinessRuleActivity
13     to t : Merge!DecisionTableDef {
14         guard : s.type = ActivityType:BUSINESS_RULE
15         t.name = s.name;
16         t.isMatched = false;
17    }

```

Listing 4.4: EML merging rules

An EML program is comprised of merging (lines 8-10, keyword `merge`) and copying (lines 11-17, keyword `transform`) rules. The merging rules are invoked for each `MatchTrace`-object, which were identified by a preceding ECL-program, and produce a new element in the merged model. The copying rules are invoked on all unmatched elements and produce a respective element in the merged model. The body of these rules contains arbitrary EOL statements, which can be used to set the attribute values of the newly created elements in the merged model. The merged model is a new artefact and serves as a global view, compare Sec. 3.3. It is important to note that the scenario in Sec. 1.3.2 involves *heterogeneous* modeling languages. This means that in order to create the merged model, one first requires a language comprising concepts from all involved languages, i.e. a metamodel encompassing BPMN, DMN and UML modulo common concepts. Note that this induces another instance of *homogenous* model matching and merging on the level of MOF-models, which is usually done manually. For more details about this issue, I refer to [134].

```

1 context Merge!DecisionTableDef {
2     constraint CR5 {
3         check : self.isMatched
4         fix { var table : new DMN!DecisionTable;

```

```

5         table.name = self.name;
6     }
7 }
8 }

```

Listing 4.5: Global Constraints

Finally, the merged model allows verifying the global consistency rule [CR5](#) in one central place, see [List. 4.5](#). Comparing the implementation in [List. 4.5](#) with [List. 4.2](#), the former is much shorter since it is based on the value of a single attribute (`isMatched`). This attribute has been introduced in the merging phase, see [List. 4.4](#) line 5 and 16. Depending on whether the `DecisionTableDef`-object has been created by a merging or copying rule, this attribute is either `true` or `false`. Finally, note that the attached repair action (lines 4-6) propagate their effect on the original model (DMN) and not the merged model.

CONSISTENCY CHECKING VIA WEAVING Due to its flexible architecture, Epsilon supports other multi-model consistency management approaches as well. Instead of creating a merged model, one might have stopped after the matching phase and continued working with the identified `MatchTraces`. According to the terminology used in this thesis, `MatchTraces` are rightly called *commonalities*. They represent an entity of their own right [\[124\]](#) and therefore it is worthwhile to store them in their own model. This particular type of a model is often called a *trace model* [\[114\]](#) and the general approach is known as *Weaving*. Systematic approaches that combine Epsilon with trace models have been reported in [\[168, 413\]](#).

In its most generic form, a trace model is nothing but a (hyper-) graph. Elements are either `TraceLinks` (edges) or `TraceLinkEnds` (vertices). The latter represent *proxies* [\[191\]](#) of elements in another model. The upper half of [Fig. 4.18](#) depicts the metamodel of generic trace models. Note the attribute represents of the `TraceLinkEnd` class. The type of this attribute (`ElementIdentifier`) shall indicate that its value is a pointer to some model element. This “trick” allows relating any two kinds of model elements with one another, see also the idea of a *linguistic extension* in [\[109\]](#). Due to its generic nature, working with such trace models often becomes cumbersome because of the additional complexity w.r.t. type checking that has to be included in the definition of consistency rules, see e.g. [List. 4.6](#)

```

1 context BPMN!Activity {
2     constraint CR1 {
3         check : self.type = BPMN!ActivityType:BUSINESS_RULE implies
4             GenTrace!TraceLink
5                 .allInstances().select(t|
6                     t.connects.exists(e1|e1.represents = self) and
7                     t.connects.exists(e2|e2.isTypeOf(DMN!DecisionTable) and e2 != self)
8                 ).count() = 1
9     }
10 }

```

Listing 4.6: Global Constraints

In a multi-model, there are specific types of commonality relationships. For example, between `BusinessRuleActivities` and `DecisionTables`. Therefore, it is worthwhile to create a *domain specific* trace metamodel, see e.g. [\[168, 413\]](#). This idea is sketched in [Fig. 4.18](#). This figure also shows the conformance-relationships between the involved

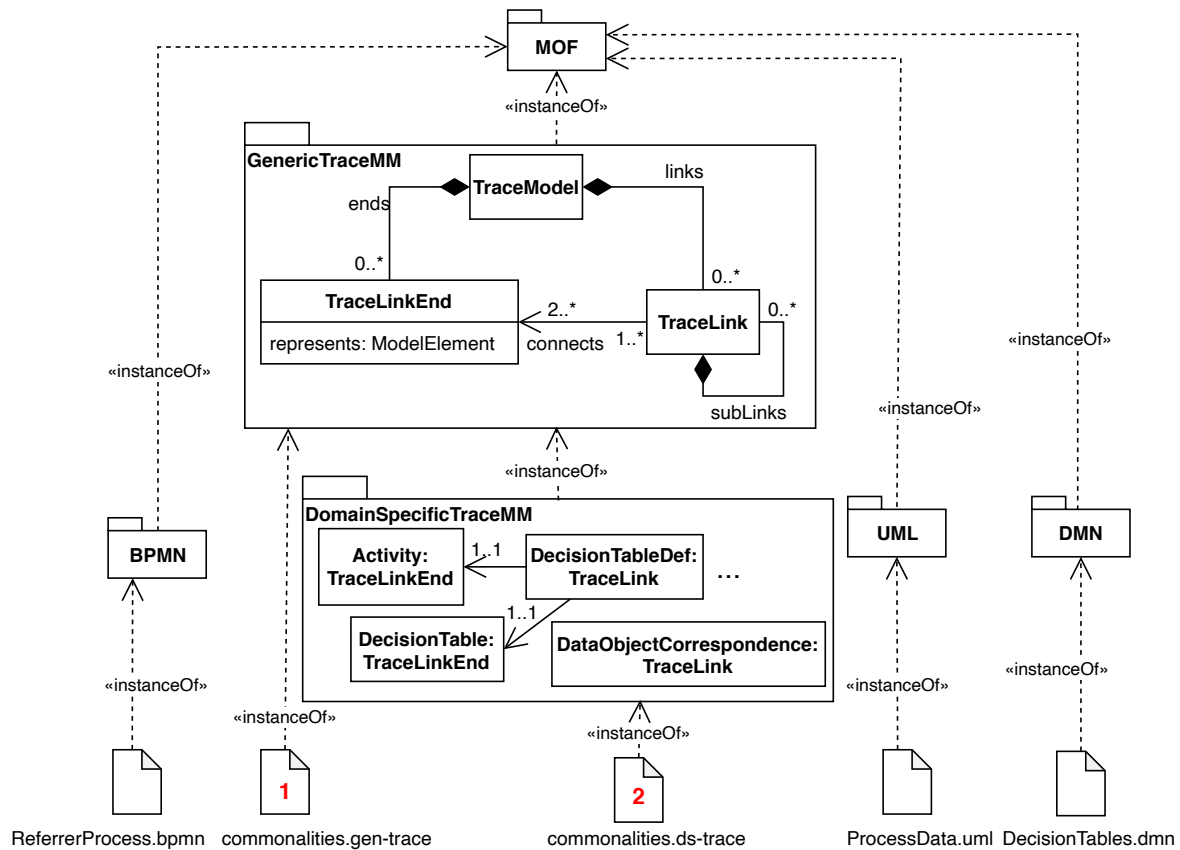


Fig. 4.18: Representing Trace Models

models and metamodels. Metamodels are depicted as packages and models as files at the bottom. Notice the double nature of the generic trace-metamodel: It can either be instantiated directly by a generic trace-model (1) or serve as the metamodel for a domain specific trace-metamodel, which is instantiated by a domain specific trace model (2). The domain specific trace metamodel is a suitable carrier for the definition of consistency rules such that global consistency verification can be executed on a domain specific trace model.

4.4.3 eMoflon

The third tool *Emoflon*⁹ [482] is taken as a representative for the class of *graph transformation (GT)* tools [146]. It is based on the concept of a *Graph Grammar (GG)* [403]. The latter is given by a set of (production) rules together with a start graph, which induces a *language*, i.e. the set of all graphs producible by applying a sequence of production rules on the start graph. Hence, the definition of a graph grammar can be interpreted as a consistency rule, see Sec. 4.2.6. There are several tool implementations based on this formalism: eMoflon, MoTe¹⁰ [193], AtoM¹¹ [111], Henshin¹² [48], Viatra¹³ [472], see also [452].

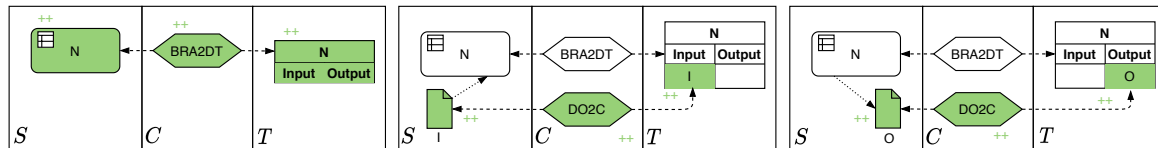
⁹<https://emoflon.org/>

¹⁰<https://www.hpi.uni-potsdam.de/giese/public/mdelab/>

¹¹<http://atom3.cs.mcgill.ca/index.html>

¹²<https://www.eclipse.org/henshin/>

¹³<https://www.eclipse.org/viatra/>



(a) Business Rule Activities correspond to Decision Tables (b) Consumed data object correspond to input side columns (c) Produced data object correspond to output side columns

Fig. 4.19: TGG production rules

I chose Emoflon in particular because it is, aside from MoTe, the only tool that fully supports TGGs, see Sec. 3.1.11. They were introduced by Schürr [422] and combine the concept of graph grammars with the notion of pair grammars [392]. A TGG can be considered as a graph grammar, which trades ordinary graphs for *triple graphs*. A triple graph is formally given by a pair of graphs¹⁴ (S, T) connected by a “correspondence graph” C that relates S and T via “graph homomorphisms”, resulting in a “span” $S \leftarrow C \rightarrow T$. In a TGG, the production rules are generally *monotonic*, i.e. they only “add” elements to an existing context. In this way, a TGG describes how two graph structures evolve together consistently.

Fig. 4.19 depicts three exemplary triple graph production rules in an integrated manner using concrete syntax. These rules implement the semantics of CR5 and parts of CR8. The Elements highlighted in green are meant to be added while the uncoloured elements represent existing context. TGGs induce a binary consistency relation between two domains of graph structures: A pair of graphs S and T is considered consistent when the language generated by the TGG contains a triple graph with S and T as its source and target (the middle part C is usually irrelevant)

The most important feature of TGGs is that they allow to “operationalise” the declarative production rules of the grammar. This means that special programs for (incremental) model transformation [144, 194], model matching [145], consistency verification [312] and update synchronisation [229, 230] can automatically derived from a TGG. The procedure has been described in [230].

Emoflon implements both GGs as well as TGGs, including the operationalisation procedure. The tool has been developed in a collaboration project between the Technical University Darmstadt and the University of Paderborn as a re-implementation of the earlier meta-CASE and graph transformation tool MOFLON [14]. It builds on top of the popular Eclipse Modeling Framework and is available as an Eclipse plugin.

Fig. 4.20 presents a screenshot of the Eclipse editor with the Emoflon plugin installed. The main feature is a textual editor in which TGG rules are defined in a respective DSL, called *eMoflon Specification Language (.msl)*. The plugin offers a generic read-only visualisation for TGG rules utilising *PlantUML*¹⁵ (the screenshot depicts the rule Fig. 4.19b). The tool generates Java programs for global consistency verification, model translation and update propagation from the textual rule definitions. These programs are based on the graph transformation concept and the graph database *Neo4J*¹⁶ is utilised to perform pattern matching.

¹⁴Often named *source* and *target* due to historic reasons.

¹⁵<https://plantuml.com/>

¹⁶<https://neo4j.com/>

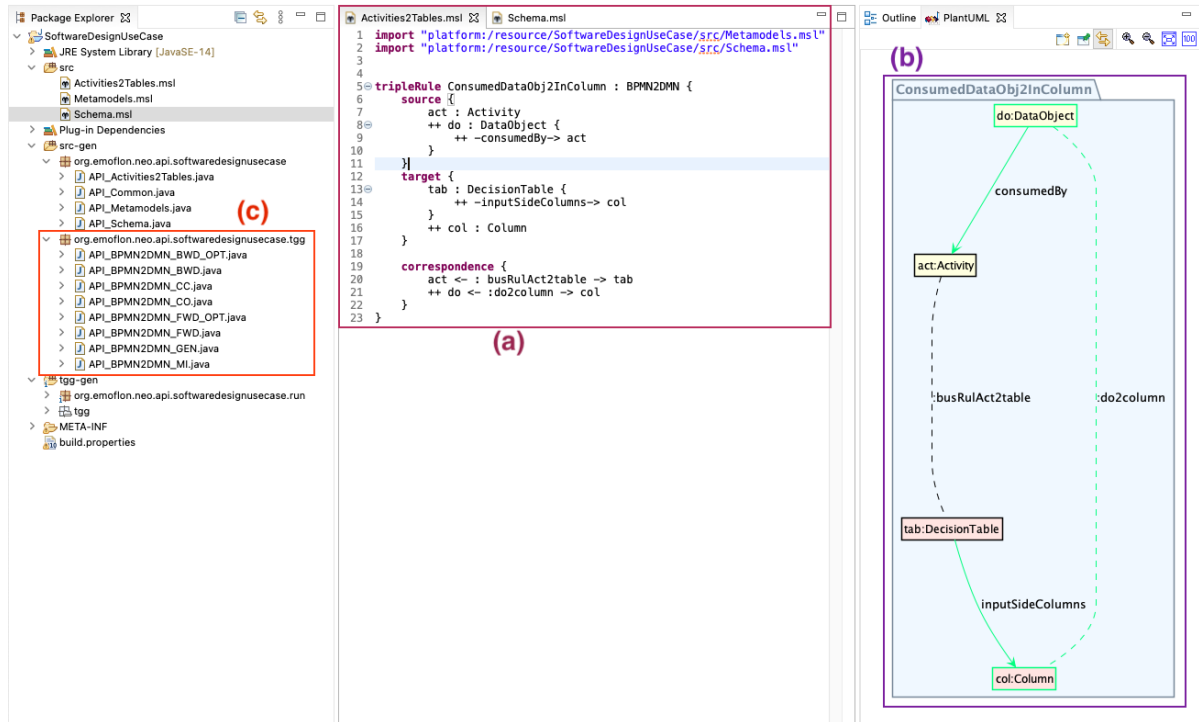


Fig. 4.20: Screenshot the Eclipse Emoflon plugin: (a) textual editor (b) visualisation (c) generated programs

4.5 Summary & Identified Limitations

In this chapter, I addressed all four research questions by summarising the state of the art, i.e. (RQ₁) analysing how models, multi-models and consistency rules are represented both technically and formally (Sec. 4.2.1–Sec. 4.2.6), (RQ₂) analysing how consistency verification is implemented (Sec. 4.2.7), and (RQ₃) how consistency restoration is implemented (Sec. 4.2.8). By looking into three particular tools (Sec. 4.4) that implement multi-model consistency management I analysed how the above concepts are orchestrated in a comprehensive architecture (RQ₄).

The latter identified one major limitation: All three tools consider binary correspondence relations only. Echo only supports binary QVTr consistency rules, the match and the merge operators in Epsilon have exactly two arguments, and the definition of TGGs is binary by design.

Binary inter-model consistency rules alone are generally not sufficient to realise all kinds of global consistency requirements, recalling the discussion in Sec. 3.3.5. I want to illustrate this on the concrete example of the consistency rules in Fig. 1.9. Consider Fig. 4.21 depicting a BPMN model (A^1), an UML model (A^2) and a DMN model (A^3). Looking at the pairs (A^1, A^2) and (A^2, A^3), the consistency rules CR₆ and CR₇ are apparently satisfied. But when it comes to CR₈, all pairings have to be considered together and looking at the example in Fig. 4.21 there is an inconsistency, which cannot be discovered by binary considerations.

Hence, Echo and Emoflon cannot realise CR₈ without further ado. Admittedly, Trollmann and Albayrak have proposed a generalisation of triple graphs [463, 464] called *graph diagrams* to cope with multi-ary correspondence relations. These diagrams,

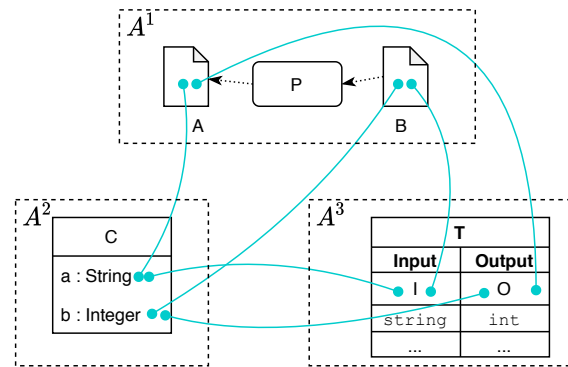


Fig. 4.21: Insufficient Ternary Relation Representation

however, are *fixed* and require that all possible types of correspondences and consistency rules are known beforehand. In Epsilon, the situation can be circumvented by iterated applications of the match and merge operator, i.e. merging A^3 with A^{12} , where A^{12} results from merging A^1 with A^2 . This, however, cause a new issue: The concrete EML-programs have to ensure that the merge happens in an *associative* and *commutative* manner [275], otherwise the order merging steps would affect the final result.

Another general limitation of contemporary multi-model consistency management approaches is their often reported limited interoperability [138, 461, 489]. For instance, the three tools presented in Sec. 4.4 have in common that they can read and write metamodels and models in the XMI format. But, it is not possibly to interchange their technical renditions of the consistency rules. Also commonalities are treated differently by each tool such that matches identified by an Epsilon ECL program cannot be reused in Emoflon or Echo. There are more reports about other multi-model consistency management approaches, each providing their own domain specific or generic trace models [168, 413].

Lack of interoperability is often associated with a lack of a common formal foundation [127]. Hence, in the following chapters, I want to develop my own solution which seeks to address the two limitations by developing a formal underpinning for multi-model consistency management supporting multi-ary correspondence relations.

“Mathematicians are like Frenchmen: whatever you say to them they translate into their own language and forthwith it is something entirely different.”

—Johann Wolfgang von Goethe

CHAPTER 5

FORMALISATION

Following the conceptualisation of the problem domain in Chap. 3 and a survey of existing solutions and in Chap. 4, which identified a range of limitations with current approaches, I will develop my own solution to the multi-model consistency management problem. My goal is to address these limitations (support for multi-ary relationships, lack of formal foundations, issues with merged artefacts, compare Sec. 4.5). I will begin at the *formal* level, i.e. utilising in the language of *mathematics*. This has the advantage that the solution is *technology-independent* and has a *univocal* interpretation.

The structure of this chapter is aligned with the three research questions: Sec. 5.1 addresses **RQ1** (multi-model representation), Sec. 5.2 addresses **RQ2** (consistency verification), and Sec. 5.3 addresses **RQ3** (consistency restoration). The main scientific contribution presented of this chapter are *comprehensive systems*, a novel formalism for representing and reasoning about collections of inter-related software models. Comprehensive Systems had recently been introduced in a series of publications at various venues: [302, 446, 448, 449]. A precursor to comprehensive systems is the “verification via merging”-approach, which is based on a slightly different idea (colimit) and which was investigated in [134, 294, 295, 445].

My formalism builds on *generalised sketches* [117, 132] a.k.a. *diagrammatic predicate graphs* [407, 409], which are formulated by means of *category theory*. Consequentially, the presentation of this chapter employs category theory as well. Category theory has been successfully applied in several domain of Computer Science and Software Engineering [130, 171, 203]. Using category theory offers several benefits for my work such as concise and abstract definitions, a graphical but formal syntax, means for comparing heterogeneous mathematical structures and last but not least theorems that support correctness proofs. At the same time, it is not as commonplace as set theory and logic. For those readers, who are less familiar with this special branch of mathematics, Appendix C contains a short introduction into the subject and further references to literature.

A FEW WORDS ON NOTATION By convention, variable names for categories use a double-struck font \mathbb{C} . The class of objects in a category \mathbb{C} is denoted by $|\mathbb{C}|$ and the class of *all* morphisms in \mathbb{C} is denoted by \mathbb{C}^{\rightarrow} . The hom-set for $A, B \in |\mathbb{C}|$ is denoted $\mathbb{C}(A, B)$. The elements of a hom-set are called morphisms or arrows. For objects, usually, upper-case letters (A, B, C, \dots) are used while morphisms are usually denoted by lower-case letters

(f, g, h, \dots). Morphisms together with their domain and codomain are denoted in a “function-definition style” $f : A \rightarrow B$ or “diagrammatic style” $A \xrightarrow{f} B$. Composition of $f : A \rightarrow B$ and $g : B \rightarrow C$ is denoted $g \circ f$. The category of sets and functions is denoted by **Set**, the total order¹ of natural numbers by **Nat**, the category of *small* categories by **Cat**, and the “category of categories”² by **CAT**. For $\mathbb{C}, \mathbb{D} \in |\mathbf{Cat}|$, the category of functors with domain \mathbb{D} and codomain \mathbb{C} and natural transformations between them is denoted by $\mathbb{C}^{\mathbb{D}}$. Should the terminology of this paragraph sound unfamiliar, a look into Appendix C is advised.

5.1 Representation

In the first step all model management concepts from Chap. 3 require a formal interpretation, namely models, conformance, change, and correspondence. Software models represent (domain) knowledge abstractly and therefore formal software modeling is closely related to *mathematical logic*. Traditionally, the main tools are *set theory*, *propositional logic* and *universal algebra*. More recently *type theory* and *category theory* have become more important in this area and are able to formally represent the same concepts. Fig. 4.2 summarises various approaches for formalising models and related artefacts: Either by defining a set-theoretical *system model* interpretation of modeling artefacts [65, 66, 214], by interpreting models as an *algebraic specification* [58, 59], by translating models into *sentences* of a suitable logic [327, 328], by considering models as *graphs* [48, 398], or by investigating them on the abstract level of *category theory* [118, 252, 254, 257].

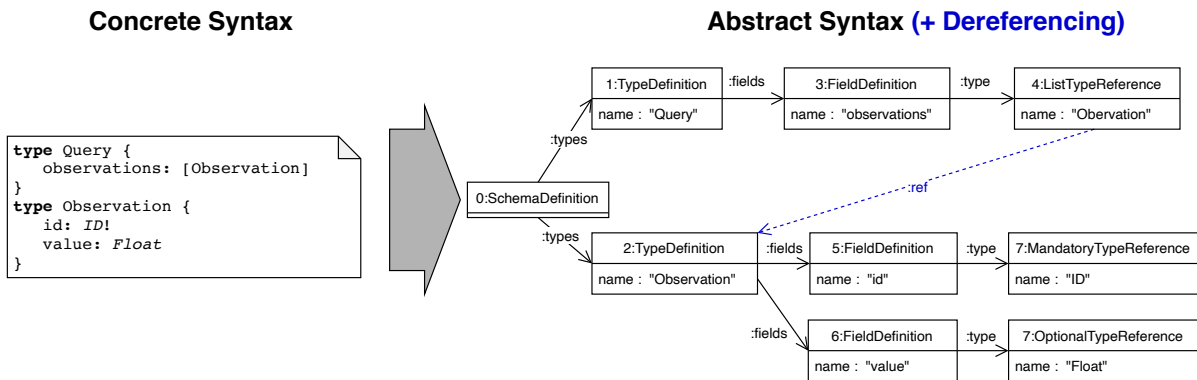
5.1.1 Formalising Models

My approach follows in the footsteps of both graph- and category-based approaches. Concretely, I am building my formalism on *generalised sketches* over *graph-like structures*. The term “generalised” means that these sketches generalise the classic notion of “Ehresmann sketches” [39]. A sketch can be seen as the categorical equivalent of a theory, i.e. a syntactical description of a class of mathematical objects. Their utility for data modeling has been recognised by several researchers [252, 314, 381]. Nonetheless, regular sketches require to introduce multiple auxiliary objects in order to express the desired properties. During the 90’s, Diskin and Makkai independently of each other developed the same generalisation of Ehresmann sketches that avoids a proliferation of auxiliary structures. Their discovery took place in different scientific contexts. While Diskin was working on concrete (software) engineering problems [117], Makkai was working on an abstract approach to logic [332]. Later, generalised sketches have been presented to the Software Engineering community [132] and the MDSE community [407–409]. In the latter case, sketches were “rebranded” under the name “*Diagram Predicate Framework (DPF)*”.

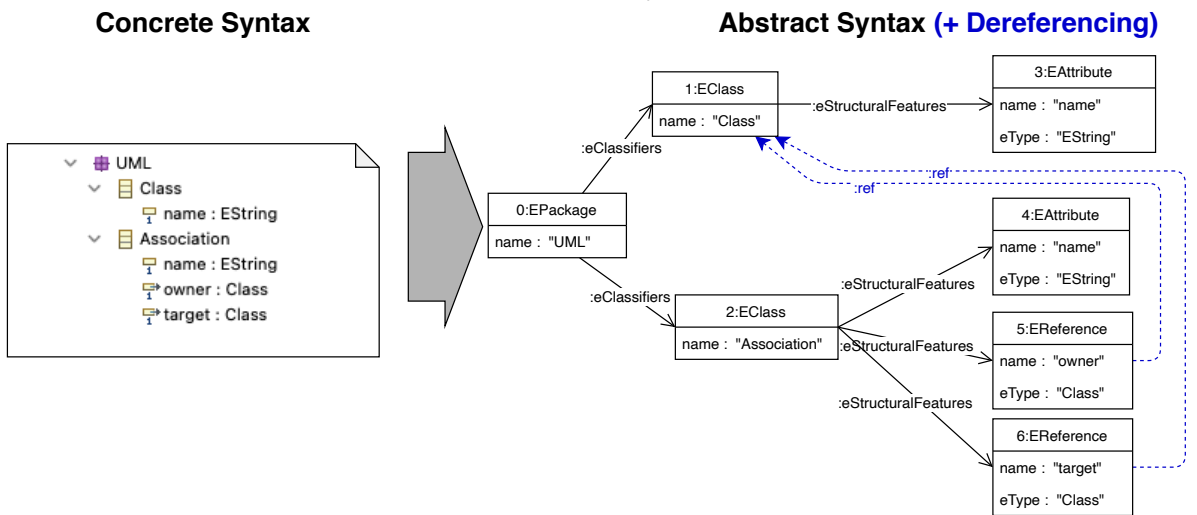
The foundation of generalised sketches are *graphs* or more generally *graph-like structure*. Every software model can formally be interpreted as a graph-like structure.

¹Every total order is a (rather simple) codiscrete category.

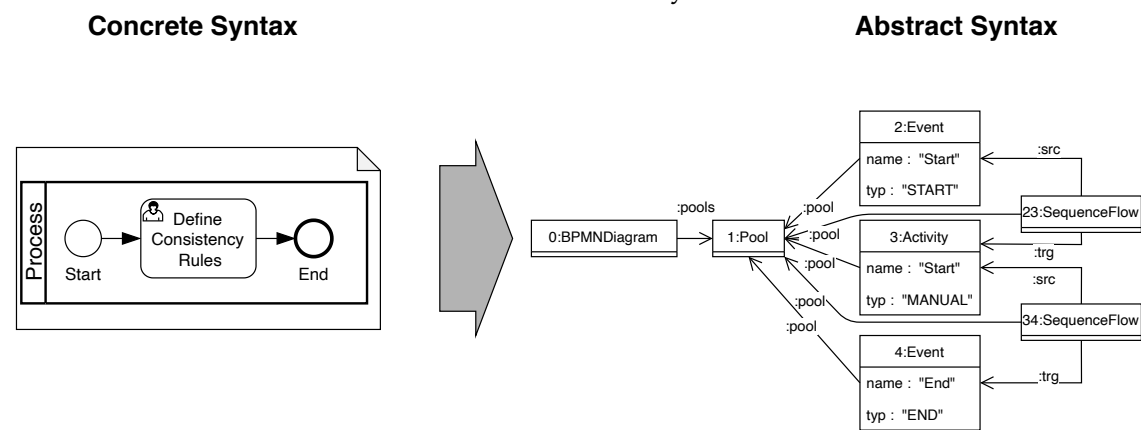
²To avoid the typical logical paradox, I refer to the concept of *Grothendieck universes*. Note also the usage of the word “class” instead of “set”.



(a) Textual Syntax



(b) Tree-based Syntax



(c) Graphical Syntax

Fig. 5.1: Syntax Abstraction

Let me motivate this idea with some examples: Fig. 5.1 depicts three types of concrete syntaxes: *textual* (e.g. the GraphQL schema definition language; Fig. 5.1a), *tree-based* (e.g. Ecore and other XML-based formats; Fig. 5.1b), and *graphical* (e.g. BPMN; Fig. 5.1c). When we apply another abstraction step and “forget” the concrete syntax, we get the *abstract syntax graph*³ of these models (the right hand sides in Fig. 5.1).

Graphs are a “lingua franca” in Computer Science and Software Engineering [403] for depicting structured information. Hence, there is a proliferation of “graph languages”. Fig. 5.2 depicts instances of common graph languages:

Directed multigraphs (Fig. 5.2a) are one of the most simple form. They comprise vertices (blue) and edges (red). Edges have an owner vertex and a target vertex (highlighted by the arrow-tip), hence the adjective “directed”. Both vertices and edges have an identity such that there can be multiple parallel edges between a pair of vertices, hence the adjective “multi-”. Directed multigraphs can be used to formally represent flow charts, state charts and many more structures encountered in SE.

Bipartite place-transition nets (Fig. 5.2b) distinguish between two types of vertices (places and transitions) and are the backbone of petri nets, which is a prevalent formalism for the analysis of concurrent software systems [148].

E-graphs [151] (Fig. 5.2c) or property graphs [186], extend the notion of directed graphs by distinguishing two types of vertices: graph nodes and data nodes, which represent (complex) objects and (simple) values respectively. Therefore, there are three types of edges: (1) those between graph nodes (called graph edges), (2) those between graph nodes and data nodes (called node attribute edges), and (3) those between graph edges and data nodes (called edge attribute edges). The latter two can be interpreted as node and edge attributes and are depicted as compartments. Hence, the depiction of E-graphs is very similar to those of UML class and object diagrams.

Finally, hypergraphs [55, 386] (Fig. 5.2d) allow edges to connect more than just two vertices. In the figure, undirected hyperedges are depicted as coloured areas. Hypergraphs have multiple applications in computer science, e.g. machine learning, satisfiability problems etc. Moreover, terms can be represented with the help of hypergraphs and an algebra can be considered as special kind of hypergraph.

All of these examples have in common that they behave essentially like *sets*. Theoretically, this has been captured by the concepts of *presheaf topoi* [207].

Definition 5.1 Presheaf

Let \mathbb{B} be a small category (Def. C.1). A functor^a $G : \mathbb{B} \rightarrow \mathbf{Set}$ (Def. C.5) from \mathbb{B} into the category of sets and functions \mathbf{Set} (Fact 20) is called a *presheaf*.

^aHistorically, a presheaf is defined as a contravariant functor $G : \mathbb{B}^{\text{op}} \rightarrow \mathbf{Set}$ due to its application in topological spaces. For my investigations, the covariant version is better suited. Covariant and contravariant version provide the same properties.

A presheaf G “interprets” every (sort) object $S \in |\mathbb{B}|$ as a (carrier) set $G(S)$ and every morphism⁴ $\text{op} : S \rightarrow S' \in \mathbb{B}$ as a (operation) mapping $G(\text{op}) : G(S) \rightarrow G(S')$. In the

³In most cases, abstract syntax trees actually turn out to be graphs as they contain cross-references established via identifiers (blue-coloured elements in Fig. 5.1)

⁴The abbreviation “op” for morphisms in \mathbb{B} shall indicate that \mathbb{B} -arrows are certain operations constituting the structure of the base language, such as *source* and *target* operations of edges in directed

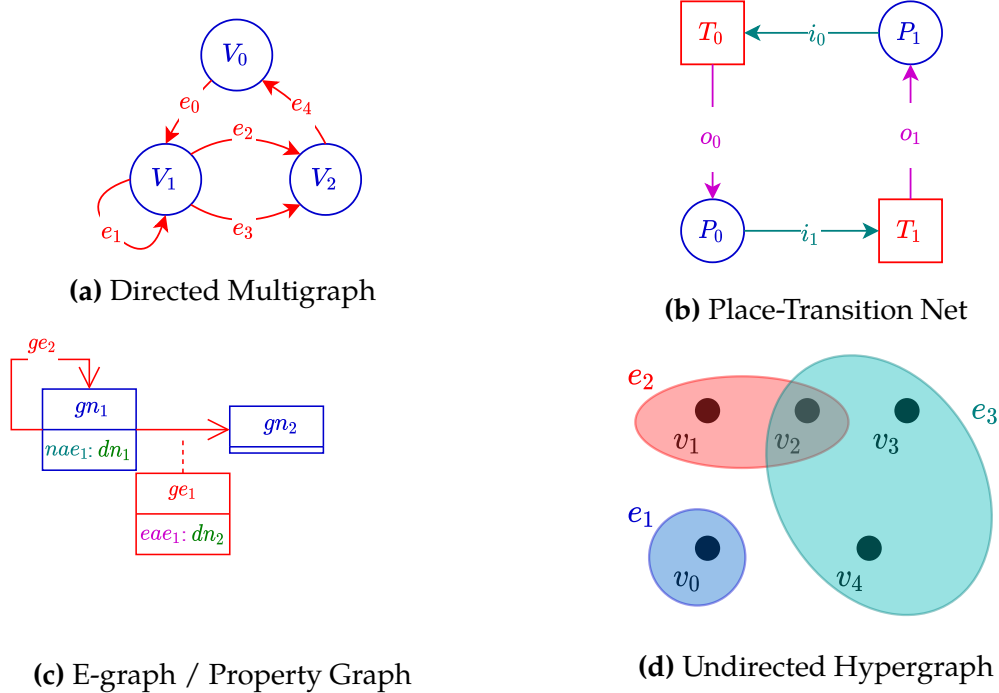


Fig. 5.2: Instance of Graph Languages

following, I will often use the alternative compact notation S_G for denoting carrier sets and $op^G : S_G \rightarrow S_G$ for denoting operations of a presheaf G . A small category \mathbb{B} can also be interpreted as a signature with unary operation symbols only by interpreting every object as a sort and every non-identity morphism as a unary operation. As an example, List. 5.1 shows the algebraic signature Σ_{DG} for the schema category \mathbb{B}_{DG} of directed multigraphs, shown in Fig. 5.3a.

```

 $\Sigma_{DG} ::=$ 
sorts: V(ertices), (E)dges
opns: s(ource), t(arget): E --> V
    
```

Listing 5.1: Signature Σ_{DG}

Fig. 5.3 depicts four small categories, drawn as directed multigraphs (the identity morphisms are omitted) which represent the respective schema categories of the graph languages featured in Fig. 5.2. For each schema category, there is a respective functor category, which has presheaves as objects. Such categories have been called *graph-like structure* in [322].

multigraphs.

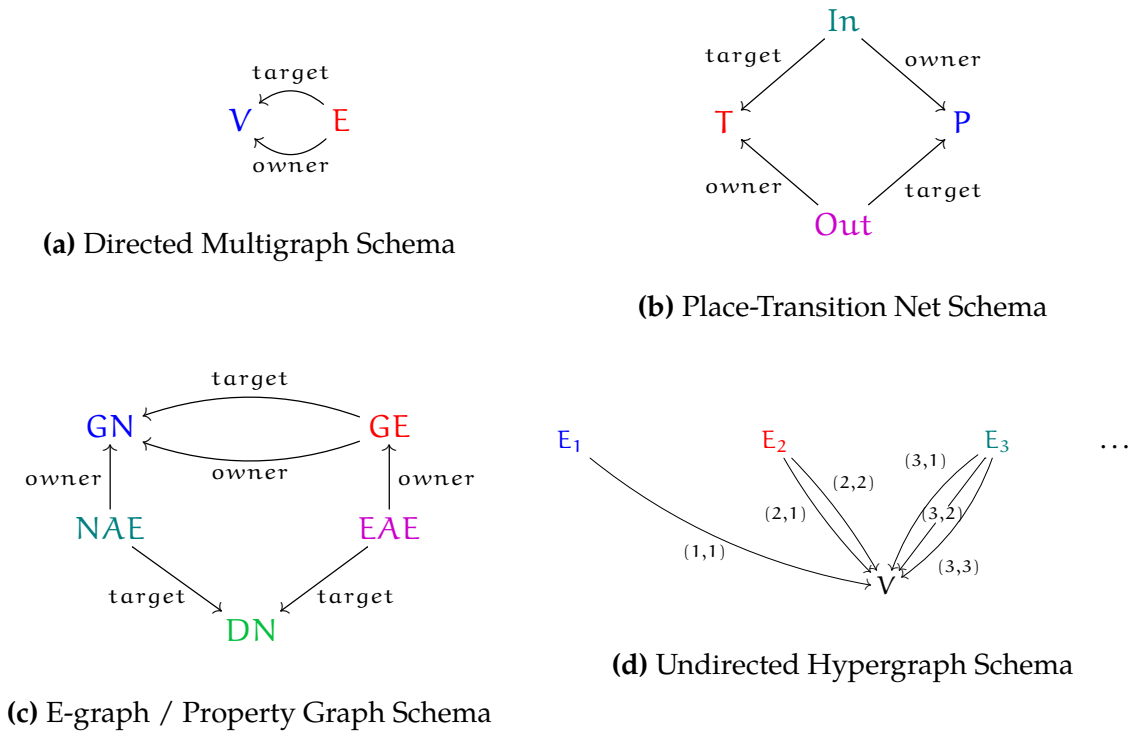


Fig. 5.3: Graph Language Schemas

Definition 5.2 Base Language \mathbb{B} and graph-like structures \mathbb{G}

Let \mathbb{B} be a small category called *base language*. The base language gives rise to a category of *graph-like structure* $\mathbb{G} := \mathbf{Set}^{\mathbb{B}}$ where

- Objects $|\mathbb{G}|$ are given by the class of all presheaves Def. 5.1 with domain \mathbb{B} .
- The hom-set $\mathbb{G}(G, H)$ for two objects $G, H \in |\mathbb{G}|$ is given by the class of all natural transformations $f : G \Rightarrow H$ (Def. C.6), i.e. $|\mathbb{B}|$ -indexed families of mappings (**Set**-morphisms) $f := (f_s : G(s) \rightarrow H(s))_{s \in |\mathbb{B}|}$ ensuring that the following diagram commutes

$$\begin{array}{ccc}
 G(s) & \xrightarrow{f_s} & H(s) \\
 G(\text{op}) \downarrow & & \downarrow H(\text{op}) \\
 G(s') & \xrightarrow{f_{s'}} & H(s)
 \end{array} \tag{5.1}$$

for each $\text{op} : s \rightarrow s' \in \mathbb{B} \rightarrow$

- Composition is defined as component-wise function composition, i.e. for two \mathbb{G} -morphisms $f : G \rightarrow H$ and $g : H \rightarrow K$, their composition $g \circ f := (g_s \circ f_s)_{s \in |\mathbb{B}|}$ is defined via composition in **Set**.
- Identities are families of **Set**-identities, i.e. $\text{id}_G : G \rightarrow G := (\text{id}_{G(s)})_{s \in |\mathbb{B}|}$.

For the remainder of this thesis, I will fix an adequate base language \mathbb{B} . The choice of this base language is arbitrary but for practical examples based on the scenarios in Sec. 1.3, E-graphs are considered as a suitable base language because they are similar to class and object diagrams. In the spirit of the terminology in [322], I will overload the term *graph* and call objects in a category of graph-like structure \mathbb{G} “graphs” and morphisms in \mathbb{G} “graph morphisms”, bearing in mind the more general setting. It is important to note that objects of \mathbb{G} are sufficiently concrete such that one can talk about *elements*: For a graph $G \in |\mathbb{G}|$, saying $x \in G$ means that there is some object $S \in |\mathbb{B}|$ such that $x \in G(S)$. If the respective sort object $s \in |\mathbb{B}|$ has no other outgoing morphism than the identity morphism, an element $x \in G(S)$ is called a “node” otherwise it is called a “(hyper-)edge”. Simultaneously, “ $\forall x \in G$ ” means “for all x of any sort S in the carrier set of G ”. This allows to consider “subgraphs” $F \subseteq G$, given by sort-wise subset relations. Categorically this is represented by an inclusion morphism $F \hookrightarrow G$, which is a special monomorphism. In general, \mathbb{G} objects behave as sets from an abstract point of view. This is due to the fact that every category of presheaves is a *topos*:

Definition 5.3 Topos

A category \mathbb{C} is called a *topos* if and only if

- \mathbb{C} has all *limits* and *colimits* (Appendix C.3),
- \mathbb{C} is *cartesian-closed* (Appendix C.3),
- and \mathbb{C} has a *subobject classifier* (Appendix C.3).

Fact 1 Presheaf Topos [207]

Every category of presheaves $\mathbf{Set}^{\mathbb{B}}$ (i.e. \mathbb{G}) a topos, see [207].

Remark 5.1 Simple Graphs

The reader may ask: “What about *simple graphs*?” The attribute “simple” refers to those graphs, where the edges form a relation among the set of vertices, i.e. a simple graph G is a tuple $G := (V, E \subseteq V \times V)$ with V being the set of vertices and E being the edge-relation among vertices. Thus, there cannot be any parallel edges between a pair of vertices. Simple graphs cannot be expressed by a set-valued functor alone (they require additional constraints) and do not form a topos [5]. Therefore, they are not considered further here.

Remark 5.2 Object Identifiers (OIDs) and “up to isomorphism”

A major difference between set theory and category theory is that “names” are not important. Most definitions and constructions in category theory are only determined “up to isomorphism”. In the context of object-oriented programming, this notion is quite natural: Object instances in a running program are internally identified via memory addresses. These addresses may vary between each

execution of the program. Hence, their concrete names are also not important.

5.1.2 Formalising Conformance

The internal structure of a model is not arbitrary, it must conform to the metamodel of a respective modeling language. The definition of the latter comprises the notions of *abstract syntax* and *structural integrity rules*. These aspects are modelled by graph-like structure and diagrammatic constraints, which will be explained below.

The example scenario in Sec. 1.3.2 features several modeling languages, e.g. BPMN, UML, DMN etc. Most of these languages are defined using OMG’s MOF [368], which is essentially a subset of the UML class diagram language comprising classes, attributes and references. For instance, Fig. 5.4 depicts a simplified variant of the metamodel of the BPMN language (ignore the blue colour for the moment). The clouds allude to the concrete syntax presentation of model elements. Metamodels of the other modeling languages from Sec. 1.3.2 can be represented in the same way and also the schemas of the information systems from the integration scenario in Sec. 1.3.1 can be expressed in this syntax, see Fig. 1.6.

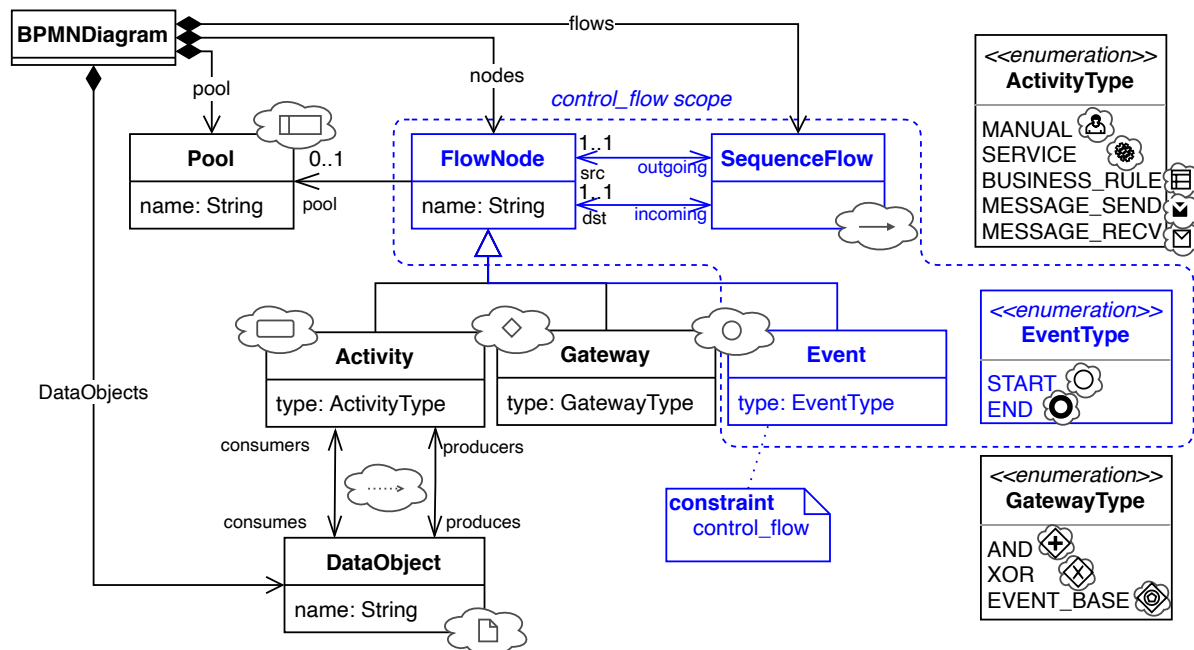


Fig. 5.4: Simplified BPMN metamodel

The drawing in Fig. 5.4 effectively shows an (E-)graph decorated with additional visual elements such as *inheritance* (arrows with an open triangle arrow tip \rightarrow), *composition*-arrows ($\blacklozenge \rightarrow$) and *multiplicity*-labels (1..1 or 0..1). The underlying graph represents the language concepts together with their structural relations, while the new visual elements come with a special semantic interpretation.

First, there is inheritance: UML/MOF class diagrams and other modeling languages comprise this object-oriented concept. It is an important feature in object-oriented design, which is used to “factor out” common features and behaviour of otherwise

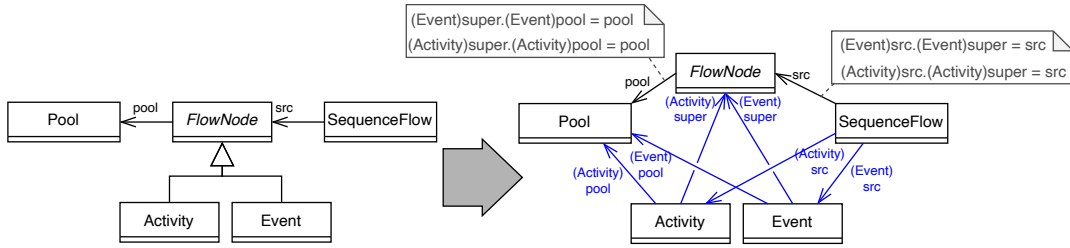


Fig. 5.5: “Desugaring” inheritance arrows

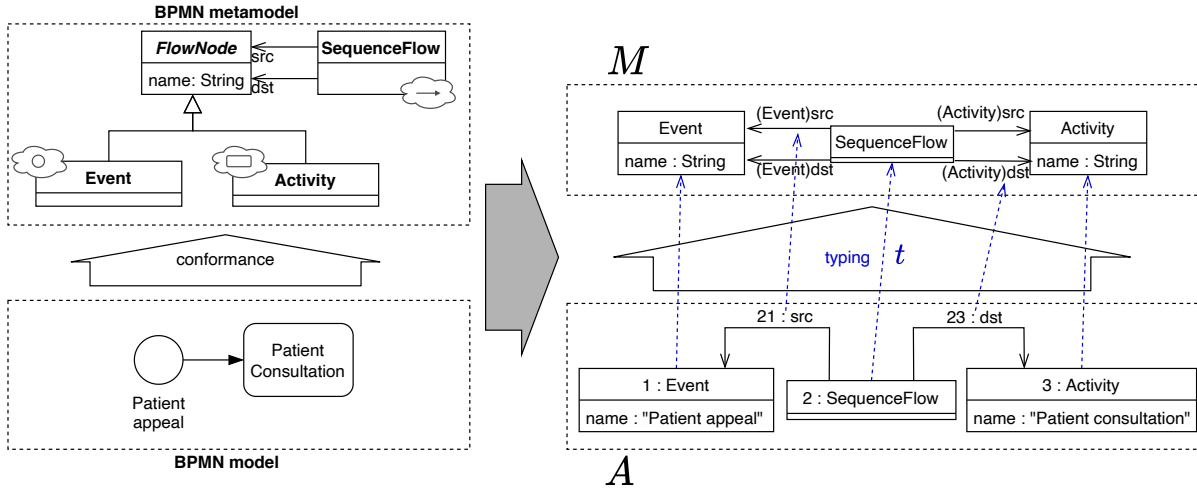


Fig. 5.6: Conformance via Typing

heterogeneous objects. Researchers have developed different approaches to formally describe this phenomenon, e.g. *clan-morphisms* [228] or morphisms with *inequality* [325]. For the scope of this thesis and all metamodel examples featured in it, it will be sufficient to simply consider inheritance as a kind of “syntactic sugar”. This means that abstract supertypes are used to group a shared set of in- or outgoing (graph or attribute) edges, which are “multiplied out” for the concrete formal representation. The original inheritance is retained as a regular edge (named “super”) such that it may be used for the definition of consistency rules later. Furthermore, “commutativity”-requirements subject to the new super-edges arise. Fig. 5.5 depicts an example of the overall translation, where the above requirement is expressed via four equations that are contained in the graphical “note” elements.

The latter equations together with the other visual elements (compositions and multiplicities) encode *constraints*, i.e. concrete syntactical conditions that the instance models must adhere to. I will present their formal interpretation shortly. But first, let me analyse the “semantics” of the abstract syntax graph alone, i.e. without further constraints attached to it. Each element of a model must be typed over one of the concepts defined in this graph. The presentation in Fig. 5.1 implicitly conveyed this notion already: Note that the edges and nodes follow the naming scheme: “ $\langle element\ id \rangle : \langle type\ name \rangle$ ”. Formally, this is expressed through a \mathbb{G} -morphism $t : A \rightarrow M$ where A is a graph representing the model elements and M is the graph containing the concepts.

Fig. 5.6 demonstrates how the conformance of a BPMN-model w.r.t. the BPMN-

metamodel (left hand side) is formally interpreted as a graph morphism $t : A \rightarrow M$ (right hand side). The assignment performed by t is depicted via blue dotted arrows in Fig. 5.6. A morphism in the category of graph-like structure \mathbb{G} is a structure-preserving mapping, compare *homomorphism* in algebra. In particular, it has to preserve owner/target-incidences, see Def. 5.2. Note how this property acts as a syntactic constraint: For instance, the element with id 21 must be either mapped to `(Event)src` or `(Event)trg` because the target node with id 1 has been mapped to `Event` while the owner node has been mapped to `SequenceFlow`.

Therefore, the homomorphism property of the typing constrains the “allowed” linkage constellations of elements in a model. Yet, this property alone is generally not enough to formally represent all structural integrity rules of a modeling language. This is why UML/MOF class diagrams comprise special graphical elements such as *multiplicity* or *composition*, which further constrain the allowed constellations of elements in the instance model. For example, every element typed over `SequenceFlow` shall have exactly one `src` and exactly one `trg` `FlowNode` and all `Pool`, `FlowNode`, `SequenceFlow` and `DataObject` elements must have a unique `BPMNDiagram`-container.

A natural formalisation of such constraints is given by the framework of generalised sketches. Simply speaking a *sketch* is a (carrier) graph together with a set of *diagrams* attached to it. The term diagram in Category Theory does not mean “a drawing” (as in SE) but “a selection of elements” of the carrier graph. Each diagram has an associated formal interpretation. In classical Ehresmann sketches, a diagram either specifies a *limit* or a *colimit*. It can be shown that this formalism is just as expressive as first order theories [4] but it generally requires the introduction of multiple auxiliary elements, which makes it cumbersome to work with [494]. Generalised sketches abstract away from the concrete meaning of a diagram, i.e. each diagram is indexed by a *predicate* providing the abstract semantics. These predicates are organised in a signature.

Definition 5.4 Diagram Predicate Signatur

A *diagram predicate signature* $\Pi = (|\Pi|, \text{ar})$ is a pair, consisting of a set of predicate symbols $|\Pi|$ and a function $\text{ar} : |\Pi| \rightarrow |\mathbb{G}|$ called *arity*, which assigns a graph to each predicate symbol.

Thus, each predicate is identified by a name and comes with an arity graph, which defines its *shape*⁵. For the running example of UML/MOF class diagrams, I further add a *graphical visualisation* for each predicate symbol. This allows highlighting the diagrams in a more natural way by re-using existing concrete syntax Fig. 5.4. Tab. 5.1 shows a suitable predicate signature for the running example.

Each of the predicate symbols in Tab. 5.1 embodies a specific constraint, e.g. that an element typed over a concept with the respective feature always shall have an outgoing link (mandatory). Semantics are defined (in a fibred way) by considering suitable sub-classes of typed structures, which relates to the `instanceOf`-relationship between models and metamodels in MOF. Categorically, it is given by the concept of

⁵In Set-based predicate logic, the arity of a predicate is simply a natural number. For instance, the predicate `lessOrEqual` (modeling \leq) has the arity 2. In my case, I am working with graph-like structure. But, the Set-based case can be represented in this more general setting by interpreting each natural number as a discrete graph.

Name p	Arity $\alpha(p)$	Visualization
composition	$0 \xrightarrow{01} 1$	
optional	$0 \xrightarrow{01} 1$	
mandatory	$0 \xrightarrow{01} 1$	
inverse	$0 \xrightleftharpoons[10]{01} 1$	
commute	$0 \xrightarrow[01]{02} 1 \xrightarrow[12]{} 2$	

Table 5.1: Predicate Signature for class diagrams

slice category:

Definition 5.5 Slice category $\mathbb{C} \downarrow C$

Let \mathbb{C} be a category and $C \in \mathbb{C}$ be an object thereof. There is a slice category $\mathbb{C} \downarrow C$ over C where

- Objects $|\mathbb{C} \downarrow C|$ are \mathbb{C} -morphisms $f : A \rightarrow C$ whose codomain is C ,
- The hom-set $\mathbb{C} \downarrow C(f, g)$ for $f : A \rightarrow C, g : B \rightarrow C \in |\mathbb{C} \downarrow C|$ is given by those \mathbb{C} -morphisms h making the following diagram commute

$$\begin{array}{ccc}
 & C & \\
 f \nearrow & & \nwarrow g \\
 A & \xrightarrow{h} & B
 \end{array} \tag{5.2}$$

- Composition and Identities are inherited from \mathbb{C} .

The semantics of a predicate “a chosen subclass of objects” taken from the slice category over the respective arity graph.

Definition 5.6 Predicate Semantics

Given a predicate signature Π and $p \in |\Pi|$ being a predicate, the semantics $\llbracket p \rrbracket$ of this predicate is a chosen subclass $\llbracket p \rrbracket \subseteq |\mathbb{G} \downarrow \text{ar}(p)|$ of the objects of the slice category over $\text{ar}(p)$, closed under isomorphisms.

Fig. 5.7 provides an illustration of Def. 5.6 for the example of the optional-predicate. The latter enforces that the edge it is imposed on shall represent a *partial function*. The collection of all graphs which can be typed over $\text{ar}(\text{optional})$ is divided into two classes, the valid and invalid ones. Note that these two classes are infinite and closed

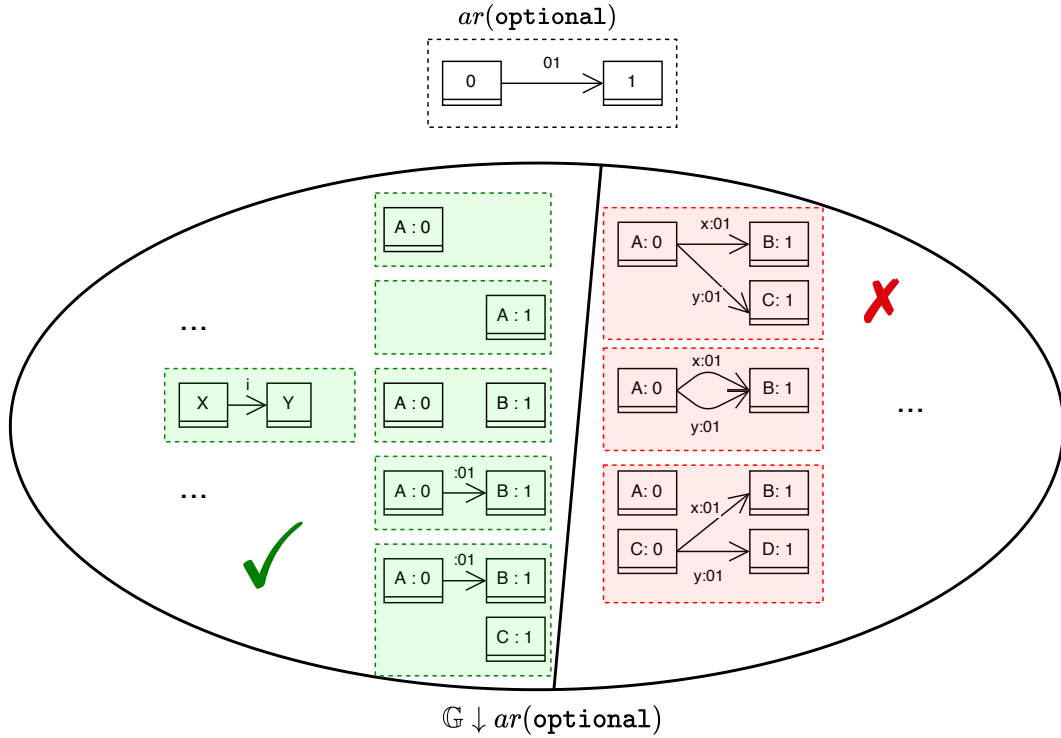


Fig. 5.7: Illustration: Predicate Semantics for optional

under isomorphisms, i.e. it does not matter how the instance elements are named. From a practical point of view, enumerating an infinite set of instances for each predicate is unfeasible. It is more practical to define predicate semantics via implementing a boolean function $check_p : |\mathbb{G} \downarrow ar(p)| \rightarrow \{\text{true}, \text{false}\}$ which is equivalent to a subset: Given a model $i : I \rightarrow ar(p) \in |\mathbb{G} \downarrow ar(p)|$ the check function applied on i returns true if and only if i belongs to the semantics ($check_p(i) = \text{true} \Leftrightarrow i \in \llbracket p \rrbracket$). We then say that i satisfies p , written $i \models p$. Alternatively, one may describe a class of instances syntactically by means of a logic theory. In the following equation, the semantics of the optional-predicate are described syntactically utilising predicate logic and assuming \mathbb{B} being the schema category of E-graphs, see Fig. 5.3c. Semantics of the other predicates in Tab. 5.1 can be defined analogously.

$$check_{\text{optional}}(i) = \text{true} \Leftrightarrow \forall e, e' \in GE_I : \text{owner}^I(e) = n = \text{owner}^I(e') \quad (5.3)$$

$$\implies e = e' \quad (5.4)$$

Remark 5.3 Default Multiplicities: UML vs. Diagrammatic Predicates

Note that interpreting the instances of a metamodel in a fibered way, i.e. as the slice category over the metamodel graph, induces a default multiplicity that differs from UML, namely $0..*$ at both owner and target sides of an edge. In UML, an attribute or association has implicitly the multiplicity $0..1$ at the target side and $0..*$ at the source side. In the class diagrams in the chapter and the remainder of the thesis, the absence of an explicit multiplicity label means $0..*$

at both ends.

It is important to note that the concrete implementation of predicate semantics is kept abstract. This allows designers to implement constraint semantics using the formalism or tool of their choice, e.g. functional programming languages, first order logic, or graph constraints [217]. Moreover, one can consider infinite predicate signatures, which comprise implementations of all possible predicates that can be expressed in languages such as EVL [288] or OCL [479]. For more references of the relationship between constraint languages and the generalised sketch framework, I refer to [493]. Constraint languages allow a designer to define a wide range of custom constraints, which exceed the expressive power of the built-in multiplicity and composition features. Fig. 5.4 exemplifies such a constraint $\phi := \text{control_flow}$, which is expressed as an OCL invariant defined in List. 5.2. This constraint requires that every `start` event must not have any incoming `SequenceFlow` [364, p. 237], whereas `end` events must not have any outgoing `SequenceFlow` [364, p. 245]. The arity of this constraint is highlighted in blue in Fig. 5.4.

```
context Event inv control_flow:
  (self.type=EventType::START implies
    self.incoming->count() = 0)
  and (self.type=EventType::END implies
    self.outgoing->count() = 0)
```

Listing 5.2: Constraint $\phi := \text{control_flow}$ formulated in OCL

Hence, *built-in* (multiplicity, composition) and *attached* (OCL,EVL) constraints are treated uniformly in this formalism. We now have collected all the ingredients to define diagrammatic constraints, which are the diagrammatic counterpart of formulas or sentences in classical (string-based) logic:

Definition 5.7 Diagrammatic Constraint

Given an graph G and a predicate signature Π , a *diagrammatic constraint* $\phi_G^\Pi := (b, p)$ on G , is given by a predicate symbol $p \in |\Pi|$ and a morphism $b : \alpha(p) \rightarrow G$, called *binding*.

To check whether an \mathfrak{S} -model satisfies a constraint $\phi = (p, b)$, one first has to “pull” the respective typing homomorphism $t : A \rightarrow |\mathfrak{S}|$ “back” along the binding homomorphism b (i.e. querying the scope) and then verify membership of the result w.r.t the semantics of p (i.e. invoking the check-function), which is illustrated in Fig. 5.8. When the binding homomorphism is injective, this “pulling-back” or querying operation simply means forgetting all parts of A , which t maps to an element outside of the scope of ϕ .

Definition 5.8 Sketch

Given a predicate signature Π , a sketch $\mathfrak{S} = (|\mathfrak{S}|, \text{Constr}(\mathfrak{S}))$ is given by a carrier graph $|\mathfrak{S}| \in |\mathbb{G}|$ and a set $\text{Constr}(\mathfrak{S})$ of diagrammatic constraints.

When predicate signature and the underlying graph are clear from the context, I will omit indices. Note that every graph can be considered as a trivial sketch with an

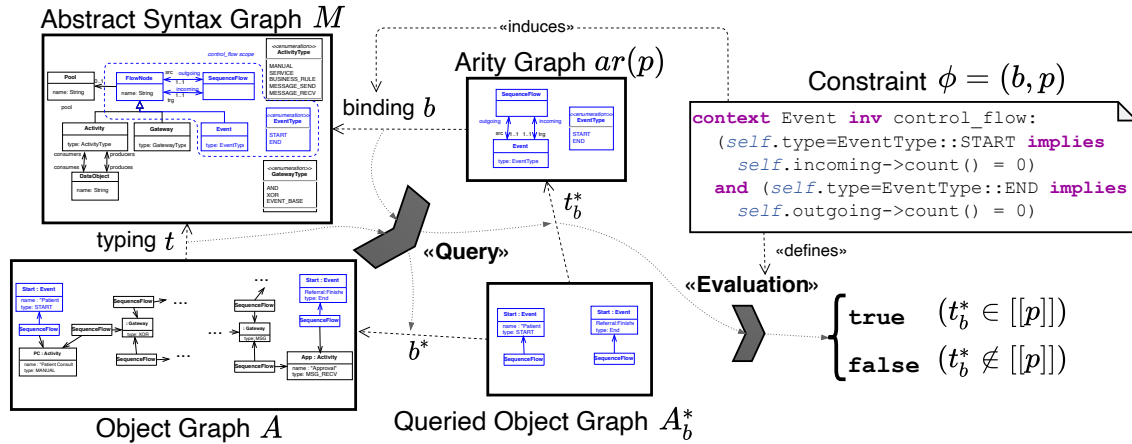


Fig. 5.8: Diagrammatic Constraints in a Nutshell

empty set of diagrams. Sketches are the diagrammatic counterpart of a “conjunctive theories” in classical (string-based) logic, i.e. they describe a set/collection of *models* called *instances*.

Definition 5.9 Instance of Sketch

Given a sketch \mathfrak{S} . An object $i \in \mathbb{G} \downarrow |\mathfrak{S}|$ of the slice category over $|\mathfrak{S}|$ is called an *instance* of $|\mathfrak{S}|$ if and only if $i^* \in [[p]]$ for all constraints $(b, p) \in \text{Constr}(\mathfrak{S})$, where $i^* : O \rightarrow ar(p)$ belongs to a chosen pullback (Def. C.13) diagram depicted on the right.

I write $i \models (b, p)$ if $i^* \in [[p]]$ and $i \models \mathfrak{S}$ if $i \models (b, p)$ for all $(b, p) \in \text{Constr}(\mathfrak{S})$.

$$\begin{array}{ccc}
 ar(p) & \xrightarrow{b} & |\mathfrak{S}| \\
 i^* \uparrow & \text{P.B.} & \uparrow i \\
 O & \xrightarrow{b^*} & I
 \end{array}$$

5.1.3 Formalising Change

So far, models are represented by means of graph-like structure \mathbb{G} , metamodels by means of (generalised) sketches and the conformance-relation via typing-morphisms and membership in the instance class of a sketch. Thus, in order to completely describe all aspects of a model space (Sec. 3.3.1), I am missing a formalisation of updates (i.e. change-relation) so far. This is going to change in this section.

Sec. 4.2.2 identified three approaches for representing changes, which are depicted schematically in Fig. 5.9. I will provide a formal interpretation of each approach in the following.

5.1.3.1 State-based = Codiscrete Categories

The State-based simply stores the state of the model *before* and *after* the update. In categorical jargon, this setting is also called *codiscrete* [258] or *chaotic* [126]. A codiscrete category \mathbb{C} is a category where every hom-set $\mathbb{C}(A, B)$ for each $A, B \in |\mathbb{C}|$ contains exactly one element. Hence, every pairing of two models (states) represents a valid

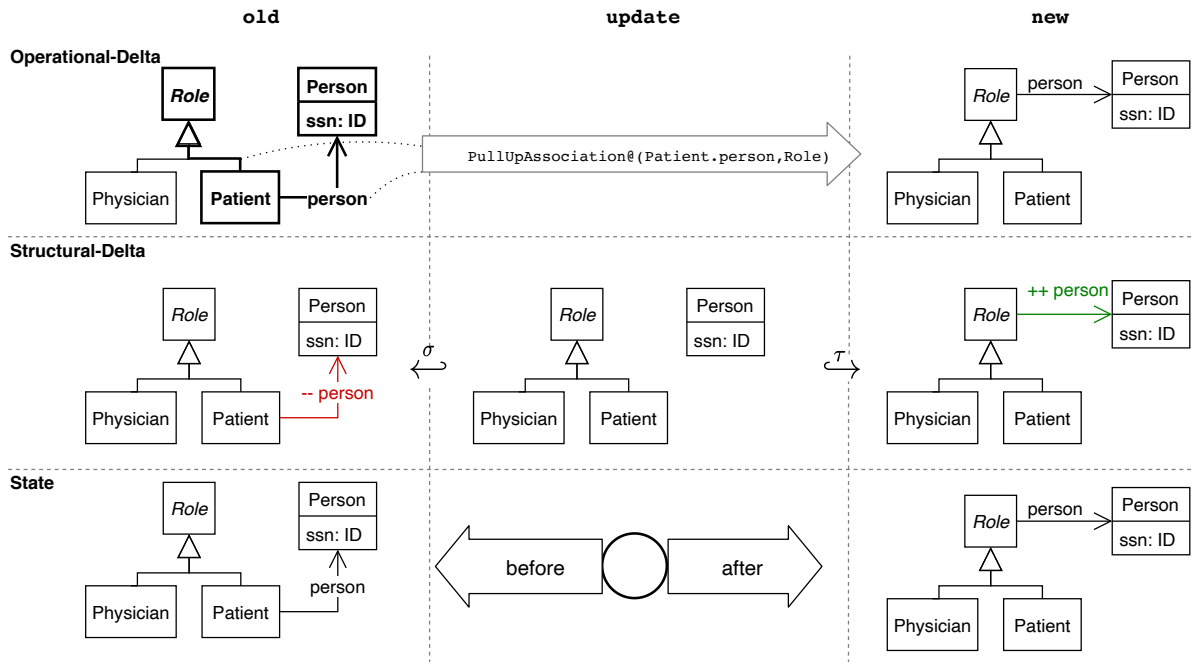


Fig. 5.9: Types of Change Representation

change. This change representation approach conveys rather limited information and provides limited opportunities for more insightful investigations.

5.1.3.2 Structural-Deltas = Span Categories



Fig. 5.10: Arrows and Composition in $\mathbf{Span}(\mathcal{C})$

Structural-deltas are strictly more expressive than the state-based variant and witness whether elements are preserved, added (see the “++”-annotation in Fig. 5.9), deleted (see the “--”-annotation in Fig. 5.9), moved, copied, merged or renamed. The formal underpinning of structural-deltas are *spans* [135]. A span is a pair of morphisms $(\sigma : X \rightarrow A, \tau : X \rightarrow B)$ sharing the same domain X. The latter object is called the *apex* of the span and the morphisms σ, τ are called *legs* of the span. From a set-theoretic point of view, a span can be interpreted as a *multi-relation*, i.e. a span can express relationships between elements of A and B but there can be multiple witnesses for a relationship between two elements $a \in A$ and $b \in B$, i.e. there are two distinct elements $x, y \in X$ ($x \neq y$) such that $\sigma(x) = a = \sigma(y) \wedge \tau(x) = b = \tau(y)$. In set-based relations, such a relationship is witnessed by exactly one element: the tuple (a, b) . For every

Formalisation

category \mathbb{C} that possesses *chosen* pullbacks⁶ there is a span category $\mathbf{Span}(\mathbb{C})$ where arrows are given by *abstract spans*⁷.

Definition 5.10 Concrete and Abstract Spans

Let \mathbb{C} be an arbitrary category, which has all pullbacks. A *concrete span* $A \xleftarrow{\sigma} X \xrightarrow{\tau} B$ between A and B ($A, B \in |\mathbb{C}|$) is given by a pair of morphisms (σ, τ) with the same domain.

An *abstract span* $[\sigma, \tau]$ between A and B is given by the equivalence class $[\sigma, \tau] := \{A \xleftarrow{\sigma'} X \xrightarrow{\tau'} B \mid (\sigma, \tau) \approx (\sigma', \tau')\}$ of all concrete spans between A and B that are related by \approx . The latter relation is defined for two spans (σ, τ) and (σ', τ') between the same pair of objects (A, B) if and only if there exists an isomorphism $\iota : X \rightarrow X'$ such that $\sigma = \sigma' \circ \iota$ and $\tau = \tau' \circ \iota$, see Fig. 5.10a.

Abstract spans can be composed via pullbacks, which can intuitively be described as *relation composition*.

Definition 5.11 Span Composition

Let $[\sigma, \tau]$ be an abstract span between A and B , and $[\lambda, \rho]$ be an abstract span between B and C ($A, B, C \in |\mathbb{C}|$). Their composition $[\lambda, \rho] \circ [\sigma, \tau] := [\sigma \circ \lambda', \rho \circ \tau']$ is given by the abstract span that contains $(\sigma \circ \lambda', \rho \circ \tau')$, where λ' and τ' arise from constructing any chosen pullback of τ and λ , see Fig. 5.10b.

Thus, we yield the span category $\mathbf{Span}(\mathbb{C})$ over \mathbb{C} . It shares the same objects with \mathbb{C} and arrows are given by abstract spans. The latter will be denoted as $[\sigma, \tau] : A \rightarrow B$ to visually distinguish them from arrows in \mathbb{C} .

Proposition 2 Span Category $\mathbf{Span}(\mathbb{C})$

For each category \mathbb{C} which has all pullbacks there is a span category, whose objects coincide with \mathbb{C} , i.e. $|\mathbf{Span}(\mathbb{C})| = |\mathbb{C}|$, and morphisms are given by all abstract spans between these objects. The identity morphism for an object $A \in |\mathbf{Span}(\mathbb{C})|$ is given by the abstract span $[\text{id}_A, \text{id}_A]$ and composition is defined in Def. 5.11.

Proof. Follows immediately from the fact that pullbacks preserve identities and their construction order is associative up to isomorphism. The fact that composition relies on chosen pullbacks does not cause a problem here because I am working with abstract spans that are closed under isomorphisms. \square

The underlying category \mathbb{C} can be embedded into $\mathbf{Span}(\mathbb{C})$. The functor that embeds \mathbb{C} into $\mathbf{Span}(\mathbb{C})$ is called the *graphing functor* Γ [226] due to historic reasons⁸.

⁶Pullbacks are only uniquely defined up-to isomorphism. In order to use pullbacks operationally, one requires some pre-made choice that provides a concrete span for a given co-span that makes the resulting square a pullback.

⁷Actually, the span-category $\mathbf{Span}(\mathbb{C})$ is a *bicategory*. To avoid working with *2-arrows* and *pseudo-functors*, I utilise equivalence classes of compatible spans.

⁸Recall that every function $f : A \rightarrow B$ gives rise to a relation $\text{graph}(f) = \{(a, f(a)) \mid a \in A\} \subseteq A \times B$,

Definition 5.12 Graphing Functor Γ

The *graphing functor* Γ is an identity-on-object functor defined as follows

$$\Gamma : \begin{cases} \mathbb{C} & \rightarrow \mathbf{Span}(\mathbb{C}) \\ f \in \mathbb{C}(A, B) & \mapsto [\text{id}_A, f] \in \mathbf{Span}(\mathbb{C})(A, B) \end{cases}$$

The span category $\mathbf{Span}(\mathbb{C})$ has some notable subcategories. One example is the category of partial morphisms $\mathbf{Par}(\mathbb{C}) \subseteq \mathbf{Span}(\mathbb{C})$, which imposes the restriction that the left legs σ of abstract spans $[\sigma, \tau]$ must be a *monomorphism*, i.e. enforcing a *right-unique* relation in a set-theoretical context. Composition in this category is well-defined because pullbacks preserve monomorphisms. For a complete treatise of arbitrary categories of partial morphisms I refer to the seminal work of Robinson and Rosolini [400]. Partial morphisms will play an important role in the coming sections, therefore I introduce a special notation for them using a half-arrow tip: $[\sigma, \tau] : A \rightarrow B \in \mathbf{Par}(\mathbb{C})^{\rightarrow}$. Recall that for $\mathbb{C} := \mathbb{G}$, one can refer to elements $x \in G$ inside an objects $G \in |\mathbb{G}|$. Therefore, there are monomorphic *inclusion*-arrows $\subseteq_{A,B} : A \hookrightarrow B$ – highlighted by a hook-arrow – that do not “rename” elements, i.e. with \mathbb{G} being the underlying signature of \mathbb{G} , for all $s \in |\mathbb{B}|$ there are inclusions $A(s) \subseteq B(s)$ between carrier sets. Since abstract spans are only determined up to isomorphism, I introduce the convention to always choose the left legs of a representative for an abstract span in $\mathbf{Par}(\mathbb{G})$ to be an inclusion-morphism. This allows me to introduce the shorthand notation for partial morphisms $f : A \rightarrow B \in \mathbf{Par}(\mathbb{G})^{\rightarrow}$, which is represented by the span $(\subseteq : \text{dom}(f) \hookrightarrow A, f : \text{dom}(f) \rightarrow B)$ where the object $\text{dom}(f) \subseteq A$ is called the *domain of definition* of f . The morphism f is called “total” if the inclusion is an identity. A span, where both legs are monomorphisms $(\sigma : X \hookrightarrow A, \tau : X \hookrightarrow B)$ is able to model that an element a

- has been *deleted* ($a \in A \wedge \nexists x \in X : \sigma(x) = a$),
- has been *inserted* ($a \in B \wedge \nexists x \in X : \tau(x) = a$),
- has been *renamed* ($\exists x \in X : \sigma(x) = a \wedge a \neq \tau(x)$), or
- stayed *unchanged* ($\exists x \in X : \sigma(x) = a = \tau(x)$).

, during the update from A to B . *Moving* a node is modelled by inserting and deleting a context-edge in a category of graph-like structures. In $\mathbf{Span}(\mathbb{C})$, i.e. without the monomorphism restriction on the span legs, one can further express *splitting* and *merging* of elements.

5.1.3.3 Operational-Deltas = Graph Transformation Rules

Operational deltas are semantically even more rich than structural deltas by providing a high-level description of “what” happened during a change. For example, the operational delta depicted in Fig. 5.9 (topmost part) is able to express the fact that the

called the *graph* of the function. The function $f : A \rightarrow B$ thus turns into the span (id_A, f) of projections from the graph to the domain and codomain of f .

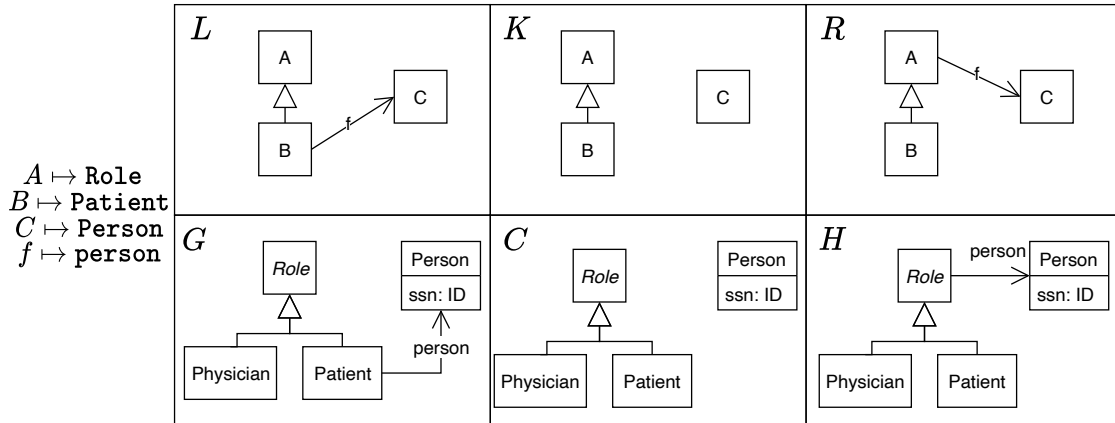


Fig. 5.11: Rule Application of a Graph Transformation Rule

Fact 3 **Graph Transformation in \mathbb{G} [304]**

Any category of graph-like structures \mathbb{G} admits DPO transformations.

5.1.4 Formalising Correspondence

Now that we are able to formalise all aspects of model spaces, it only remains to provide a formalisation of the concept of correspondences to cover all entities of multi-model consistency management. But first, let me define the setting in which multi-models are considered.

First, I have to define a *multi-model setting* which demarcates the scope of model spaces under consideration.

Definition 5.14 **Multi-model setting**

A *multi-model setting* $(\mathbb{B}, n, \{\mathcal{M}^1, \dots, \mathcal{M}^n\})$ is given by a *base language* \mathbb{B} , a natural number $n \in |\mathbb{Nat}|$ called the *arity* of the multi-model setting, and an n -indexed family of \mathbb{G} -sketches $(\mathcal{M}^j)_{1 \leq j \leq n}$, representing the *metamodels* of the domains (*model spaces*) under consideration. The carrier $M^j := |\mathcal{M}^j|$ for $1 \leq j \leq n$ is called a *metamodel graph*. A \mathbb{G} -morphism $t^j : A^j \rightarrow M^j \in |\mathbb{G} \downarrow M^j|$ for $1 \leq j \leq n$ is called a *local model* in the multi-model setting.

For the remainder, I will fix a multi-model setting with sufficiently large arity n (the maximal number of domains under consideration). As a consequence, I will be regularly working with indices. By convention, I use i and j as index variables, where i runs between $0 \leq i \leq n$ and j runs between $1 \leq j \leq n$, if not specified otherwise.

A multi-model is a concrete representative for a correspondence relation. According to the principle stated in in Sec. 3.3.2, a multi-model comprises a tuple of models augmented with *commonalities*. Sec. 4.2.4 identified various concrete approaches to represent this type of information (e.g. via complements, trace-links, etc.). In this section, I am looking for a universal formal interpretation. Multi-models can be defined among metamodels as well as (instance) models. Hence, commonalities represent different kind of inter-model relation. Let us consider the situation depicted in Fig. 1.9

Formalisation

as a concrete example. The multi-model setting has an arity n of 3. M^1 representing the graph of BPMN metamodel (Fig. 5.4), M^2 being the graph of the UML metamodel, and M^3 being the graph of the DMN metamodel. The respective local models $t^1 : A^1 \rightarrow M^1$, $t^2 : A^2 \rightarrow M^2$, and $t^3 : A^3 \rightarrow M^3$ are shown in Fig. 1.9. The figure contains coloured links, which represent different traceability relationships, i.e. the commonality data among the instances. These relations are subject to global consistency rules (CR5–CR8). In order to automate the verification process, these must be formulated in a formal language. There are specialised languages for the definition of global consistency rules such as ATL, QVTr or JTL. Sec. 4.5 uncovered that the majority these languages lack support for multi-ary relationships. Notable exceptions are the QVTr-specification (yet, there is no QVTr-implementation supporting multi-ary relations) and a language presented by Klare and Gleitze in [279].

```
1 // Realises CR5
2 commonality DecisionTableDef {
3   // Projections
4   with BPMN:Activity whereat BPMN:Activity.type equals BUSINESS_RULE
5   with DMN:DecisionTable
6   // Features
7   has input referencing ColumnAttrImpl {
8     = BPMN:Activity:consumes
9     -> DMN:Table:inputSideColumns
10  }
11  has output referencing ColumnAttrImpl {
12    = BPMN:Activity:produces
13    -> DMN:Table:outputSideColumns
14  }
15 }
16 // Realises CR6
17 commonality DataObjectImpl {
18   with BPMN:DataObject
19   with UML:(Class or Attribute)
20 }
21 // Realises CR7 and CR8
22 commonality ColumnAttrImpl {
23   with DMN:Column
24   with UML:Attribute
25   with BPMN:DataObject?
26
27   has type referencing BaseType {
28     = UML:Attribute.type
29     = DMN:Column.type
30   }
31   has entails referencing DataObjectImpl {
32     = BPMN:DataObject
33     -> UML:Attribute
34   }
35 }
36 // Required to define the feature type in the commonality above
37 commonality BaseType {
38   with UML:DataType
39   with DMN:ColumnType
40
41   has name {
42     = UML:DataType.name
43     = DMN:ColumnType.name
44   }
45 }
```

Listing 5.3: Consistency Rules in *Commonalities* language [279]

List. 5.3 gives a concrete example featuring a formal definition of the consistency rules CR5-CR8. Fittingly, the central keyword in this language is named *commonality*,

which defines a so-called *participation* constraint among the referenced concepts (keyword with). A participation constraint describes a rule of the form “for every element of type X in model one there must be a corresponding element of type Y in model two and vice versa”, compare the enforcement semantics of QVTr [327, 367]. The language comprises various additional, e.g. filter criteria (whereat in line 4), optional participation or set to be optional (?-symbol in line 25), value comparisons (=symbol at multiple lines), or dependent participations (->-symbol at multiple lines). Most interestingly, participation constraint can have “features” (keywords has/referencing), which establish dependencies between the constraints.

I do not want to go much further into the intricate details of this language and its not important to understand all its technical details (the interested reader may have a look into Gleitze’s Bachelor thesis [196]). The example List. 5.3 is primarily intended to serve as an illustration of a language for defining global consistency with support for multi-ary inter-model correspondences. If we abstract away from the concrete semantics and only consider the syntactical “skeleton”, the content of List. 5.3 is simply a collection of relationships between elements (concepts) from different metamodels. These relationships are also linked together, i.e. there is an internal structure. Graph-like structures (Sec. 5.1.1) have been shown to capture internal structure. Therefore, it is reasonable to use the same “language” to formally represent the content of List. 5.3.

The resulting graph M^0 is shown in the centre of Fig. 5.12. Each participation constraint represents a node¹⁰ while their features are represented as edges. The elements of M^0 are depicted using dashed lines and I call them *commonality witnesses*. Commonality witnesses reify a “tupling” of terms from disparate (meta-) models. Fig. 5.12 visualises these references as dashed arrows (p_1^M, p_2^M, p_3^M) . I call them *projections* and they represent the “fundamental innovation” compared to considering only tuples of local (meta-)models. For example, lines 22-35 specify a commonality of the triple DataObject (M^1), Attribute (M^2), and Column (M^3) reified under the name ColumnAttrImpl in M^0 . Simultaneously, lines 28-29 specify a commonality between the type features of Attribute and Column in M^2 and in M^3 . Common edges require that their respective owner and target nodes are also related, e.g. the type-commonality *depends* on a commonality between Attribute and Column, which is already given by the surrounding commonality-statement, as well as commonality between ColumnType and DataType (lines 37-45). Formally, this can be expressed by stating the requirement that commonality specifications must preserve edge-node-incidences.

The formal result of the specification in List. 5.3 are 3 *projection morphisms* $p_j^M : M^0 \rightarrow M^j$ ($j \in \{1, 2, 3\}$), depicted by dotted arrows in Fig. 5.12. For example, $p_j^M(\text{DecisionTableDef}) = \text{Activity} \in M^j$.

The above required edge-node-incidence means that definedness of $M^j(e)$ entails definedness of $p_j^M(v)$, where v is the owner of e in p_j^M , and

$$p_j^M(v) = \text{owner}^{M^j}(p_j^M(e)) \quad (5.6)$$

for all edges e in M^0 and likewise for targets (and generally for all $\text{op} : s \rightarrow s' \in \mathbb{B}^{\rightarrow}$).

¹⁰Note that the participation constraint realising CR6 references two elements in M^2 , thus there must be two nodes for DataObjectImpl in Fig. 5.12

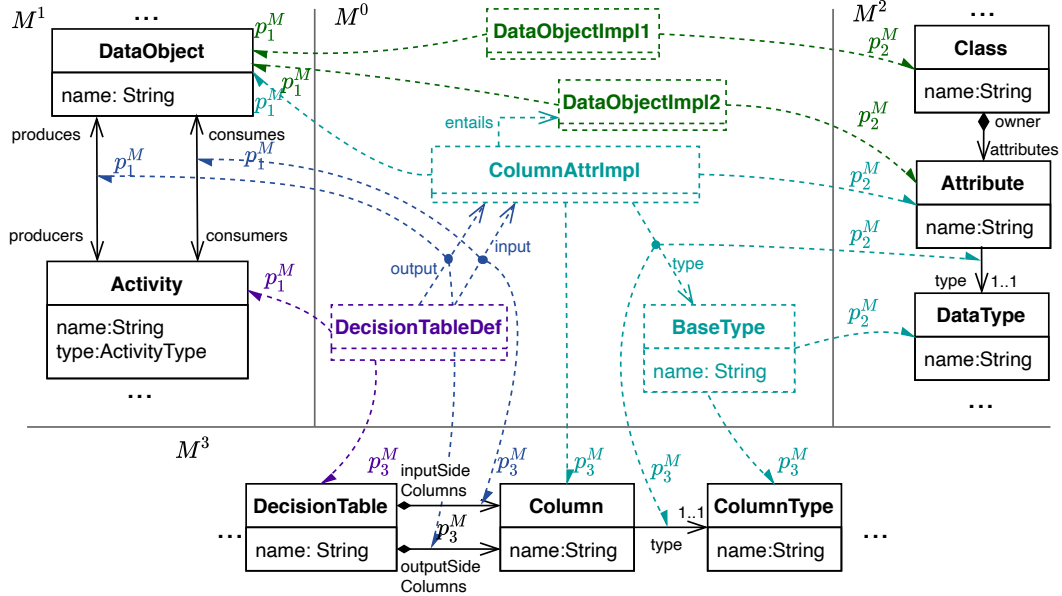


Fig. 5.12: Commonality representative metamodel M^0

Since the commonality tuples can be of arbitrary arity, the morphism may be *partial* as exemplified by the following example : The named tuple

$$\text{BaseType}(\text{UML}:\text{DataType}, \text{DMN}:\text{ColumnType})$$

is expressed by a node $\text{BaseType} \in M^0$ where the projection morphisms p_1^M, p_2^M, p_3^M are defined as follows (a mapping that is undefined for a given element is expressed via \perp):

$$p_1^M(\text{BaseType}) = \perp, p_2^M(\text{BaseType}) = \text{DataType}, p_3^M(\text{BaseType}) = \text{Type}$$

Therefore, the transition from binary spans (see Sec. 5.1.3.2) to multi-ary spans simultaneously requires the transition from *total* to *partial* morphisms. An alternative approach would be to define a collection of auxiliary graphs M^{ij} together with spans $M^i \leftarrow M^{ij} \rightarrow M^j$ for each pair (i, j) of indices between 1 and n . This setting, however, is more complex, because of the rapidly growing number of auxiliary structures for which, additionally, consistency requirements have to be considered¹¹. But even more problematic: This approach cannot express multi-ary relationships, which are, for instance, required to formulate CR8, compare the discussion in Sec. 3.3.5. My goal is to overcome current limitations (Sec. 4.5), especially offering support for multi-ary situations. Therefore, I choose the centralised approach.

Categorically, a multi-ary span of partial graph morphisms can be expressed as a special diagram functor $\mathcal{M} : \mathbb{I}_n \rightarrow \mathbb{G}$, i.e. a selection of objects and morphisms in \mathbb{G} . The schema category \mathbb{I}_n has the star-shape defined in (5.7) (identity arrows of \mathbb{I} are omitted). These functors are subject to the condition that the inner edges $(10, \dots, n0)$

¹¹ For instance, a commonality declaration in $M^{1,2}$ of elements $x_1 \in M^1$ and $x_2 \in M^2$ and a declaration in $M^{2,3}$ specifying correspondence of $x_2 \in M^2$ and $x_3 \in M^3$ must necessarily (redundantly) be considered in $M^{1,3}$.

Formalisation

Fig. 1.9, it turns out that the coloured lines crossing the model boundaries can be expressed as multi-model span \mathcal{A} . Every line joint becomes a commonality witness and the projections are given by ends of these lines. The only difference is that one is additionally dealing with *typing*, i.e. there are typing morphisms $t^j : A^j \rightarrow M^j$. Luckily, this typing extends to A^0 as well because elements a_j and a_k ($j \neq k$) of model components A^j and A^k are relatable only if their types $t^j(a_j)$ and $t^k(a_k)$ are related via a commonality witness $w \in M^0$. The latter is prepared in List. 5.3.

A natural typing t^0 of a commonality representative v of a_j and a_k is $t^0(v) := w$, such that

$$p_j^M(t^0(v)) = p_j^M(w) = t^j(a_j) = t^j(p_j^A(v)), \quad (5.9)$$

which shows that t_0 integrates smoothly into a composed typing expressed by the multi-model span morphism $t : \mathcal{A} \rightarrow \mathcal{M}$.

Proposition 4 Multi-model span category \mathbb{M}

Multi-model spans together with their morphisms establish a category \mathbb{M} .

Proof. Follows immediately from the fact that $\mathbb{M} \subseteq \mathbb{G}^{\mathbb{I}}$ is a full subcategory of the functor category $\mathbb{G}^{\mathbb{I}}$. \square

The fact that multi-model spans form a category immediately yields the notion of slice categories $\mathbb{M} \downarrow \mathcal{M}$. This means that one can consider instances for a given multi-model span, which immediately yields a formal interpretation of domain specific trace-models: An alignment of metamodel graphs (e.g. M^1, M^2, M^3) augmented with type-commonalities (e.g. List. 5.3) forms a type-multi-model span (i.e. the domain specific trace metamodel). The respective instance (i.e. the domain specific trace model) $t : \mathcal{A} \rightarrow \mathcal{M}$ is given by the models (e.g. $t^1 : A^1 \rightarrow M^1, t^2 : A^2 \rightarrow M^2, t^3 : A^3 \rightarrow M^3$) augmented with typed trace links (e.g. the links in Fig. 1.9).

Conditions (5.6) (compatibility of projections with owner/target) and (5.9) (compatibility of typing and projections) are encoded into Def. 5.15 (\mathbb{I} -arrows are interpreted as graph morphisms) and Def. 5.16 (commutativity of (i) and (ii)) respectively. Fig. 5.13 shows an excerpt of $t : \mathcal{A} \rightarrow \mathcal{M}$ to illustrate both conditions at the running example.

5.2 Verification

Consistency verification is based on consistency rules, compare Sec. 3.3.3. Sec. 4.2.6 showed that there is a plethora of ways for representing consistency rules and implementing consistency verification. The goal in this chapter is to remain as “technology independent” as possible. *Institutions* are a formal framework invented by Goguen and Burstall [205] with the aim to integrate the logic systems used in various formal verification approaches.

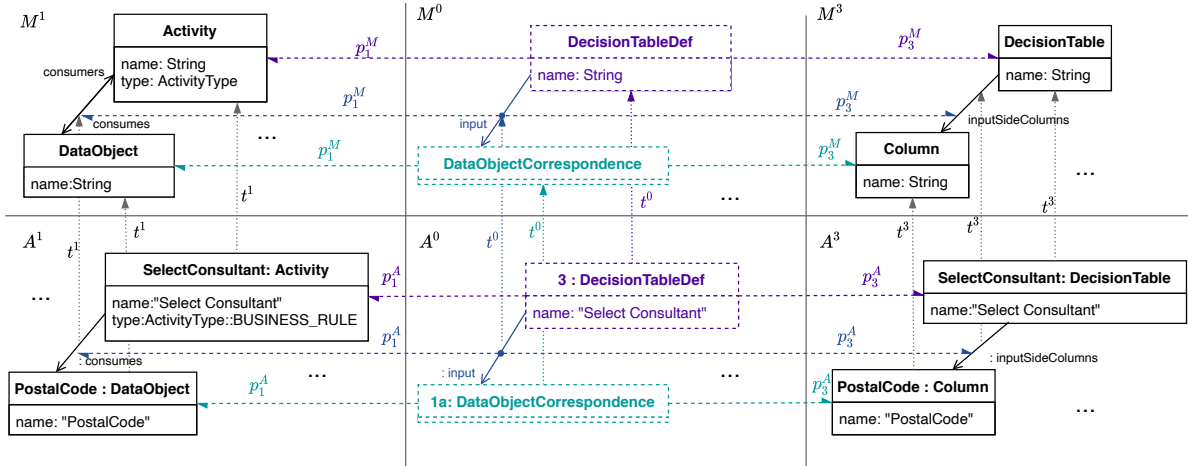


Fig. 5.13: Compatibility of typing

Definition 5.17 Institution [205]

An institution $\mathcal{I} := (\Sigma^{\mathcal{I}}, \text{Sen}^{\mathcal{I}}, \text{Mod}^{\mathcal{I}}, \models^{\mathcal{I}})$ is given by

- A category $\Sigma^{\mathcal{I}}$, whose objects are called *signatures*,
- A functor $\text{Sen}^{\mathcal{I}} : \Sigma^{\mathcal{I}} \rightarrow \mathbf{Set}$, assigning to each signature a set whose elements are called *sentences*,
- A contravariant functor $\text{Mod}^{\mathcal{I}} : \Sigma^{\mathcal{I} \text{op}} \rightarrow \mathbf{CAT}$ assigning to each signature a category of *models*, and
- A family of relations $\models^{\mathcal{I}} := (\models_S^{\mathcal{I}} \subseteq |\text{Mod}^{\mathcal{I}}(S)| \times \text{Sen}^{\mathcal{I}}(S))_{S \in |\Sigma^{\mathcal{I}}|}$ called *satisfaction*

which is subject to the following equivalence:

$$M \models_S^{\mathcal{I}}, \text{Sen}^{\mathcal{I}}(\sigma)(\phi) \Leftrightarrow \text{Mod}^{\mathcal{I}}(\sigma)(M) \models_{S'}^{\mathcal{I}} \phi \quad (5.10)$$

for each $\sigma : S \rightarrow S' \in \Sigma^{\mathcal{I} \rightarrow}$, $M \in |\text{Mod}^{\mathcal{I}}(S')|$ and $\phi \in \text{Sen}^{\mathcal{I}}(S)$.

As long as a formal approach forms an institution, it can be integrated with and re-used by other approaches via the concepts of *institution (co-)morphisms* [204]. There is tool support for this idea in the form of the *heterogeneous toolset (Hets)* [347]. Due to its high acceptance in the formal verification community, it is reasonable to adopt institutions as an overarching framework to realise consistency verification: Consistency rules are given by sentences and the verification operation is “implemented” by the satisfaction relation.

It is important to note that generalised sketches, which have been introduced in Sec. 5.1.2 to model the (syntactical) relationship between models and their meta-models, can be embedded into this framework. They do not form a “proper” institution, only a *pseudo-institution* (due to the model-functor being a pseudo-functor). Yet, this can be considered a mere technicality for the time being as it does not cause further problems

for my approach. The general idea of how generalised sketches form an institution is described below, see [493] for a more detailed treatment.

Theorem 5 Generalised Sketches induce Pseudo-Institution [493]

Each category \mathbb{C} with chosen pullbacks and a predicate signature Π forms a pseudo-institution $\mathcal{S}(\mathbb{C}|\Pi) := (\mathbb{C}, \text{Sen}^{\mathcal{S}(\mathbb{C}|\Pi)}, \mathbb{C} \downarrow _, \models^{\mathcal{S}(\mathbb{C}|\Pi)})$ where

- Signatures are given by the underlying category \mathbb{C} ,
- The sentence-functor $\text{Sen}^{\mathcal{S}(\mathbb{C}|\Pi)}$ maps each $G \in |\mathbb{C}|$ to the set of all Π -sketches whose carrier is G and each morphism $f : G \rightarrow H$ is mapped to a function that translates between sets of sketches, i.e. a sketch $\mathcal{C} \in \text{Sen}^{\mathcal{S}(\mathbb{C}|\Pi)}$ ($|\mathcal{C}| = G$) is mapped to $(H, \{(f \circ b, p) \mid (b, p) \in \text{Constr}(\mathcal{C})\})$, see (5.11).

$$\begin{array}{ccc}
 & \text{ar}(p) & \\
 b \swarrow & & \searrow f \circ b \\
 G & \xrightarrow{f} & H
 \end{array} \tag{5.11}$$

- The model-functor is $\mathbb{C} \downarrow _ : \mathbb{C}^{\text{op}} \rightarrow \mathbf{Cat}$, which maps each $G \in |\mathbb{C}|$ to the slice category $\mathbb{C} \downarrow G$ and the contravariant morphism mapping is defined via the chosen pullback-functor, i.e. a morphism $f : G \rightarrow H$ is mapped to the following functor

$$f^* := \begin{cases} \mathbb{C} \downarrow H \rightarrow \mathbb{C} \downarrow G \\ i : I \rightarrow J \in |\mathbb{C} \downarrow H| \mapsto i^* : O \rightarrow G \in |\mathbb{C} \downarrow G| \\ k : j \rightarrow i \in \mathbb{C} \downarrow H \mapsto k^* : j^* \rightarrow i^* \in \mathbb{C} \downarrow G \end{cases}$$

where i^* and j^* are defined via chosen pullbacks and k^* arises from the universal pullback property, see (5.12).

$$\begin{array}{ccccc}
 Q & \xrightarrow{f^*} & J & & \\
 \swarrow j^* & & \searrow k & & \\
 & O \xrightarrow{f^*} I & & & \\
 \downarrow i^* & \lrcorner & \downarrow i & & \\
 G & \xrightarrow{f} & H & &
 \end{array} \tag{5.12}$$

- The satisfaction relation $\models^{\mathcal{S}(\mathbb{C}|\Pi)}$ is defined in Def. 5.9.

This construction is a *pseudo*-institution since the model-functor is only a pseudo-functor, i.e. pullback-functors compose only up to isomorphism [75].

Proof. The proof that (5.10) holds is given Appendix B.3. □

Hence, diagrammatic constraints cannot only be used to capture syntactic well-formedness rules but also for arbitrary consistency rules. Yet, we are still missing a

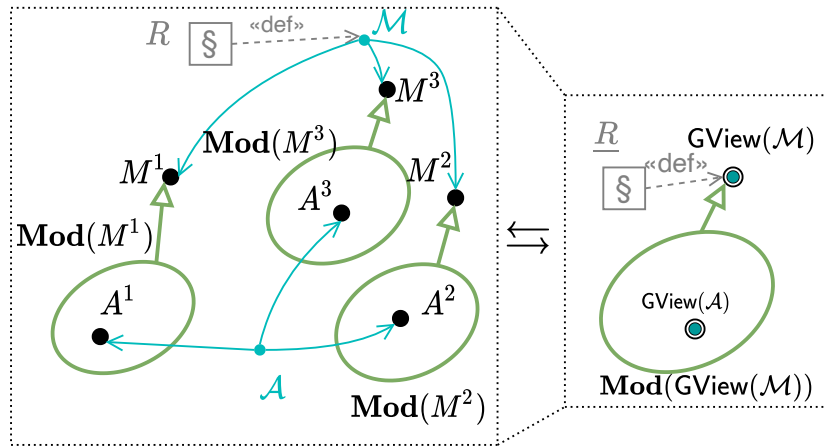


Fig. 5.14: Global View approach schematically

possibility to impose consistency rules on correspondence relations: Diagrammatic constraints are attached to single carrier graphs but correspondence relations are expressed by means of multi-model spans, i.e. a network of graphs and (partial) graph morphisms. To address this issue I will utilise the idea of “*global views*” and decided to use a centralised approach, compare Tab. 3.1, because, in Sec. 5.1.4, I already decided to represent all commonalities together in one commonality model.

This idea behind global views is schematically depicted in Fig. 5.14. The left hand side shows three model spaces where there are correspondences among their members and also their metamodels. The correspondence on the metamodel has a consistency rule attached to it. The right hand side of Fig. 5.14 depicts the same information but through the lens of the global view (GView). There, a multi-model \mathcal{M} comprising three metamodels is combined into a single metamodel that gives rise to a model space hosting a model that reifies a multi-model \mathcal{A} among three instance models on the left hand side. The integrated presentation offered by the global view has the advantage that the multi-models on the left hand side are turned into “regular” models. Hence, the global consistency rule can be treated as any other local consistency rule and *global verification* (Fig. 3.9f) can be implemented in terms of *internal verification* (Fig. 3.9e).

In the following I am seeking for a formal construction that realises this conceptual idea. During the course of this PhD project, I pursued two different approaches for realizing global views: *merging* and *weaving* (= comprehensive systems).

5.2.1 Verification via Merging: Colimit

The most well-known example of a global view approach is the “consistency-checking-via-merging” [70, 134, 390, 412] approach, which has been referred to as the “standard” approach for verifying the consistency of multiple inter-related structural models [275, 284], see Sec. 4.4.2 for a practical example. A formal definition of this approach is based on the categorical concept of a *colimit* (Def. C.17). The latter can intuitively be described as a coordinated merge where all elements from the individual models are collected into a new model wherein elements related by commonalities are identified. The idea of describing a merge of different system by means of colimits was first proposed by Goguen in the 70’s [202]. The idea was introduced to the software

engineering community by Sabetzedah and Easterbrook [410] and subsequently extended by several researchers regarding heterogeneous models [134], localisation possibilities [294, 294], and applications for bidirectional model synchronisation [445].

Note that every \mathbb{M} -object is a diagram in \mathbb{G} and that \mathbb{G} possesses all colimits since it is a topos (Fact 1). Thus, one can construct a colimit $(M^+, (\bar{m}^j : M^j \rightarrow M^+)_{1 \leq j \leq n})$ for every multi-model span $\mathcal{M} = (p_j^M : M^0 \rightarrow M^j)$. For the category of graph-like structure \mathbb{G} , the definition of the colimit can be given constructively.

Definition 5.18 Merge of Multi-model span (Colimit in \mathbb{G})

Let a multi-model span $\mathcal{M} = (p_j^M : M^0 \rightarrow M^j)_{1 \leq j \leq n}$ be given and $\bigsqcup \mathcal{M}$ denote the coproduct of all involved graphs including the commonality graph M^0 : $\bigsqcup \mathcal{M} = \bigsqcup_{0 \leq i \leq n} M^i$. We now define a relation \sim on $\bigsqcup \mathcal{M}$.

$$x \sim x' \text{ iff } \exists j \in \{1, \dots, n\} : p_j^M(x) = x'$$

Let \equiv be the least equivalence relation on $\bigsqcup \mathcal{M}$ comprising \sim^a .

I then define

$$M^+ := \left(\bigsqcup_{0 \leq i \leq n} M^i \right) / \equiv$$

as the *merge* of the multi-model span \mathcal{M} .

For each $i \in \{0, 1, \dots, n\}$ let $\bar{m}^i : M^i \rightarrow M^+$ be the graph morphism which maps each x to its equivalence class $[x]_{\equiv}$.

^aActually this is a family of equivalence relations, one for each $s \in |\mathbb{B}|$.

The following statements about Def. 5.18 hold:

Proposition 6 Soundness

1. \equiv is a congruence relation, i.e. it is compatible with the operational structure (the morphisms) in \mathbb{B}
2. M^+ is a graph.
3. $(\bar{m}^i : M^i \rightarrow M^+)_{0 \leq i \leq n}$ is the colimiting cocone of the diagram \mathcal{M} in \mathbb{G} .
4. If \mathcal{A} is another multi-model span and A^+ is its merge, and there is a multi-model span morphism $t : \mathcal{A} \rightarrow \mathcal{M}$, then there is a unique morphism $t^+ : A^+ \rightarrow M^+$ such that for all $0 \leq i \leq n$ it holds that $t^+ \circ \bar{a}^i = \bar{m}^i \circ t^i$.

Proof. The proof of (1) follows from the definition of \sim , (2) from the fact that colimits in graphs can be computed sortwise, i.e. for each object $s \in |\mathbb{B}|$ [207] and (3) from standard text books on category theory, e.g. [20], [326]. (4) follows immediately from the universal property of colimits. \square

The final statement (4) in Proposition 6 guarantees that the merge is able to reflect multi-model span-morphisms as well, i.e. the colimit construction $_+$ extends to a

functor from \mathbb{M} to \mathbb{G} . Thus, typing relationships between multi-model spans can be translated to typing relationships between merges. Moreover, Def. 5.18 can directly be translated into an algorithm that is applicable for all kinds of graph-like structure.

Algorithm 1 Merge Computation

Let initially $M^+ = \emptyset$ and $k_i : M_i \rightarrow M^+$ be totally undefined.

For all $i \in \{1, \dots, n\}$ **do:**

For all $x \in M_i$ **do:**

- **If** k_i is defined on x , **then** next x .
- **If** k_i is undefined for x , **then**
 1. Determine all elements $x_0 \in M_0$ with $x_0 \sim x$ and further for all $j \in \{1, \dots, n\}$ all $x_j \in M_j$ with $x_0 \sim x_j$. **Repeat** this procedure with $x := x_j$ for all j until the set of collected elements is constant.^a
 2. x together with the set of these x_j make up equivalence class $[x]_{\equiv}$. Add $[x]_{\equiv}$ to M^+ .
 3. Extend the definitions of k_i by $k_i(x) = [x]_{\equiv}$ and the definitions of all k_j by $k_j(x_j) = [x]_{\equiv}$, **if** some $x_j \in M_j$ was detected in (1).

^a For this, it is necessary to limit considerations to *finite* graphs, which is no loss of expressiveness for practical scenarios.

Remark 5.4 “Naming”-Strategy

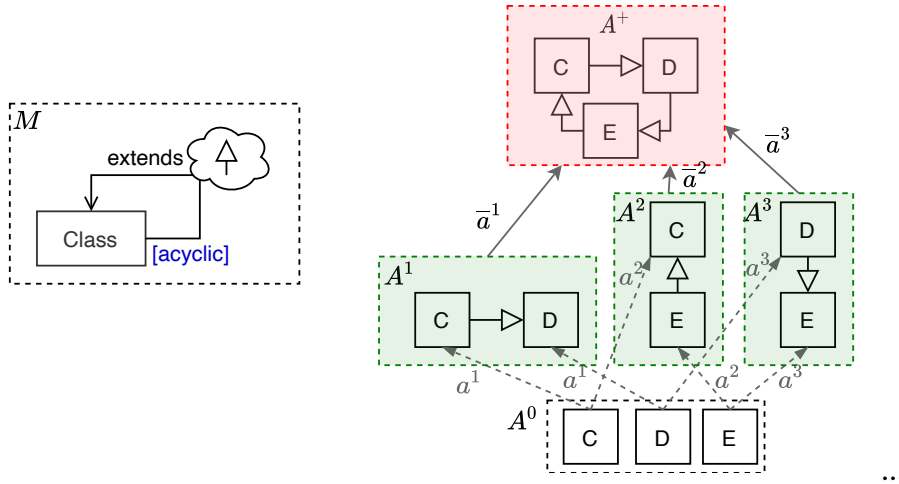
Colimit objects are only uniquely defined up to isomorphism, i.e. the names of the resulting elements (congruence classes) can be chosen arbitrarily. Therefore, a practical implementation of Algorithm 1 for a concrete data structure representing graph-like structures requires a strategy for naming elements that represent non-empty congruence classes (singleton classes take over the name of their only element). One could, for instance, use the name of the commonality element in M^0 as the name of the class so that multi-model spans also serve as a specification for the resulting merge (this idea will be revisited in Sec. 6.1.4).

The colimit object M^+ for a multi-model span \mathcal{M} is a graph and represents a global view over all elements in the involved models. Hence, one can impose diagrammatic constraints representing global consistency rules on M^+ , which results in a sketch \mathfrak{M}^+ . A multi-model instance $t : \mathcal{A} \rightarrow \mathcal{M}$ can be represented as a \mathbb{G} -morphism $t^+ : A^+ \rightarrow M^+$. Verifying global consistency means verifying whether t^+ is an instance w.r.t. \mathfrak{M}^+ or not, i.e. $t^+ : \text{Inst}(\mathfrak{M}^+)$?

Example 5.1 Inconsistent Class Diagrams revisited

The drawing below depicts the Example from Fig. 3.13, which comprises three class diagrams violating the acyclicity of the inheritance relationship. The left hand side shows the metamodel sketch \mathfrak{M} , which is the same for each of the three

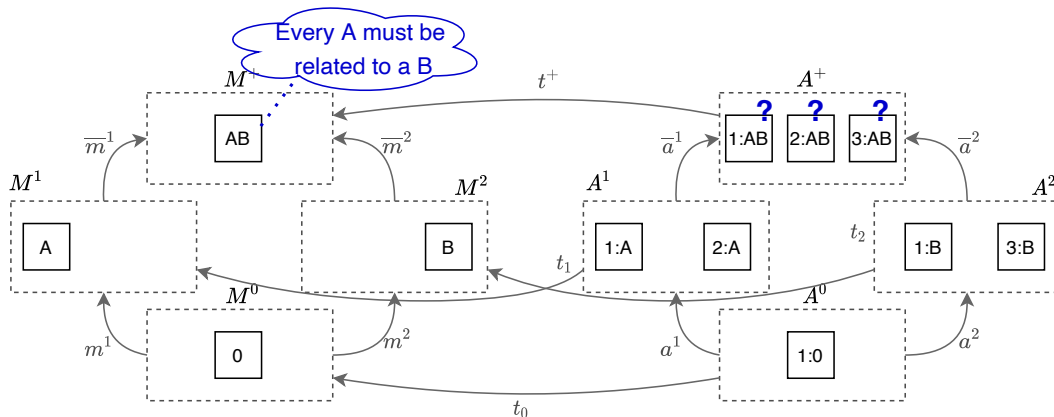
models A^1, A^2 and A^3 . The annotation [acyclic] indicates a diagram, whose predicate semantics require the absence of cycles in the network of inheritance edges. The topmost model on the right-hand side represents the colimit object A^+ , which allows to detect the global inconsistency.



However, the merge approach has some issues, which made me question its aptitude for realising global views in multi-model consistency management and instead pursue the alternative *comprehensive system* approach

Problem 5.1 Colimit “loses” information [minor issue]

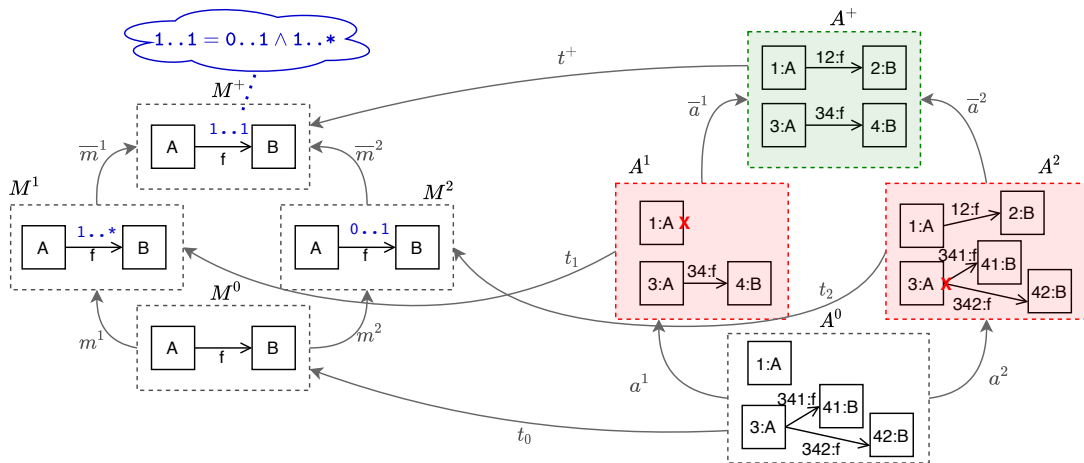
The first (debatable) shortcoming refers to the fact that the merge (colimit object) loses the *origin* information of the individual elements. For example, the drawing below shows two multi-model spans \mathcal{M} (metamodel) and \mathcal{A} (model) of arity 2 and their respective merges M^+ and A^+ . In A^+ all elements have been retyped to AB and it is not possible to see where these elements originally came from in order to verify a constraint like “Every A-element there must exist a B-element”.



This issue, however, can easily be overcome in practice by adding the respective origin information during the construction of the merge programmatically. Theoretically, one retains the origin information by considering the complete colimit $(M^+, \bar{m}^i : M^i \rightarrow M^+)_{0 \leq i \leq n}$ and not only the colimit object.

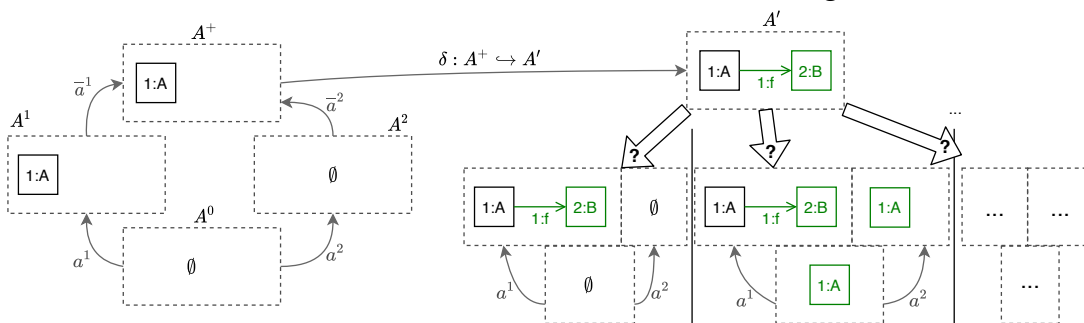
Problem 5.2 Local Consistency is not reflected

The first major drawback of the merge approach is that it has problems to reflect the *local* consistency state in a multi-model. The left-hand side of the drawing below contains the metamodels of the models shown on the right-hand side. The two multiplicity constraints in M^1 (at least one) and M^2 (at most one) are merged into “exactly one” in M^+ . The instance multi-model contains two models $t^1 : A^1 \rightarrow M^1$ and $t^2 : A^2 \rightarrow M^2$ that are both inconsistent w.r.t. their respective metamodels. However, by defining commonalities as shown in A^0 , the resulting merged model $t^* : A^+ \rightarrow M^+$ becomes consistent.



Problem 5.3 Backpropagation is unclear

The second major drawback is that retaining local model states from a merged model is not always clear, i.e. the “colimit-map” that assigns a merged model $M^+ \in |\mathbb{G}|$ to each multi-model span $\mathcal{M} \in |\mathbb{M}|$ is generally not injective. The drawing below exemplifies this: There is an inclusion morphism $\delta : A^+ \hookrightarrow A'$ that shall represent an update (Sec. 5.1.3) on the merged model A^+ (attaching a newly added B-object to the existing A-object). Now, there are multiple choices in translating this \mathbb{G} -morphisms into a respective \mathbb{M} -morphism whose codomain’s colimit is A' . These choices are indicated through the bold arrows.



5.2.2 Verification via Weaving: Comprehensive System

The aforementioned shortcomings motivated me and my supervisors to pursue a different approach for global views. The result of this endeavour is a formal construction

Formalisation

called *comprehensive system*. This construction was first introduced in [446] and was subsequently further investigated further in [302, 448, 450] Comprehensive Systems are motivated by the fact that *commonalities*, i.e. structural relationships between elements from disparate models, are the *essential* information distinguishing a multi-model from “normal” (local) models. Comprehensive Systems internalise this information into their structure and provide a global view. The build-up of a comprehensive system is similar to graph-like structure (Def. 5.2). It encompasses local models (called components) together with their commonalities (called witnesses + projections):

Definition 5.19 Comprehensive Systems, Components, Commonalities

A *comprehensive system* C comprises

1. For every $s \in |\mathbb{B}|$ and $0 \leq i \leq n$, there is a set $C_i(s)$
2. For every $\text{op} : s \rightarrow s' \in \mathbb{B}^{\rightarrow}$ and $0 \leq i \leq n$, there is a *total* function $C_i(\text{op}) : C_i(s) \rightarrow C_i(s')$.
3. For every $s \in |\mathbb{B}|$ and $1 \leq j \leq n$, there is a *partial* function $p_{j,s}^C : C_0(s) \rightarrow C_j(s)$

such that for all $\text{op} : s \rightarrow s' \in \mathbb{B}$ and $1 \leq j \leq n$ the following statement holds:

$$\text{If } p_{j,s}^C(x) \text{ is defined, then } p_{j,s'}^C(C_0(\text{op})(x)) \text{ is defined} \quad (5.13)$$

$$\text{and } p_{j,s'}^C(C_0(\text{op})(x)) = C_j(\text{op})(p_{j,s}^C(x)). \quad (5.14)$$

The sets $C_j(s)$ together with the total maps $C_j(\text{op})$ constitute the *components*, the sets $C_0(s)$ and total maps $C_0(\text{op})$ constitute the *commonality witnesses*, and the partial functions $p_{j,s}^C$ represent the *projections*. Note that (5.13) and (5.14) correspond to the edge-node-incidence requirements from multi-model spans, compare (5.6).

Comprehensive Systems can be related by respective structure-preserving mappings.

Definition 5.20 Morphism between Comprehensive Systems

Let C, D be comprehensive systems as defined in Def. 5.19. A *morphism between comprehensive systems* is a family

$$(f_{i,s} : C_i(s) \rightarrow D_i(s))_{s \in |\mathbb{B}|, 0 \leq i \leq n}$$

of mappings compatible with (operation) arrows, i.e. $\forall i \in \{0, \dots, n\}, \forall \text{op} : s \rightarrow s' \in \mathbb{B}^{\rightarrow}$:

$$f_{i,s'} \circ C_i(\text{op}) = D_i(\text{op}) \circ f_{i,s} \quad (5.15)$$

and compatible with partial (projection) mappings: For all $j \in \{1, \dots, n\}, s \in |\mathbb{B}|$ and $x \in C_0(s)$:

$$p_{j,s}^C(x) \text{ is defined} \implies p_{j,s}^D(f_{0,s}(x)) \text{ is defined and} \quad (5.16)$$

$$p_{j,s}^D(f_{0,s}(x)) = f_{j,s}(p_{j,s}^C(x)) \quad (5.17)$$

Alternatively, one can visualize Def. 5.20 by a family of commutative cubes in **Set**,

shown in (5.18) and indexed by all $op : s \rightarrow s' \in \mathbb{B}$ and $1 \leq j \leq n$. Commutativity of the top and bottom faces encode that the projections in the comprehensive systems C and D fulfil (5.13)+(5.14), while left and right faces encode compatibility of f with operation arrows (5.15), and back and front faces encode compatibility of f with projections (5.16)+(5.17). Compare also (5.18) with the example in Fig. 5.13.

$$\begin{array}{ccccc}
 & & D_j(s) & \xleftarrow{p_{j,s}^D} & D_0(s) \\
 & D_j(op) \swarrow & \uparrow & & \nwarrow D_0(op) \\
 D_j(s') & \xleftarrow{p_{j,s'}^D} & D_0(s') & & \\
 & \swarrow f_{j,s} & \downarrow & & \nwarrow f_{0,s} \\
 & & C_j(s) & \xleftarrow{p_{0,s}^C} & C_0(s) \\
 & f_{j,s'} \uparrow & \swarrow C_j(op) & & \nwarrow C_0(op) \\
 C_j(s') & \xleftarrow{p_{j,s'}^C} & C_0(s') & & \\
 & & \downarrow f_0(s') & & \\
 & & C_0(s) & &
 \end{array} \tag{5.18}$$

Equations (5.16) and (5.17) (f substituted by t) reflect the demanded property (5.9), i.e. compatibility of commonalities and typing discussed in Sec. 5.1.4.

Proposition 7 **Category of Comprehensive Systems \mathbb{CS}**

Comprehensive Systems together with their homomorphisms constitute a category \mathbb{CS} .

Proof. An identity is a family of identities, composition is composition of mappings $f_{i,s}$. This yields neutrality and associativity. Moreover, composed homomorphisms are still compatible with the inner structure $(C_i(op), p_{j,s}^C)$. Whereas this follows in the usual way for $op : s \rightarrow s'$, transitivity of the definedness implication in (5.16) also yields compatibility with partial functions. \square

Note that Def. 5.19 is actually parametrised by the base language \mathbb{G} and the arity n of the multi-model setting and so is the definition of multi-model spans Def. 5.15. Thus, \mathbb{CS} and \mathbb{M} must rightfully be written as $\mathbb{CS}(n, \mathbb{B})$ and $\mathbb{M}(n, \mathbb{B})$, respectively. But since I fixed an arbitrary multi-model setting in the beginning of Sec. 5.1.4, I will continue to omit these parameters when referring to \mathbb{CS} and \mathbb{M} .

Comparing the definitions of multi-model spans and multi-model span morphisms with comprehensive systems and comprehensive system morphisms, there is a strong resemblance between both constructs. The multi-model span-based definition depicts commonalities *externally* while comprehensive systems *internalise* them. In fact, the respective categories turn out to be equivalent, which is captured in the following Theorem:

Theorem 8 **Equivalence of Categories**

$\mathbb{CS} \cong \mathbb{M}$.

Proof. See Appendix B.1. \square

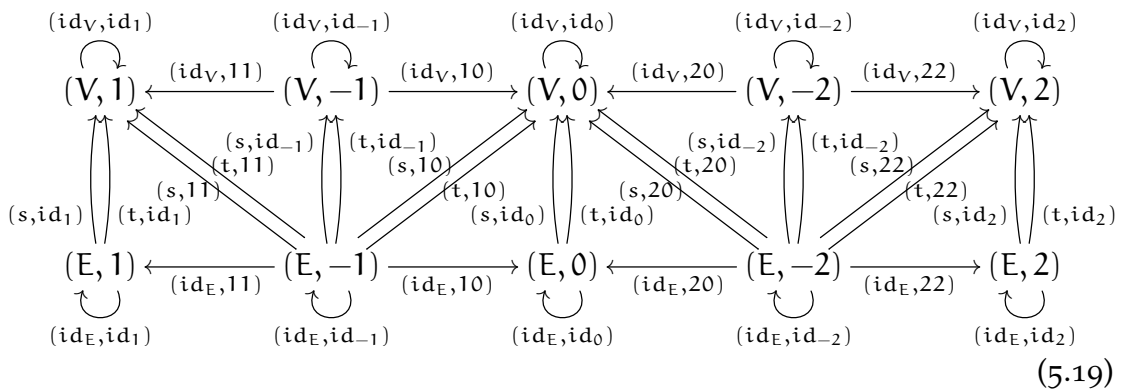
Formalisation

It is important to note that Theorem 8 is not just a technical detail: Note that an object of \mathbb{M} is a diagram, i.e. a selection of graphs and graph morphisms together. A \mathbb{CS} -object represents a single integrated structure comprising model elements and commonality relations, whose definition resembles those of graph-like structures. Thus, the equivalence in Theorem 8 guarantees that the external notion \mathbb{M} and the internal notion \mathbb{CS} are fully interchangeably and the two functors hidden in that equivalence allow to convert between the two representations on demand.

The proof of Theorem 8 relies on the fact that the functor $\mathbb{M} = \mathbb{G}^{\mathbb{I}} = (\mathbf{Set}^{\mathbb{B}})^{\mathbb{I}}$ can be “uncurried” to $\mathbf{Set}^{\mathbb{B} \times \mathbb{I}}$ (Fact 40) and that \mathbb{M} and \mathbb{CS} are subcategories of those functor categories. The product category $\mathbb{B} \times \mathbb{I}$ represents a “flattening” of the operational structure of a graph language and the star shape in (5.7). Example 5.2 gives an impression of such a product category on the example of $\mathbb{B} := \mathbb{B}_{\text{DG}}$ being the base language of directed multigraphs (Fig. 5.3a) and \mathbb{I} having the arity 2.

Example 5.2 Product of Categories: $\mathbb{B}_{\text{DG}} \times \mathbb{I}_2$

The objects and morphisms of $\mathbb{B}_{\text{DG}} \times \mathbb{I}_2$ are depicted by the graph in (5.19)



Identities are pairs of identity morphisms from \mathbb{B}_{DG} and \mathbb{I}_2 . The compositions for the non-identity morphisms are defined as follows:

$$\begin{aligned}
 (s, \text{id}_1) \circ (\text{id}_E, 11) &= (s, 11) & (t, \text{id}_1) \circ (\text{id}_E, 11) &= (t, 11) \\
 (s, \text{id}_1) \circ (\text{id}_E, 10) &= (s, 10) & (t, \text{id}_1) \circ (\text{id}_E, 10) &= (t, 10) \\
 (s, \text{id}_1) \circ (\text{id}_E, 22) &= (s, 22) & (t, \text{id}_1) \circ (\text{id}_E, 22) &= (t, 22) \\
 (s, \text{id}_1) \circ (\text{id}_E, 20) &= (s, 20) & (t, \text{id}_1) \circ (\text{id}_E, 20) &= (t, 20)
 \end{aligned}$$

It turns out that \mathbb{CS} -objects not only look similar to graph-like structures, they can also act as carriers for diagrammatic constraints, which was the original goal with global views.

Theorem 9 Pullbacks in \mathbb{CS}

\mathbb{CS} possesses chosen pullbacks.

Proof. The complete proof is given in Appendix B.2. □

Comprehensive Systems are not graph-like structures since they contain additional constraints. Still, they are sufficiently concrete that one can consider their elements and

Theorem 9 guarantees that we (theoretically) can apply existing means for consistency verification on local models. From Fact 5 and Theorem 9 the following Corollary follows, which allows us to re-use all existing means for consistency verification as long as they form an institution.

Corollary 10

Let Π be predicate signature comprising a set of predicate symbols $|\Pi|$ and $\alpha : |\Pi| \rightarrow |\mathbb{CS}|$ a function that assigns a comprehensive system to each predicate symbol. Then there is a pseudo-institution $\mathcal{I}(\mathbb{CS}|\Pi)$ of Π -constraints over comprehensive systems \mathbb{CS} .

Example 5.3 Synchronisation Predicate Signature

Let \mathbb{B} be the language of E-graphs and $n = 2$. Then the following table defines a predicate signature for multi-model synchronisation, i.e. to verify whether the information contained in two systems is up to date.

Name p	Arity system M	Visualization
forall		
integrity		

The semantics for both constraints are defined below

$$i : I \rightarrow M \in \llbracket \text{forall} \rrbracket \Leftrightarrow (\forall a \in I_1(\text{GN}) \exists c \in I_0(\text{GN}), b \in I_2(\text{GN}) :$$

$$p_{1,\text{GN}}^I(c) = a \wedge p_{2,\text{GN}}^I(c) = b)$$

$$\wedge$$

$$(\forall b \in I_2(\text{GN}) \exists c \in I_0(\text{GN}), a \in I_1(\text{GN}) :$$

$$p_{1,\text{GN}}^I(c) = a \wedge p_{2,\text{GN}}^I(c) = b)$$

$$i : I \rightarrow M \in \llbracket \text{integrity} \rrbracket \Leftrightarrow (\forall f_1 \in I_1(\text{NAE}), f_2 \in I_2(\text{NAE}) : \exists c \in I_0(\text{GN}) :$$

$$p_{1,\text{GN}}^I(c) = I_1(\text{owner})(f_1) \wedge p_{2,\text{GN}}^I(c) = I_2(\text{owner})(f_2))$$

$$\implies$$

$$\exists \text{id} \in I_0(\text{DN}) : p_{1,\text{DN}}^I(\text{id}) = I_1(\text{target})(f_1) \wedge$$

$$p_{2,\text{DN}}^I(\text{id}) = I_2(\text{target})(f_2)$$

Intuitively, `forall` yields the same set of objects for a pair of classes in different components, while `integrity` checks that all features of a related pair of objects is assigned to the same values.

5.3 Restoration

Sec. 5.2 motivated how the consistency verification of multi-models can be understood as regular (local) verification performed on a global view. I want to apply the same idea to multi-model consistency restoration. The schematic workflow is shown in Fig. 5.15 for a multi-model setting with arity $n = 3$: First, a family of updates comprising δ_1 , δ_3 and a *idle* update in $\mathbf{Mod}(M^2)$ is translated into a global view of this family of updates represented by an update called d . Thus, the global view must not only be able to produce an integrated representation of a multi-model (object) but also be able to produce an integrated representation of a family of updates on the components of the multi-model. This may also include updates to the commonalities contained inside a multi-model (Note that the GView-operation has four parameters). Afterwards, the global view representation is used to perform consistency verification and model repair, relying on the assumption that there are established means that can be reused. The repair will produce another (global) update r (the fix). In the final step, this update will be interpreted as a family of updates on the multi-model by using the global view operation in the opposite direction. Note that this approach has two implicit features: First, the propagated fixes can affect the models, which were causing the synchronisation in the first place. This type of update has been called *amendment* [126] and is not considered in traditional BX, e.g. see the extra update δ'_1 that is propagated back to $\mathbf{Mod}(M^1)$. Secondly, *concurrency* is implicitly embodied because the global view always considers families of “instantaneous” updates. In practice, amendments and concurrency will require further attention (e.g. policies) during the repair process, see the recent studies [189, 370, 483].

5.3.1 Back- and Forth-propagation

Sec. 5.2 discusses two distinct implementation approaches for global views: colimits and comprehensive systems. Problem 5.3 showed that back-propagation in the colimit approach is unclear. Hence, I will analyse the back- and forth-propagation capabilities with regard to comprehensive systems. Comprehensive Systems embed local models faithfully and it is possible to retain each of them via component projection functors.

Proposition 11 Component Projection Functor

There are $n + 1$ *component projection* functors $(_)^i : \mathbb{CS} \rightarrow \mathbb{G}$ that return the respective component in a comprehensive system (on objects) and the local effect of a \mathbb{CS} -morphism respectively (on morphisms).

Proof. Using Theorem 8, for each comprehensive system C there is a respective multi-model span $\mathcal{M} : \mathbb{I} \rightarrow \mathbf{Set}^{\mathbb{B}}$ and $\mathcal{M}(i) : \mathbb{B} \rightarrow \mathbf{Set}$ is the sought graph, for $0 \leq i \leq n$. Likewise for every comprehensive system morphism $f : C \rightarrow D$ there is an multi-model span morphism $\mu : \mathcal{M} \rightarrow \mathcal{N}$, and $\mu(i) : \mathcal{M}(i) \rightarrow \mathcal{N}(i)$ is the sought \mathbb{G} -morphism. \square

A family of $n + 1$ such projection functors yields the back-propagation. Likewise, a tuple of n \mathbb{G} -morphisms representing a family of updates on local models can always

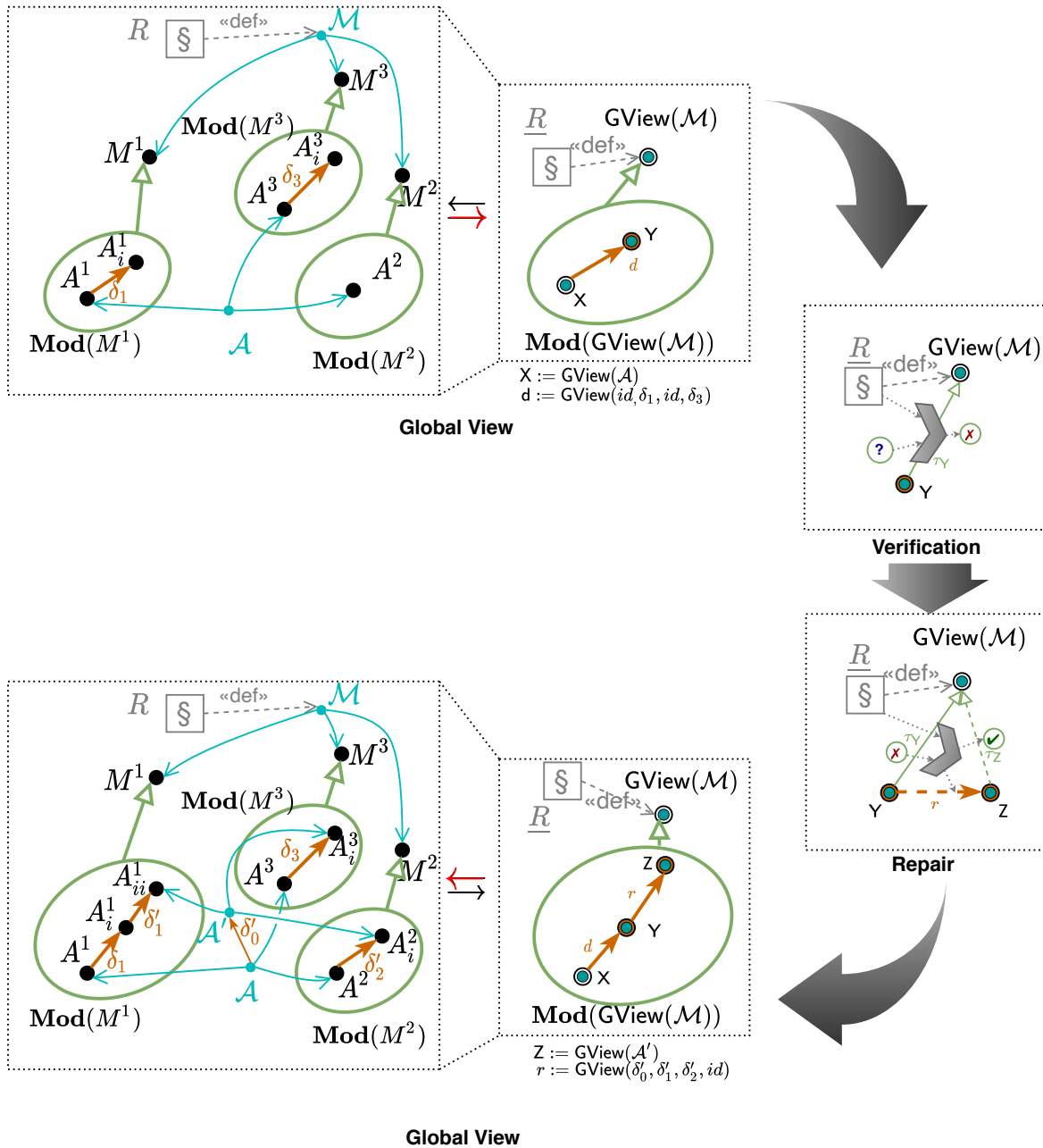


Fig. 5.15: Restoration through Global View Model Repair

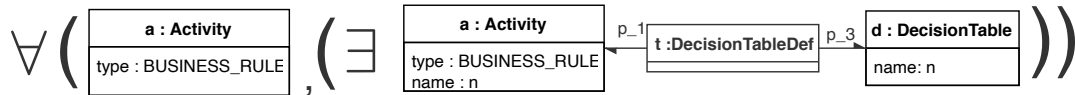


Fig. 5.16: Graph Condition Example

be interpreted as one CS-morphism, where \mathbb{G} -morphisms are the image under the respective projection functor (leaving the commonalities untouched).

5.3.2 Adhesivity

In order to investigate the model repair in greater detail, I have to leave the abstract level of institutions and generalised sketches to analyse repair possibilities of concrete rules. Sec. 4.2.8 identified two general implementation strategies for model repair: *rule-based* and *search-based*. Luckily, the framework of algebraic graph transformations that was shortly mentioned in Sec. 5.1.3 provides formal concepts to pursue either approach.

Rule-based repair approaches consider a set of rules that are going to be applied when an inconsistency is detected. A rule-based approach based on graph transformation rules and implemented in the context of EMF has been presented in [297, 354]. The repair rules in this approach are tailored towards concrete syntactic consistency rules in the EMF domain. A more general approach is given in [269, 453], which defines the notion of so-called *consistency preserving rules*. A consistency preserving rule can be split into an *edit* and a *repair* part. Whenever, one detects the application of an edit rule alone, a respective repair rule w.r.t. the definition of the consistency preserving rules can be found such that consistency is restored [453]. This idea generalises the synchronisation mechanism, which is implemented by TGGs [230], see Sec. 4.4.3 The production rules of a TGG can be interpreted as consistency preserving rules, which can be split into edit rules, which only act on the source or target of a triple graph, and repair rules, which perform the “propagation” in accordance with original production rule. The theory of the graph transformation framework guarantees that a sequence of split rule applications yields the same result as one consistency preserving rule application [144].

A big share of search-based approaches are implemented via solvers, i.e. based on translation of models and consistency rules into a logic representation. In his seminal paper *Courcelle* [100] demonstrated how graph grammars can be conceived as a logic theory and that their formal expressiveness is equal to that of First Order Logic or Monadic Second Order Logic respectively. This idea has been developed further in the form of *graph constraints* [397] and *nested graph conditions* [217]. Nested graph conditions are basically first-order sentences build over graph patterns. Fig. 5.16 shows an example, which implements the semantics of CR5. Pennemann [378] demonstrated that one can construct solvers based on graph conditions that outperform classical solvers working on set-based first order theories. Nested graph conditions have recently been used for developing a theory on search-based model repair in [218, 418, 419].

In order to apply either approach, it is required that the underlying category (CS) represents a so-called *weak adhesive high level replacement (HLR) category*. Originally, graph transformations were defined over labelled directed graphs [149]. This setting was

later abstracted to HLR structures [147] and since the discovery of adhesive categories in [304], the ambient category \mathbb{C} is generally assumed to be an arbitrary (weak) adhesive (HLR) category [150], which guarantees the validity of various theorems (parallelism, concurrency, embedding, confluence, termination) about graph transformation systems [146]. Such a category is required to have pushout diagrams that have the so-called (weak) *van Kampen* property, which asserts a certain interplay of pushouts and pullbacks and was originally discovered in [304]. Such pushouts are not necessarily required to exist for all spans but only those where at least one of the legs is a monomorphism or a special monomorphism. The class of such special monomorphisms is denoted by \mathcal{M} and must fulfil the so-called “admissibility” requirements.

Definition 5.21 Admissable Class of Monomorphisms \mathcal{M}

A subclass \mathcal{M} of all monomorphisms in a category with pullbacks \mathbb{C} is called *admissable*, if and only if

- \mathcal{M} contains the class of all isomorphisms.
- \mathcal{M} is closed under composition, i.e. for $m : A \rightarrow B, n : B \rightarrow C \in \mathcal{M}$ it holds that $n \circ m \in \mathcal{M}$.
- \mathcal{M} is stable under pullback, i.e. for $A \xleftarrow{m'} D \xrightarrow{n'} C$ being the pullback of $A \xrightarrow{n} C \xleftarrow{m} B$ and $n \in \mathcal{M}$ ($m \in \mathcal{M}$) then $n' \in \mathcal{M}$ ($m' \in \mathcal{M}$).

Definition 5.22 Van Kampen square [304]

A pushout square (f, m, n, g) as shown in the bottom of (5.20)

$$\begin{array}{ccccc}
 & & A' & \xrightarrow{m'} & B' \\
 & f' \swarrow & \downarrow n' & & \swarrow g' \\
 C' & \xrightarrow{\quad} & D' & & \\
 \downarrow c & & \downarrow a & & \downarrow b \\
 & f \swarrow & A & \xrightarrow{m} & B \\
 & \downarrow & \downarrow d & & \swarrow g \\
 C & \xrightarrow{\quad} & D & &
 \end{array} \tag{5.20}$$

is called a *Van Kampen* square if and only if

$$\text{back faces are pullbacks} \Rightarrow (\text{front faces are pullbacks} \Leftrightarrow \text{top face is pushout}) \tag{5.21}$$

Definition 5.23 Weak Adhesive HLR Category w.r.t. \mathcal{M}

Let \mathbb{C} be a category and \mathcal{M} an admissible class of monomorphisms in \mathbb{C} . A category \mathbb{C} is called a *weak adhesive HLR (High Level Replacement) category* if and only if

- \mathbb{C} has pushouts and pullbacks along \mathcal{M} -morphisms and
- pushouts along \mathcal{M} -morphisms have the *weak Van Kampen property*, i.e. (5.21) is required to hold for commutative situations with $m \in \mathcal{M}$ and ($f \in \mathcal{M}$ or $\{b, c, d\} \subseteq \mathcal{M}$), cf. the diagram in (5.20).

Hence, in order to reuse existing means for rule-based or search-based model repair based on graph transformations, the following theorem has to be proven.

Theorem 12

\mathbb{CS} is a weak adhesive HLR category w.r.t. some admissible class of monomorphisms \mathcal{M} .

First, I have to define the class of admissible monomorphisms for \mathbb{CS} and \mathbb{M} respectively. It turns out that one cannot choose *all* monomorphisms: For example, let $(m : \mathcal{A} \rightarrow \mathcal{B}, f : \mathcal{A} \rightarrow \mathcal{C})$ be a span of \mathbb{M} -morphisms. If there is an incomplete commonality specification in \mathcal{A} containing a commonality representative which relates not as many elements as its images in \mathcal{B} and \mathcal{C} , the pushout construction may produce a multi-model span \mathcal{D} , in which the projection is no longer well-defined. This effect has been studied in [296, Ex.6.] as well.

Example 5.4

Consider a category of multi-model spans $\mathbb{M}(n, \mathbb{B})$ with $n = 1$ and $\mathbb{B} := \mathbf{1}$ being a terminal object in \mathbf{Cat} (i.e. $\mathbb{G} \approx \mathbf{Set}$).

$$\begin{array}{ccccc}
 B^0 = \{\bullet\} & \xleftarrow{m_0} & A^0 = \{\bullet\} & \xrightarrow{f_0} & C^0 = \{\bullet\} \\
 \subseteq_1^B \uparrow & & \uparrow \subseteq_1^A & & \uparrow \subseteq_1^C \\
 \text{dom}(p_1^B) = \{\bullet\} & \xleftarrow{m_{-1}} & \text{dom}(p_1^A) = \{\} & \xrightarrow{f_{-1}} & \text{dom}(p_1^C) = \{\bullet\} \\
 p_1^B \downarrow & & \downarrow p_1^A & & \downarrow p_1^C \\
 B^1 = \{\bullet\} & \xleftarrow{m_1} & A^1 = \{\bullet\} & \xrightarrow{f_1} & C^1 = \{\bullet\}
 \end{array} \quad (5.22)$$

(5.22) gives an example of the situation mentioned above. Constructing a pushout of the span $(m_{-1} : \text{dom}(p_1^A) \rightarrow \text{dom}(p_1^B), f_{-1} : \text{dom}(p_1^A) \rightarrow \text{dom}(p_1^C))$ will result in a two element set, while the pushout of $(m_0 : A^0 \rightarrow B^0, f_0 : A^0 \rightarrow C^0)$ will result in a one element set. Therefore, the morphism $\subseteq_1^D : \text{dom}(p_1^D) \rightarrow D^0$ cannot be a monomorphism and the projection in \mathcal{D} is not well-defined.

Thus, I cannot expect the existence of pushouts in general. However, I claim that for \mathcal{M} being the class of *reflective* monomorphisms, \mathbb{CS} becomes a weak adhesive HLR

category, in particular pushouts along \mathcal{M} -morphisms exist.

Definition 5.24 Reflective CS-Monomorphisms

Let C, D be two comprehensive systems, a *reflective CS-monomorphism* $m : C \rightarrow D$ is a family

$$(m_{i,s} : C_i(s) \rightarrow D_i(s))_{s \in |\mathbb{B}|, 0 \leq i \leq n}$$

as defined in Def. 5.20 where every $m_{i,s}$ is injective and, additionally, the implication in (5.16) is turned into an equivalence. Thus, definedness of a projection is not only *preserved* but also *reflected*.

Remark 5.5 Reflective Monomorphisms in \mathbb{M}

As a consequence of Theorem 8, there must be a corresponding notion to Def. 5.24 in \mathbb{M} . For this, let us investigate the consequences of strengthening the definition of \mathbb{M} -morphisms (Def. 5.16) by requiring

$$p_j^N \circ [id_{M^0}, m_0] = [id_{M^j}, m_j] \circ p_j^M \tag{5.23}$$

to be a commutative square in $\mathbf{Par}(\mathbb{G})$ instead of requiring the two diagrams in (5.8) to commute. Recall that the definition of composition of partial morphisms involves pullbacks and consider the \mathbb{G} -diagram in (5.24):

$$\begin{array}{ccccc}
 M^0 & \xrightarrow{id_{M^0}} & M^0 & \xrightarrow{m_0} & N^0 \\
 \uparrow \subseteq_j^M & & \uparrow \subseteq_j^M & & \uparrow \subseteq_j^N \\
 \text{dom}(p_j^M) & \xrightarrow{id_{\text{dom}(p_j^M)}} & \text{dom}(p_j^M) & \xrightarrow{m_{-j}} & \text{dom}(p_j^N) \\
 \downarrow p_j^M & & \downarrow p_j^M & & \downarrow p_j^N \\
 M^j & \xrightarrow{id_{M^j}} & M^j & \xrightarrow{m_j} & N^j
 \end{array} \tag{5.24}$$

The bottom left and top right squares are pullbacks, which are needed for the composition of $[id_{M^j}, m_j] \circ p_j^M$ and $p_j^N \circ [id_{M^0}, m_0]$, respectively, see Fig. 5.10b. Condition (5.23) requires the apexes of these pullbacks to be equal and since the bottom left pullback is constructed along an identity, this apex can be chosen as $\text{dom}(p_j^M)$. Hence, we may define a morphism in comprehensive systems as a family $(m_{-n}, \dots, m_{-j}, \dots, m_0, \dots, m_j, \dots, m_n)$ of \mathbb{G} -morphisms defined on domains of definitions of projection morphisms $\text{dom}(p_j^M)$, on M^0 , as well as on components M^j , as shown in the right-hand side of (5.24), such that the upper squares $m_0 \circ \subseteq_j^M = \subseteq_j^N \circ m_{-j}$ are pullbacks and the lower squares $m_j \circ p_j^M = p_j^N \circ m_{-j}$ commute. Comparing this notion with Def. 5.20, the upper pullback corresponds to turning the implication in (5.16) into an equivalence (while commutativity of the lower right square corresponds to (5.17)). In such a way, definedness of a projection is not only preserved but also *reflected*.

To provide an intuition of the consequences of Def. 5.24, think of monomorphisms m as models of *insertion*: An element $x \in D$ that is not in the image of m is thought

Formalisation

of as being “inserted”, the other elements are thought of as “already existing”. Then, a reflective monomorphism is not allowed to “touch” the projections of already existing witnesses. Example 5.5 provides a concrete example of a non-reflective $\mathbb{C}\mathbb{S}$ -monomorphism.

Example 5.5 Non-reflective $\mathbb{C}\mathbb{S}$ -Monomorphisms

Consider a category of comprehensive systems $\mathbb{C}\mathbb{S}(n, \mathbb{B})$ with $n = 2$ and $\mathbb{B} := \mathbf{1}$ being a terminal object in \mathbf{Cat} . Further let L and R be two comprehensive systems with $L_1 = R_1 = \{a\}$, $L_2 = R_2 = \{b\}$, and $L_0 = R_0 = \{c\}$. The projections are defined as follows: $p_1^R(c) = a$, $p_2^L(c) = p_2^R(c) = b$, and $p_1^L(c)$ is undefined. Now let $m : A \rightarrow B$ be a comprehensive system morphism that is component-wise the identity. This morphism is monic but not reflective since definedness of p_1^R is not reflected.

Next, I have to show that \mathcal{M} is admissible.

Proposition 13

The class of all reflective monomorphisms \mathcal{M} is *admissible*.

Proof. The complete proof is given in Appendix B.4. □

Proving Theorem 12 requires to verify the existence of pushouts (where some morphisms of the pushout diagram belong to the special class \mathcal{M}) in our category $\mathbb{C}\mathbb{S}$ (or \mathbb{M} equivalently) and to check whether pushouts have the so-called (*weak*) *van Kampen* property, see Def. 5.23. The latter enforces a well-behaved interplay between pushouts and pullbacks.

Tobias Heindel, in his PhD thesis [226], showed that it equivalently suffices to show the existence of (i) pushouts along \mathcal{M} -morphisms, (ii) \mathcal{M} -*partial-arrow classifiers*, and that (iii) pushouts are preserved by pullbacks. This is the strategy I am going to use to prove Theorem 12.

Now, one can show the existence of pushouts along \mathcal{M} -morphisms, i.e. for spans where one of the legs is a reflective $\mathbb{C}\mathbb{S}$ -monomorphism.

Theorem 14 Pushouts in $\mathbb{C}\mathbb{S}$

$\mathbb{C}\mathbb{S}$ has pushouts along \mathcal{M} morphisms.

Proof. The complete proof is given Appendix B.5. □

The next part of the proof, following Heindel’s approach, concerns *partial arrow classifiers*. Intuitively, a partial arrow classifier adds a substructure to a given object that represents “error” (failed computations or unmappable elements), similar to the `java.util.Optional` data type in Java or the `Maybe-monad` in Haskell. In the category \mathbf{Set} , the partial arrow-classifier $\mathcal{L}A$ applied to a set A adds the new element \perp to the contents of A . More details on partial arrow classifiers can be found in Appendix C.3.

Theorem 15

\mathbb{CS} has \mathcal{M} -partial arrow classifiers.

Proof. The complete proof is given in Appendix B.6. \square

In the context of weak adhesive HLR categories, this construction becomes relevant because it turns out to represent a right-adjoint to the *graphing functor* Γ (Def. 5.12). This guarantees that pushouts are *hereditary*, a property closely related to the weak van Kampen (Def. 5.22) property [227], which was originally introduced in [272].

The final ingredient is stability of pushouts under pullbacks, which corresponds to the “ \Leftarrow ”-direction in the definition of van Kampen squares (Def. 5.22).

Theorem 16

Pushouts along \mathcal{M} -morphisms in \mathbb{CS} are stable under pullbacks, i.e. when (n, g) is the pushout of (f, m) in (5.25) and all vertical faces (front, back, left, right) are pullbacks then also (n', g') is the pushout of (f', m') .

$$\begin{array}{ccccc}
 & & A' & \xrightarrow{m'} & B' \\
 & f' \swarrow & \downarrow n' & & \swarrow g' \\
 C' & \xrightarrow{\quad} & D' & & \\
 \downarrow c & & \downarrow a & & \downarrow b \\
 & & A & \xrightarrow{m} & B \\
 & f \swarrow & \downarrow d & & \swarrow g \\
 C & \xrightarrow{\quad} & D & \xrightarrow{n} &
 \end{array} \tag{5.25}$$

Proof. Using Theorem 8 the proof can be performed in \mathbb{M} . Next, recall that pullbacks and pushouts in \mathbb{M} are constructed component-wise in \mathbb{G} as demonstrated in the proofs of the Theorems 9 and 14, see Appendices B.2 and B.5. Hence, the picture in (5.25) can be disassembled into a \mathbb{I} -indexed family of instances (5.25)-diagrams in \mathbb{G} . Using the fact that pushouts are stable under pullbacks in \mathbb{G} [304], stability of pushouts under pullbacks holds for each component. The resulting family of pullbacks/pushouts in \mathbb{G} can be re-assembled into pullbacks/pushouts in \mathbb{M}/\mathbb{CS} . \square

5.3.3 Triple Graph Grammars and Graph Diagrams

TGGs [422] represent a powerful and well-established rule-based approach to multi-model repair, see Sec. 4.4.3. Triple graphs are closely related to comprehensive systems; both of these concepts are based on graph-like structure and their formulation is given in categorical terms. Triple graphs can be defined as a functor category $\mathbb{C}^{\mathbb{D}_{\text{TGG}}}$, with \mathbb{C} being a suitable weak adhesive HLR category and \mathbb{D}_{TGG} being the small *shape* category, depicted in (5.26) (identities are omitted):

$$1 \xleftarrow{01} 0 \xrightarrow{02} 2 \tag{5.26}$$

Formalisation

Thus, the solution space is limited to binary scenarios. Trollmann and Albayrak [463, 464] generalised the TGG framework to cope with multiple models within a *graph diagram* (GD) framework. The idea is to allow for different shapes \mathbb{D} , which must satisfy the condition that the set of objects can be divided into two disjoint sets of *models* N and *relations* R , i.e. $|\mathbb{D}| = R \sqcup N$. All non-identity morphisms are required to have a domain in R (relations) and codomain in N (models). Further, there is at most one arrow in $\mathbb{D}(r, m)$ for fixed $r \in R$ and $m \in N$. In such a way, graph diagrams, i.e. functors $D : \mathbb{D} \rightarrow \mathbb{C}$, can specify relations of different arities. Graph diagrams (GD) subsume TGGs, with $R = \{0\}$ and $N = \{1, 2\}$.

They are, however, *static*: If $r \in R$ has k outgoing morphisms with targets $m_1, \dots, m_k \in N$, $D(r)$ is a k -ary correspondence relation with representatives which relate to exactly one element in each of the k models $D(m_j)$. Thus, one must distinguish relations of different arity in the underlying shape for graph diagrams. In larger system landscapes ($n > 3$), there may be many more commonality relations of arbitrary arity $k \leq n$, which would cause a considerable amount of heterogeneity in the underlying shape for graph diagrams. Moreover, the graph diagram shape and hence the basic setting for implementations must be altered, each time new commonality relations are added.

I will show that my framework is more general than graph diagrams $\mathbb{G}^{\mathbb{D}}$ for the case that \mathbb{C} is a category of graph-like structures ($\mathbb{C} = \mathbb{G} = \mathbf{Set}^{\mathbb{B}}$) in that there is an embedding functor $T : \mathbb{G}^{\mathbb{D}} \rightarrow \mathbb{CS}$. The latter further preserves pushouts, which model derivations in Graph Diagram Grammars (GDG). Hence we are able to replay all TGG/GDG-computations in comprehensive systems, yet being able to cope with new relations *without* changing the schema category if the parameter n in Def. 5.19 is large enough. The construction is based on *coproducts* and thus I first prove a another result for comprehensive systems.

Proposition 17 Coproducts and Initial Object in \mathbb{CS}

The category \mathbb{CS} possesses all *binary coproducts* (Def. C.11) and an *initial object* (Def. C.10).

Proof. The complete proof is given in Appendix B.7. □

In the following, I write $\coprod_{i \in I} D_i$ to denote the coproduct of a collection $(D_i)_{i \in I}$ of \mathbb{G} -objects. Note that a collection $(f_i : D_i \rightarrow D)_{i \in I}$ of morphisms yields the morphism $\coprod_{i \in I} f_i : \coprod_{i \in I} D_i \rightarrow D$ by the universal property of coproducts, i.e. the morphism, which acts as f_i on each D_i . Further, we introduce a shorthand notation: $\mathbb{C}^{\rightarrow}(_, B) := \{f \in \mathbb{C}^{\rightarrow} \mid \text{codom}(f) = B\}$, i.e. the objects of the slice category $\mathbb{C} \downarrow B$.

By Theorem 8, it suffices to define a functor from $\mathbb{G}^{\mathbb{I}}$ to \mathbb{M} . The composition of this functor with the equivalence yields the desired result. This functor will also be called T .

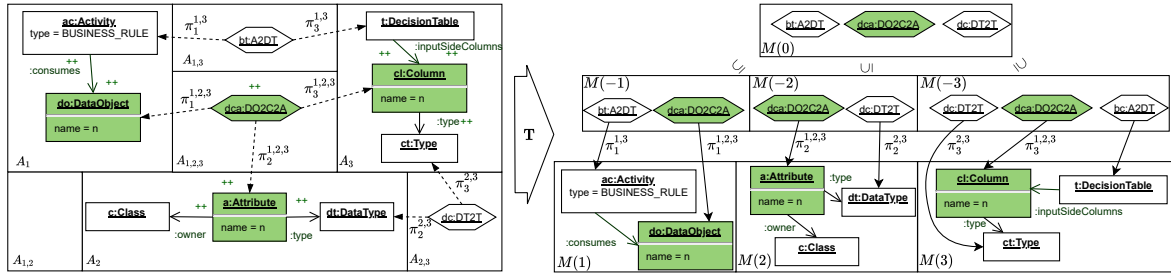


Fig. 5.17: Graph Diagram production rule

Definition 5.25 Translation Functor T

Let a schema category \mathbb{D} for graph diagrams be given with $|\mathbb{D}| = R \sqcup N$ and let n be the cardinality of N . Without loss of generality, I assume $N = \{1, \dots, n\}$. Let D be a graph diagram, then I define a multi-model span $\mathcal{M} := T(D)$ intuitively as follows (recall the schema in (5.7)): The model components of $\mathcal{M}(j)$ ($j \in N$) are the same as those of D , the commonality specification $\mathcal{M}(0)$ is the disjoint union of all relations in D , the middle objects $\mathcal{M}(-j)$ are the union of those relations, the model $D(j)$ participates in:

$$\begin{aligned} \mathcal{M}(j) &:= D(j) && \text{(Models are untouched)} \\ \mathcal{M}(0) &:= \coprod_{r \in R} D(r) && \text{(Coproduct of all relations)} \\ \mathcal{M}(-j) &:= \coprod_{f \in \mathbb{I} \rightarrow (-, j)} D(\text{dom}(f)) && \text{(Participating Relations of } D(j)) \end{aligned}$$

for all $j \in \{1, \dots, n\}$. Furthermore,

$$\begin{aligned} \mathcal{M}(jj) &= \coprod_{f \in \mathbb{I} \rightarrow (-, j)} D(f) && \text{(Projections)} \\ \mathcal{M}(j0) &: \coprod_{f \in \mathbb{I} \rightarrow (-, j)} D(\text{dom}(f)) \hookrightarrow \coprod_{r \in R} D(r) && \text{(Domains)} \end{aligned}$$

Morphisms $\mathcal{M}(jj)$ are the unions of the domains of those morphisms that have target $D(j)$ and inclusions arise from the fact that coproducts in the above definition of $\mathcal{M}(-j)$ (taken over some relations) are always subgraphs of the complete coproduct $\mathcal{M}(0)$ (which is taken over all relations).

The definition of T on arrows is straightforward and we give it only informally: If $n : D \Rightarrow D'$ is an arrow (natural transformation) between graph diagrams, then (1) $T(n)_i$ is a morphism which acts in the same way as n_i on $D(i)$, if $i > 0$, (2) it amalgamates the actions of n on relations, if $i = 0$, which (3) naturally restricts to the respective actions, if $i < 0$. It is then easy to see, that $T(n)$ is a natural transformation.

I illustrate the construction in Def. 5.25 at the example of a graph diagram production rule depicted in the left side half of Fig. 5.17. The figure shows a production rule $r : B \hookrightarrow A$ in an integrated way, where B (before) and A (after) are graph diagrams ($A, B \in \mathbb{G}^{\mathbb{D}}$). A contains all elements shown in Fig. 5.17 and B contains only those elements, which are not shaded and missing the ++-annotation, compare Fig. 4.19 in Sec. 4.4.3. The set of models in \mathbb{D} is a three element set: $N = \{1, 2, 3\}$ representing the three model spaces for BPMN, UML and DMN. The relation set in \mathbb{D} contains four

Formalisation

elements: $R = \{(1, 2), (2, 3), (1, 3), (1, 2, 3)\}$, representing all binary relations between the three model spaces and the ternary relation between all of them. Elements of R are tuples and morphisms in \mathbb{D} are projections $\pi_N^R : R \rightarrow N$. This schema is visualized by compartments in Fig. 5.17, where each compartment depicts a graph (object in \mathbb{G}), i.e. the image of $A(x)$ ($B(x)$) for an $x \in |\mathbb{D}|$. I introduce the notation: $G_x := G(x)$ and if x is a tuple we may omit parentheses. The application of the translation functor T on A will produce a multi-model span \mathcal{M} with degree $n = 3$, which is depicted in the right side half of Fig. 5.17. The graphs $\mathcal{M}(j)$ are identical to A_j ($1 \leq j \leq 3$). The commonalities graph $\mathcal{M}(0)$ is the coproduct (disjoint union) of $A_{1,2}$, $A_{2,3}$, $A_{2,3}$ and $A_{1,2,3}$, i.e. the nodes $\{bt, dca, dc\}$. The domain of definition $\mathcal{M}(-1)$ for the projection on component 1 is the coproduct of $A_{1,3}$, $A_{1,2}$ and $A_{1,2,3}$, i.e. the nodes $\{bt, dca\}$. The other domains of definition are constructed accordingly. The mappings $\mathcal{M}(j0)$ are simply the inclusion between coproducts, and $\mathcal{M}(jj)$ are given by universal coproduct property ($j \in \{1, 2, 3\}$): Taking the union of all arrows from all relations into component j , see Fig. 5.17. Since \mathbb{M} is equivalent to \mathbb{CS} , there is a respective representation in terms of a comprehensive system. This system is presented in a similar way as in the right hand side of Fig. 5.17 with the only difference that the middle components ($\mathcal{M}(-j)$) are replaced by partial projection mappings.

Theorem 18

Functor $T : \mathbb{G}^{\mathbb{D}} \rightarrow \mathbb{CS}$ is an embedding and preserves pushouts.

Proof. See Appendix B.8. □

Thus, as a consequence:

Corollary 19

Every sequence of rule applications in $\mathbb{G}^{\mathbb{D}}$ has a unique representation of corresponding rule applications in \mathbb{CS} and hence can be replayed in the general framework of comprehensive systems.

As a conclusion, this section showed that “everything” which is possible with to do with graph grammars and the framework of algebraic graph transformation is possible to do with comprehensive systems as well. In particular, search-based (graph constraints) and rule-based (consistency preserving rules) repair.

5.4 Summary & Future Directions

Let me summarise the content and contributions of this chapter: First, presheaves – rebranded under the name graph-like structures (Def. 5.2) – were (re)discovered as a suitable formal interpretation for software models. Secondly, multi-ary “star-shaped” spans of partial morphisms (Def. 5.15) were identified to capture the essence of multi-models. The latter further admit an “integrated representation” (Theorem 8) in the form of comprehensive systems whose build up is similar to graph-like structures. Finally, it was shown that the resulting formalism can be embedded into abstract frameworks

for consistency verification (i.e. institutions, Corollary 10) and model repair (i.e. weak adhesive HLR categories, Theorem 12). Hence, the research questions RQ1–RQ3 are addressed on a formal level and also the limitations from Sec. 4.5 are addressed since comprehensive systems support general n -ary ($n \geq 2$) correspondence relations and simultaneously provide a formal foundation for multi-model consistency management. Nonetheless, the theory on comprehensive systems is still in its infancy and there are several directions for extending this theory, which are sketched below.

DERIVATION OF REPAIR RULES The validity of Theorem 12 guarantees that we can employ the framework of algebraic graph transformation for the implementation of model repair. A promising concrete strategy for utilising this framework is based on the idea of decomposing *consistency-preserving rules* into *edit* and *repair rules* [453]. The latter is based on the rule-decomposition theorem [206] and is simultaneously the underlying idea behind the operationalisation of TGGs [144]. I have also shown that comprehensive system grammars represent a generalisation of graph diagram grammars and TGGs (Corollary 19). Hence, the operationalisation procedure that splits a production rules into rules acting only on a single component (edit) and rules that propagate to the remaining components including commonalities and projections (repair) can seamlessly applied. In addition, the schema of comprehensive systems allows for even more possibilities of rule splitting, e.g. parallel edits on multiple components.

However, the manual definition of comprehensive system grammars can be laborious because they may have to consider comprehensive systems with many components. It will be interesting to investigate whether rules can, under certain conditions, be defined for single graph-like structure (\mathbb{G}) and then lifted to the level of comprehensive systems (\mathbb{CS}), possibly exploiting a colimit construction, see Sec. 5.2.1. Hence, further research on the definition of consistency rules and derivation of repair rules in the category of comprehensive systems is necessary.

DATA TYPE ALGEBRAS Structural modeling languages generally distinguish between *objects* (dynamic; arise during run-time) and *values* (static; completely known at design-time) and thus also between *references/links* (connecting objects) and *attributes* (connecting objects and values). This provided the rationale for introducing *E-graphs*, see Fig. 5.2c. Technically, E-graphs alone do not suffice to model attributes and values faithfully. The first reason is that the names of value nodes are important while names of object nodes are not, see Remark 5.2. Pushouts, which are used to model changes on comprehensive systems, are only defined up to isomorphisms and therefore they are theoretically allowed to rename the value 23 to 42, which is undesired in practice. The second reason is that one wants to do computations on attributes, e.g. a transformation rule that increments the age value of a Person-object. Both aspects can formally represented by algebras [415] via *constants* and *multi-ary operations* specified by an equational theory. This has been well-investigated in the research domain of *algebraic specification*. The representation discrepancy between the graph part (objects and links) versus the data/algebra part (values and attributes) of a model was discussed by the authors in [319], who propose to represent models rather by means of partial algebras. A popular approach in the graph transformation community is to consider *attributed*

graphs [151], which combine (E-)graphs with algebras, i.e. the set of data nodes DN (see Fig. 5.3c) is required to be the disjoint union of carriers sets of a respective data type algebra. Attributed graphs were actually one of the main reasons for the introduction of weak adhesive HLR categories [150] because categories of algebras are generally not adhesive¹², i.e. one has to restrict themselves by considering the special class of attributed graph morphisms that are isomorphisms on the data part.

For the examples of thesis, it will be enough to simply pose the convention the pushouts and other universal constructions are not changing the names of elements in the carrier set of DN when working with E-graph representations since I will not consider rules with calculations on attributes. But for the future it will be important to also include a proper treatment of values and attributes into the picture. My conjecture is that all the propositions and theorems in this chapter still hold when replacing the base category \mathbb{G} with attributed graphs. Another, slightly different approach that will be interesting to look at more closely was proposed by Schultz et al. [420], who formalise instances of database schemas via (algebraic) profunctors connecting a graph-schema with a data type algebra. Hence, treatment of (base type) values and attributes poses another important future research domain.

DIAGRAMMATIC OPERATIONS The structural commonalities relationships inside comprehensive systems relate elements with each other. In order to relate a feature (node, edge, ...) with a feature in another model, there must be a corresponding element of the same type in that model. In practice, the definition of correspondence relationship involves intermediate computations. A classical example is that one model represents the name of a person with two fields `firstname` and `lastname` whereas another model stores the same information in a single field `fullname`. A `Person`-object in the former model is related to a `Person`-object in the latter model if the result of the string-concatenation of `firstname` and `lastname` equals the value of `fullname`. This cannot immediately be expressed via a multi-model span since the definition of the respective projection morphism will refer to a *derived element*. Diskin et al.[129] presented an approach for formally representing such relationships with the help of “*Kleisli-categories*”. The latter is based on the concept of *diagrammatic operations*, i.e. the operation-counterpart to the diagrammatic predicated introduced in Sec. 5.1.2. The general topic of diagrammatic operations requires more theoretical work before it can be applied to comprehensive systems. First attempts in this directions have been started in [495] in the form of *graph operations* and the technical report [131].

GENERALISATION OF COMPREHENSIVE SYSTEMS Note that the definition of multi-model spans and thus also the definition of comprehensive systems is indexed by a natural number n (= the maximal number of domains under consideration). Hence, comprehensive systems can be expressed by a functor¹³ (in **CAT**)

$$\mathbf{Nat} \xrightarrow{\mathbf{CS}} \mathbf{Cat}$$

¹²[322] contains some elucidating examples on this matter

¹³Recall that the total order of natural numbers **Nat** is a category as well.

In practice, this technical detail has minor significance since the number of domains under consideration is finite number and therefore one simply chooses an n that is sufficiently big. From the theoretical stance it will be elegant to generalise comprehensive systems by considering \mathbb{CS} the directed colimit (in **CAT**) of all categories of comprehensive systems for $n = 1, n = 2, n = 3, \dots$ and so on.

SYMMETRIC MONOIDAL CATEGORIES The latest “hot topic” in the applied category theory community [177] are *symmetric monoidal categories (SMCs)*, which comprise the notations of *sequential* (\circ) and *parallel* composition (\otimes). The big benefit of these categories is that they offer an intuitive but fully formal visual representation in the form of *string diagrams* [425], which resemble block diagrams known in other Engineering disciplines. It is important to note that the category of *comprehensive systems* possess coproducts and initial objects. Further, initial objects and coproducts always induce a symmetric monoidal structure on the underlying category. Hence, comprehensive systems give rise to a SMC. But further work around alternative constructions and the concrete implications is required.

“Es gibt nichts Gutes, außer man tut es. (germ. lit.:
Actions speak louder than words.)”

—Erich Kästner

CHAPTER 6

IMPLEMENTATION

This chapter presents my *practical* contribution towards the issue of multi-model consistency management i.e. a prototypical tool called CORR_{LANG}. This tool is based on the theoretical framework from Chap. 5, and embodies an approach of multi-model consistency management based on global views, see Sec. 3.3.5. The technical foundation for this tool had been laid in the context of a Master Thesis [475] written by Ole von Barga, a student that I supervised in Fall 2019. The result of his thesis was a tool called GraphQLIntegrator¹, which generates a federated GraphQL web service on top of other GraphQL web services. This work was based on the colimit-based model merging framework presented in Sec. 5.2.1 and has been presented in an article [447], which is part of the proceedings of the ECMFA2020 conference.

GraphQLIntegrator provided the technical foundation for CORR_{LANG}, which arose from a merger of the code bases of the former and a Xtext²-based prototype implementation³ of comprehensive systems, mentioned in [446] and [451]. Compared to GraphQLIntegrator, CORR_{LANG} represents a substantial extension that adds support for MDSE frameworks (i.e. EMF), alternative model alignment possibilities and basic consistency management functionality. It is publicly available under an open-source licence⁴ but has not been presented in other publications yet. Therefore, this thesis represents the first scientific presentation of CORR_{LANG}. The structure of this chapter follows the historical development of the tool in the form of iterations. I begin with the first iteration (i.e. GraphQLIntegrator) before adding model management functionality in the second iteration, and finally consistency management functionality in the third iteration. Speaking of research questions, the first iteration (Sec. 6.1) deals with multi-model *representation* (RQ₁), the second iteration (Sec. 6.2) focuses on the underlying *architecture* (RQ₄), and the third iteration (Sec. 6.3) deals with *verification* (RQ₂). Consistency *restoration* (RQ₃) is mostly left for future work (Sec. 6.4).

¹<https://gitlab.com/olevonbarga/graphqlintegrator>

²<https://www.eclipse.org/Xtext/>

³<https://github.com/webminz/comprehensivesystems-emf-prototype>

⁴<https://github.com/webminz/corr-lang>

6.1 First Iteration: GraphQL Federation

6.1.1 Background: Web Services & GraphQL

Web services (WSs) have been introduced in Sec. 1.3.1 as one of the main approaches to facilitate system integration (as instance of Remote Procedure Calls) and are essential for designing *microservice architectures* [183] or *service-oriented architectures* [262]. Each system is abstractly conceived as set of *operations*, which clients can call through HTTP requests. In general, there are two ways for implementing a WS: *Simple Object Access Protocol (SOAP)* [498] and *REpresentational State Transfer (REST)* [172]. The SOAP-approach requires that all operations and associated data types are defined in a schema beforehand. This has the advantages of “type-safety” and “discoverability” during design- and runtime. The main disadvantage of SOAP is seen in its *complexity* because this protocol comprises various concepts (e.g. sessions, transactionality, and authorisation) such that constructing SOAP-messages is accompanied with a significant overhead, which often necessitates code generation. REST on the other hand abstains from a concrete schema definition and instead encodes operations via conventions, i.e. the combination of HTTP-method and URL-path identifies an operation. For example, “GET /customers” identifies a query-method retrieving customer-objects whereas “PUT /customers/23” identifies a method that updates the customer-object with id=23. The type of the input and output parameters is not explicitly defined. Thus, the main advantage of the REST-approach is less-overhead and higher flexibility. The lack of an explicit schema is the main disadvantage of this approach since it is a potential source for runtime errors so that REST relies solely on “developer discipline”. An analogy for the relationship between SOAP and REST can be seen in the dichotomy between statically and dynamically typed programming languages.

GraphQL [161] is a novel approach for developing WS, which is designed to mitigate the disadvantages of SOAP and REST. It was originally developed by Facebook for their own applications before it was eventually open sourced in 2015. Since then, it has been adopted by other companies such as Twitter, Airbnb, and Github, also there are first use cases in academia, see e.g. [349, 466]. GraphQL seeks to combine the advantage of an explicit schema (without the overhead found in SOAP) with the flexibility of REST by allowing clients to perform arbitrary queries over the schema. The schema is defined in terms of *entity types*, which represent the underlying domain model and abstractly form a *graph*, compare Sec. 5.1.1. This definition is given in a textual *Schema Definition Language (SDL)*. To illustrate this, Fig. 6.1 depicts a GraphQL schema for an information system storing *sales* data of a hypothetical retail company.

The keyword *type* initiates the declaration of a so-called *object type* (think *class* in UML, and see Fig. 6.3). An object type has a name and comprises multiple *fields*, which again have a name and a *type*, which is either a *scalar type* (think *data type* in UML) such as `String`, `Int` or another object type. Each field must be implemented by a so-called *resolver*-function in the system behind the schema. Given a *context* object (which represents an instance of the object type container for the field), this function must return a value or a list of values. This enables the system developers to apply intermediate domain specific logic and to implement an arbitrary persistence layer of their choice. The type of a field can be marked as

```

type Query {
  customer(customer: ID!): Customer
  allCustomers: [Customer]
  store(store: ID!): Store
  allStores: [Store]
}
type Mutation {
  createCustomer(name: String!, email: String): Customer!
  updateCustomer(customer: ID!, name: String, email: String): Customer
  deleteCustomer(customer: ID!): Customer
  setAddress(customer: ID!, street: String, city: String, postalCode: String, state: String,
country: String): Customer
  createPurchase(customer: ID!, date: String!, store: ID!): Purchase
  addItemToPurchase(purchase: ID!, product: ID!, quantity: Int): PurchaseItem
}

```

(a) Facade Part

```

type Customer {
  id: ID!
  name: String!
  email: String
  address: Address
  purchases: [Purchase]
}
type Purchase {
  id: ID!
  date: String!
  customer: Customer!
  store: Store!
  items: [PurchaseItem]
}

```

(b) Types (Part 1)

```

type PurchaseItem {
  productID: ID!
  quantity: Int
}
type Store {
  id: ID!
  manager: ID!
  purchases: [Purchase]
  location: Address!
}
type Address {
  street: String
  city: String
  postalCode: String
  state: String
  country: String
}

```

(c) Types (Part 2)**Fig. 6.1:** Sales System GraphQL Schema

Implementation

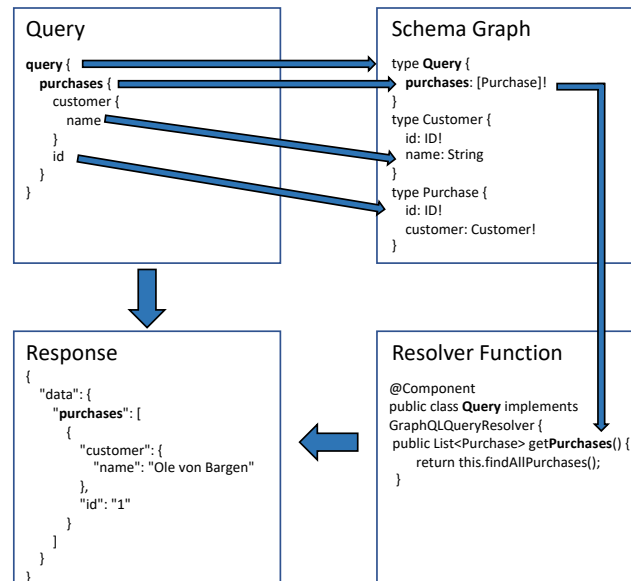


Fig. 6.2: GraphQL Query Mechanism explained

- i *mandatory* (denoted by an exclamation mark !), i.e. the resolver will provide *exactly one* value,
- ii *list-valued* (denoted by brackets []), i.e. the resolver will provide *a list of values*, and
- iii *optional* (default), i.e. the resolver will provide *one* value or null.

Furthermore, a field may have a list of *arguments*, which are passed on to the respective resolver. An argument has a name and a scalar type. Every schema may contain the special entity types Query and/or Mutation, see Fig. 6.1a. These types represent *singletons* [191] and act as *facades* [191], i.e. their fields represent the operations offered by the WS. The fields of the Query type are intended to be side-effect-free functions (retrieval), whereas fields of the Mutation type are meant to change the state of the underlying database (modification). In a typical architecture, the facade-level functions are implemented via retrieval from a persistence backend (see the *Repository* design pattern in [160]) while the resolvers for the fields of the remaining object types are implemented as simple “property-getters”. A schema together with a set of resolver-functions (one for each field) constitutes an *endpoint*.

A client interacts with an endpoint by sending a *query* to it. A query represents a tree of field names, optionally containing *value assignments* for the arguments of the field. The root of this tree is a reference to either Query or Mutation. Recall that these types are singletons such that there is always an instance of these types forming an “entry point” to system’s data. Compared to other query languages such as SQL, GraphQL only features “projection”. Filtering and aggregation has to be implemented in a respective resolver implementation.

The GraphQL-engine takes care of parsing the query, calling the resolvers accordingly, and assembling the individual results into a single *document*. Usually, this document is technically represented as JSON. A schematic picture of the query-mechanism is shown in Fig. 6.2.

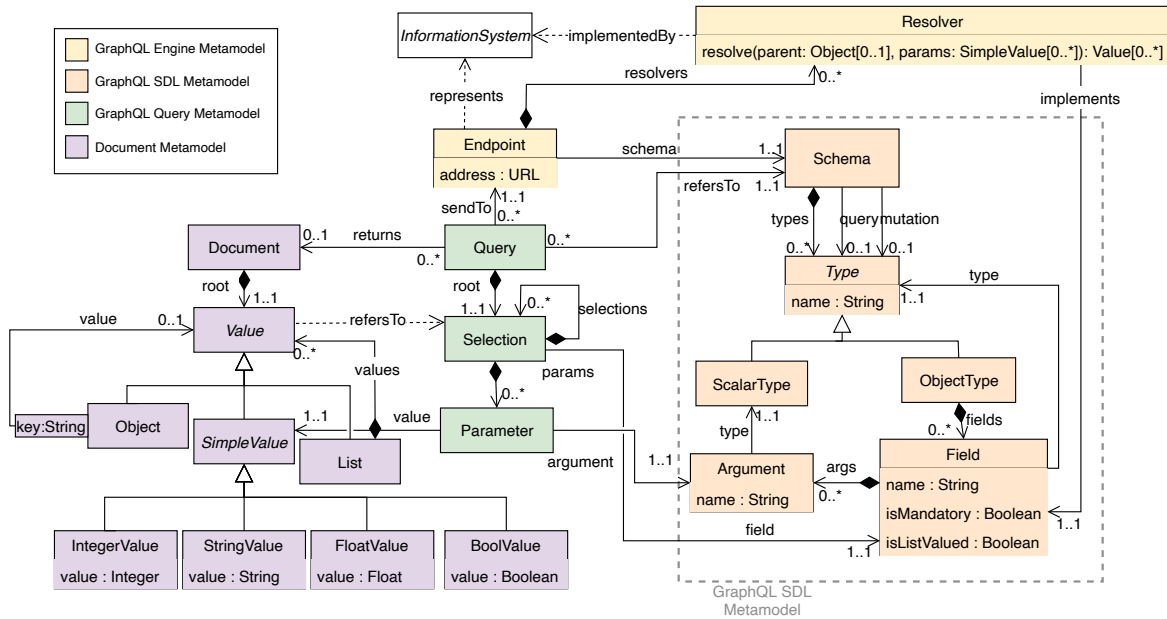


Fig. 6.3: Metamodel of GraphQL concepts

Fig. 6.3 depicts a metamodel, which summarises the GraphQL concepts. The highlighted fragment represents the metamodel of the GraphQL SDL, which can be seen as textual DSL. There is a strong resemblance between this DSL and class diagrams or MOF-models. Therefore, in the following, I will use the graphical syntax of the latter for depicting GraphQL-schemas in a more compact and visual way. Every (meta-)class corresponds to an object type, every attribute to a field with a scalar type, and every directed reference to a field with an object type. The multiplicity 1..1 indicates the respective field being mandatory (!), and 0..* indicates the field being list-valued ([]). From the conceptual viewpoint, each schema can be interpreted as a metamodel. Its model space are all possible documents resulting from query executions, which is technically induced by the resolver implementations.

6.1.2 Problem Statement: Federation

GraphQL facilitates a decoupling between the frontend and the backend, i.e. when developers on both ends have agreed on the common schema, they can develop queries and resolvers independently of each other. However, system landscapes in organisations comprise multiple systems (endpoints). This is reinforced by contemporary software architecture approaches such as microservices. Heterogeneous system landscapes imply a higher level of complexity.

Let me illustrate this with a small example of a hypothetical retail company, which operates three information systems. Fig. 6.4 depicts this system landscape comprising the three systems *Sales* (introduced in the previous section), *Invoice*, and *HR*, which are accessible through the GraphQL endpoints EP₁, EP₂ and EP₃. Let us assume that this company faces several challenges: First, there is a growing problem of *redundant data* and thus possibly inconsistent data, e.g. in the past there had been issues where customers relocated to a new address but the records had not been updated consistently such that invoices were not delivered to the right customers. Secondly, there is a difficulty

Implementation

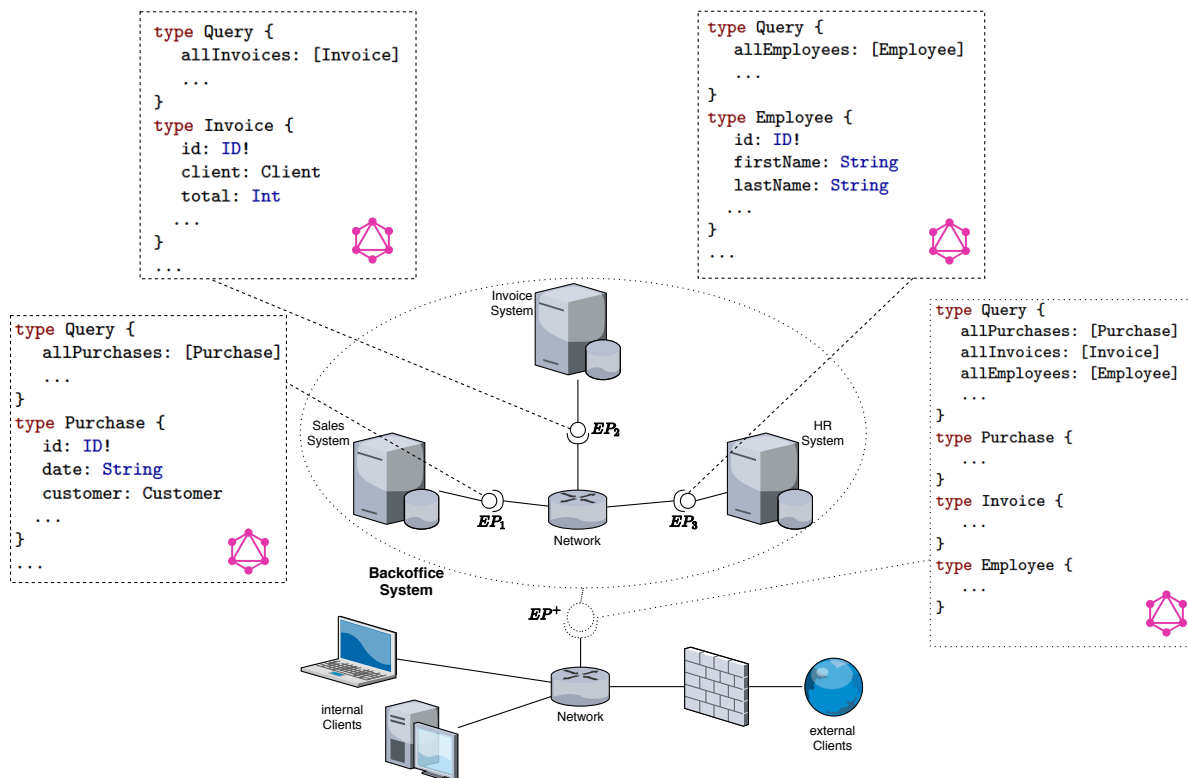


Fig. 6.4: GraphQL Federation: an overview

in implementing *new use cases*, e.g. the company wants to implement a policy, which gives their employees discounts on purchases in the company's stores. The latter requires to align the data set of the HR system with the two other systems. Thirdly, the company's CEO is interested in the *reporting and analysis* of the data contained in the information systems. These three issues are examples of the most common reasons for data integration, see [136].

Physically integrating all systems behind a single endpoint would require a lot of manual effort and it is not always possible either, especially if a system has been bought from an external vendor. Thus, another possibility is to integrate the systems *logically*, i.e. they remain independent but there is an additional logical integration layer on top of them. In the database domain, this idea is called *federation*. Applied to the situation in Fig. 6.4, a federation of the three systems results in a new virtual GraphQL endpoint EP⁺ referred to as the *virtual "Backoffice"* system. For the outside it will appear as any other GraphQL endpoint and internally it is implemented by delegating the respective queries to the individual systems. The latter requires to align the heterogeneous schemas and consolidate records representing the same information.

The federated endpoint addresses the practical issues mentioned above: A unified data set allows to detect address duplicates, to identify persons who are both customers and employees, and to facilitate global reporting. The implementation of federated systems, in practice, often happens in an ad-hoc manner [136]. In the following, I want to look into more rigorous and declarative approaches for developing federated endpoints for GraphQL.

6.1.3 Existing Tool: Apollo Federation

The most popular *JavaScript* implementation of GraphQL is *Apollo GraphQL*⁵. This framework is accompanied by several custom extensions of the GraphQL specification [161]. One of them is a tool called *Apollo Federation* [343], which enables organizations to split the definition of a single endpoint into smaller definitions and develop them independently of each other in order to support *Separation of Concerns (SoC)*.

Apollo Federation is based on a *language extension* of the GraphQL SDL. The new language features are a new keyword (`extend`) and directives (`@key`, `@external`, and `@provides`). Directives are the GraphQL equivalent of annotations in programming language; i.e. they have no further meaning but can be used by a concrete GraphQL-implementation to add proprietary features. The keyword `extend` can be put in front of an object type definition. This means that the respective type represents an extension of an object type defined elsewhere. As an example, consider the types `Customer` in *Sales* and `Client` in *Invoices* (see Fig. 6.4). Both types refer to the same real-world concept and therefore we want to align both type definitions with each other. Apollo's type-extension concept allows to realise this by declaring one definition as the original type definition and the other one as an extension (the extension inherits all fields of the original definition and possibly adds additional fields). Thus, there is *single responsibility* requirement, i.e. for every object type (except `Query` and `Mutation`) there must be one responsible origin schema. For the example, let us assume that the Invoice-system provides the original definition and the Sales-system is extending this definition, which is shown in the Listings in Fig. 6.5. Type extension is established via equality of type names and therefore I renamed `Customer` and `Client` to the more neutral term `Partner`.

A type-extension-relationships means that there may be pairs of instances (records) of the related types, which are considered "equal". Apollo Federation enables to consolidate such replicas via a *key* concept. Keys are defined by placing the `@key` directive on the original type definition. There can be multiple `@key`-directives, which represent *key alternatives*. The `@key`-directive refers to one or multiple fields of the respective object type. Thus, allowing to define *composite keys*. Take Fig. 6.5a as an example: `Partners` can be globally identified either by their `id` or their `name` while `Addresses` posses only a single key: the combination all of all fields, i.e. two `Address` objects are identified when the value of all their fields match. The existence of keys requires the responsible endpoint to provide a special resolver, which retrieves the object for given key values. A different endpoint, e.g. *Sales*, can extend the original type definition by placing the `extend`-keyword in front of the definition of an object type with the same name and by choosing one of the key alternatives, e.g. the `Partner` type extension in Fig. 6.5b chooses the key `id`. A type extension may include fields from the original type definition into its own definition by annotating them with the `@external`-directive. Values for the fields marked as `@external` are retrieved by asking the original endpoint, which is possible due to the global identification mechanism.

After having augmented the endpoints with the Apollo Federation features, the endpoints are automatically integrated into a federated endpoint, which aggregates the type definitions from all local endpoints. For our example, this federated endpoint

⁵<https://www.apollographql.com/>

Implementation

<pre>extend type Query { clients: [Client] ... } type Partner @key(fields: "id") @key(fields: "name") { id: ID! name: String address: Address invoices: [Invoice] } type Address @key(fields: "street city postalCode state country") { street: String city: String postalCode: String state: String country: String } ... </pre>	<pre>extend type Query { customers: [Partner]! ... } extend type Partner @key(fields:"id") { id: ID! name: String @external() address: Address @external() email: String purchases: [Purchase] } extend type Address @key(fields: "street city postalCode state country") { street: String city: String postalCode: String state: String country: String } ... </pre>
---	---

(a) Schema Extension: *Invoices*

(b) Schema Extension: *Sales*

Fig. 6.5: Apollo Federation Syntax Extension

comprising *Sales* and *Invoices* will contain a single *Address* type, a single *Partner* comprising the fields *id*, *name*, *email*, *address*, *invoices*, and *purchases*, and there will be a single *Query* type aggregating the query methods from all endpoints. The motivating issue of “inconsistent address updates” is addressed by defining *Invoices* to be authoritative on this matter. Apollo Federation provides a powerful framework to realize SoC by breaking down a single Endpoint into multiple ones but there are some general limitations with this approach:

- Lim#1* The **Single Responsibility** requirement implicitly requires that there is a global underlying conceptual model that every system schema agrees with. Thus, Apollo Federation presumes standardisation (see Sec. 1.3.1) or a canonical data model. There is no way to reconcile “conflicting” schema definitions, i.e. not having declared name and address in Figure 6.5 as `@external` would have caused an error because it would violate the single responsibility requirement.
- Lim#2* Type Extension is a **binary relation**. This has implications on the amount of data that can be aggregated within a single query. A query against a field, whose type is an extended type will aggregate data from the current endpoint and the endpoint, which contains the original type definition. But it is not possible to retrieve data from more than two endpoints at the same time. Considering the example, if we want to include the HR-system into the federation shown in Fig. 6.5 and treat the *Employee* type as a type extension of *Partner* as well, it will not be possible to retrieve *Partner*-objects from the *Sales*-system, the *Invoices*-system the *HR*-system at the same time, only data from EP_1 and EP_2 or EP_2 and EP_3 can be retrieved at the same time.
- Lim#3* **Identification Capabilities are limited**; Apollo Federation allows to identify elements by key alternatives and composite keys but it is limited to field-references.

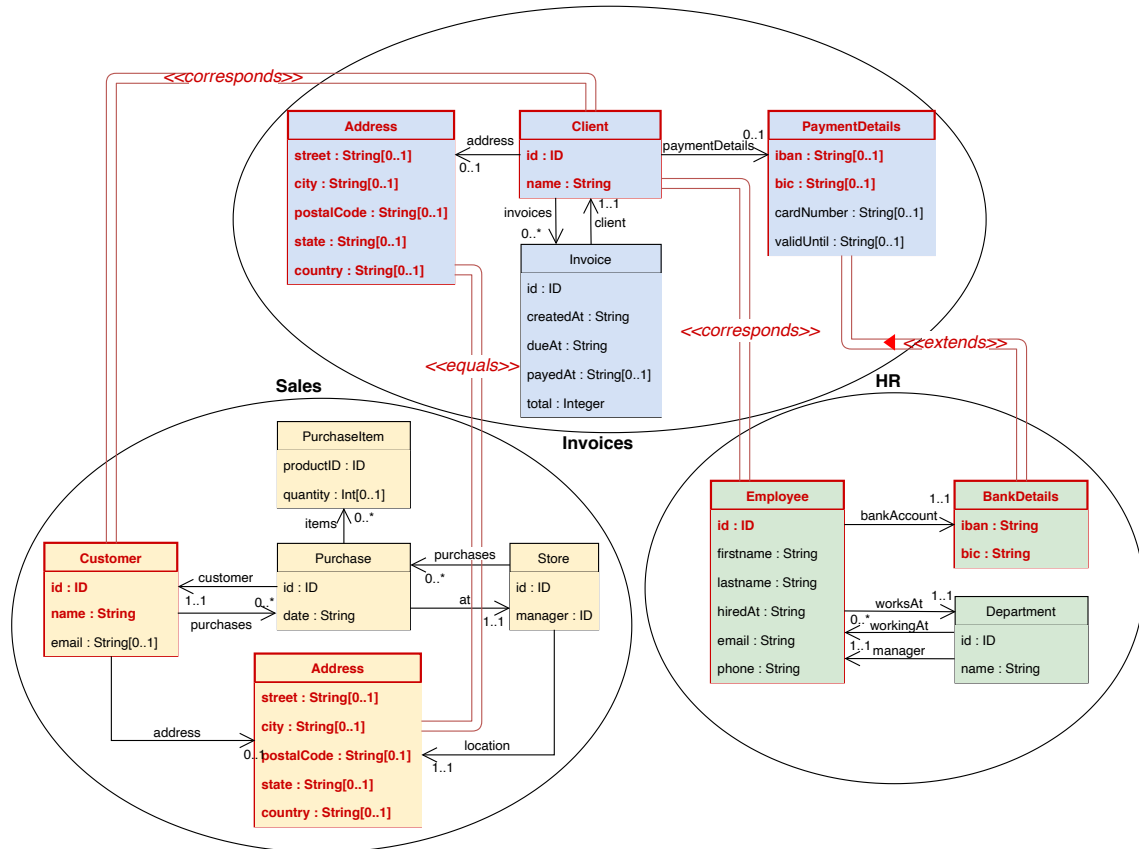


Fig. 6.6: Domain Models of the Systems

There are no possibilities to perform intermediate computations. As an example, imagine identifying a Client/Customer-object with an Employee-object if the name of the Client/Customer is equal to the concatenation of `firstname` and `lastname` of the Employee.

Lim#4 Apollo Federation is **intrusive**, i.e. existing systems must be changed in order to participate in the federation: Their schemas must be modified and they must implement special resolver functions.

Lim#5 Apollo Federation is **technology dependent** by requiring every endpoint to follow and implement a specific protocol. Thus, it only works with a chosen selection of GraphQL implementations, see [343]

6.1.4 Solution Design: Declarative Schema Merging

The goal of this section is to design a general solution for the federation of GraphQL endpoints while addressing the limitations of Apollo Federation.

First, in order to address the main limitations *Lim#1* and *Lim#2*, binary type-extension will be replaced by multi-ary type-merging. Merging is understood as described in Sec. 5.2.1, i.e. formally described as a colimit-object in a suitable category. This relationship generalises type-extension and other types of type-relationships such as sameness, see [116, 134]

Implementation

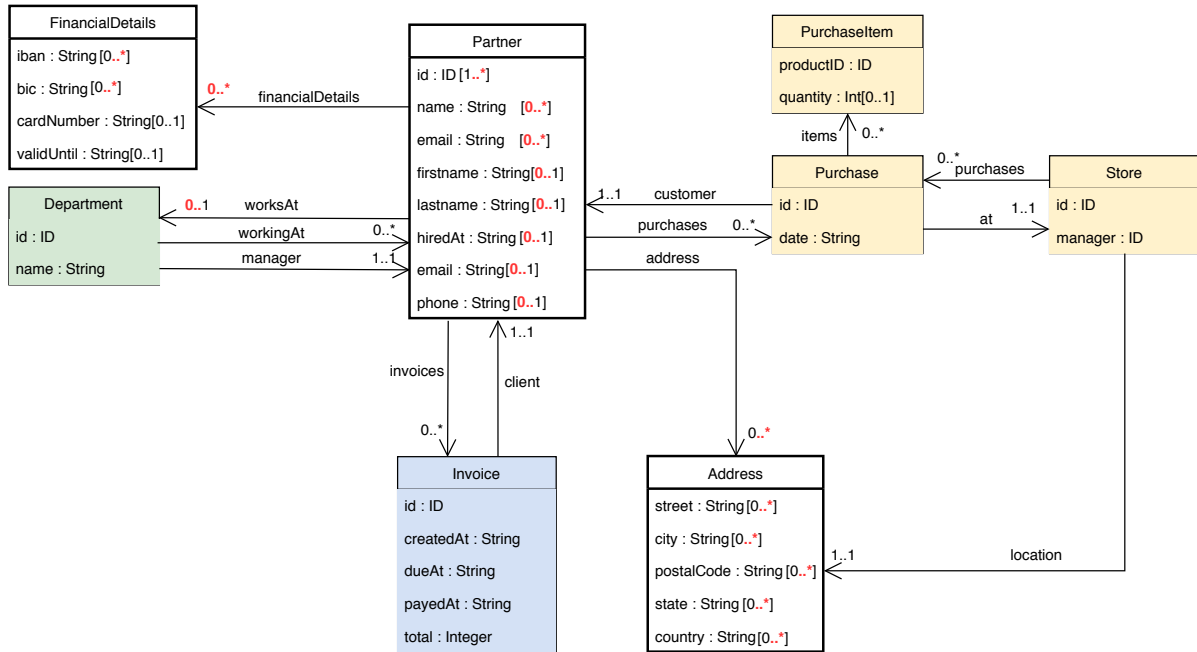


Fig. 6.7: Merged Domain Models for the Federation

For this consider Fig. 6.6, which depicts the schemas of the three systems from Fig. 6.4 in UML class diagram notation. Ovals and different shading indicate the systems' boundaries. Query and Mutation types are not shown but let us assume that there are *Create*, *Read*, *Update*, and *Delete* (CRUD) methods for each object type. There are several relationships among these schemas, which are shown in red: The same type *Address* appears in *Sales* and *Invoices*, the type *PaymentDetails* in *Invoices* extends the type *BankDetails* in *HR*, and there is the relationship between *Customer*, *Client* and *Employee*, which was discussed before. Merging these types along these relationships means that for each of them, the federated schema will contain one merged type, which contains all fields (field with the same name and type are identified) from the types in this relationship. All other types are included in the federated schema without further modifications. The resulting merge is shown in Fig. 6.7. Note that this merge construction changes multiplicities for fields that are identified (highlighted in red) in a liberal manner because a merged type may contain more or less data.

Instances w.r.t. merged object types may be identified as well. Here, the key-concept from Apollo is adopted. Additionally, in order to address *Lim#3*, a concept for *key calculations* is introduced, i.e. keys may not only contain references to fields but also simple operations that are applied to field-values or constants. As a first operation, I will consider "concatenation" as a useful and common operation.

Finally, to address *Lim#4* and *Lim#5*, a declarative approach based on a custom DSL is chosen. The endpoints are treated as black-boxes and will remain totally unaware that they participate in a federation. The custom DSL is used to define the relationships based on which the merge will be conducted and to define the identification rules for instances of such merged elements. Hence, this DSL must offer two features: On the one hand it must allow the definition of multi-ary type and field relationships (*Lim#2*). On the other hand it must allow the definitions of element identification rules supporting composite keys, key alternatives and key computations (*Lim#3*).


```

1 endpoints {
2   "http://localhost:4011/" as sales
3   "http://localhost:4012/" as invoices
4   "http://localhost:4013/" as hr
5 }
6 correspondences {
7   relate {sales.Query, invoices.Query, hr.Query} as Query with {
8     relate { sales.Query.customers, invoices.Query.clients, hr.Query.employees} as partners
9   }
10  relate { sales.Customer, invoices.Client, hr.Employee }
11    as Partner
12    with {
13      relate { sales.Customer.id, invoices.Client.id, hr.Employee.id } as id
14      relate { sales.Customer.name, invoices.Client.name } as fullName
15      relate { sales.Customer.address, invoices.Client.address } as address
16      relate { invoices.Client.paymentDetails, hr.Employee.bankAccount } as financialDetails
17      identify where {
18        or {
19          equals { sales.Customer.id, invoices.Client.id }
20          equals {
21            invoices.client.Name,
22            concat {hr.Employee.firstname, " ",hr.Employee.lastname}
23          }
24        }
25      }
26    }
27  relate { invoices.PaymentDetails, hr.BankDetails } as FinancialDetails
28    with {
29      relate { invoices.PaymentDetails.iban, hr.BankDetails.iban } as iban
30      relate { invoices.PaymentDetails.bic, hr.BankDetails.bic } as bic
31      identify where {
32        and {
33          equals { invoices.PaymentDetails.iban, hr.BankDetails.iban }
34          equals { invoices.PaymentDetails.bic, hr.BankDetails.bic }
35        }
36      }
37    }
38  relate { sales.Address, invoices.Address } as Address
39    with {
40      relate all
41      identify where all equals
42    }
43 }

```

Listing 6.1: Correspondence DSL example

List. 6.1 represents a first proposal of what such a language might look like. It encodes the relationships shown in Fig. 6.6. First, the endpoints participating in the federation have to be specified (lines 2-4). They are identified via their respective URL and for the remainder identified by a symbolic name (keyword `as`) Afterwards, type-relationship (initiated by the keyword `relate`) are defined. There are relationships among `Client`, `Customer` and `Employee` (lines 10-26), among `PaymentDetails` and `BankDetails` (lines 27-37), and the `Address` types (lines 38-42). A type-relationship further contains field-relationships, which are initiated by the keyword `with` and follow a similar structure as type-relationships. Furthermore, relationships can receive a symbolic name (keyword `as`), which defined the name of the resulting type or field in the resulting federated schema. A type correspondence may introduce an identification rule (lines 17-25, 31-36, and 41). An identification is a boolean expression in disjunctive normal form, with literals being equalities among field references or expressions built from operations (e.g. `concat`), constants (e.g. `" "`), and field references (e.g. `sales.Customer.id`). The idea is that a disjunction translates to key alternatives and

Implementation

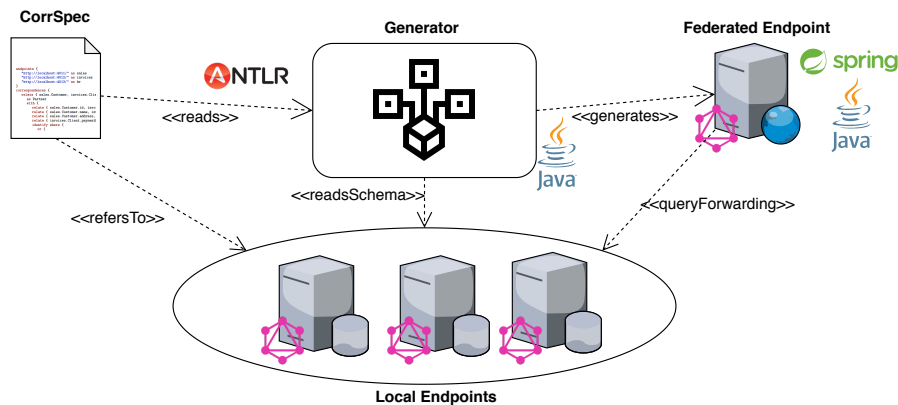


Fig. 6.8: Workflow of the Prototype

a conjunction translates to a composite key. The ability to use arbitrary expressions addresses [Lim#3](#). In [List. 6.1](#) Clients can be identified with Customers via their id and Clients can be identified with Employees by comparing their name with the concatenation of first- and lastname. Finally, I introduce some syntactic sugar (line 40-41) to identify types, which are structurally identical.

Eventually, the declarative specification in [List. 6.1](#) has to be transformed into a running endpoint, i.e. a schema and resolvers for every field in that schema. The technical details of this transformation are described in the following section.

6.1.5 Solution Implementation: GraphQLIntegrator

The conceptual solution described in the preceding section has been implemented in a proof-of-concept implementation⁶. Technical details about this implementation are found in the master thesis [475]. An abstract overview of how this tool works is sketched in [Fig. 6.8](#). First, a textual specification ([List. 6.1](#)) is parsed. Such a declarative specification is called a **CorrSpec**. Parsing is facilitated via *ANTLR* [376]. The **GraphQLIntegrator** follows a code generation approach. Thus, every **CorrSpec** is translated into a Java Web application, which acts as the comprehensive endpoint. The generated application utilises *Spring Boot* and *graphql-java* to leverage the technical implementation details and could be easily exchanged with another technology stack.

The domain model of the generator is shown in [Fig. 6.9](#), where the concepts of the DSL from [List. 6.1](#) are shaded with a blue colour. This DSL refers to the GraphQL schema elements and therefore the relevant part of that metamodel, see [Fig. 6.3](#), is contained in [Fig. 6.9](#) as well (the distinction between object and scalar types and also Arguments have been left out for the first iteration due to simplicity reasons). Every **CorrSpec** refers to at least two Endpoints and defines arbitrary many **TypeRelations** and **FieldRelations** (the “relate”-statements in [List. 6.1](#)), which control how the merge is conducted. The latter procedure has been defined in [Alg. 1](#) in [Sec. 5.2.1](#) and can be implemented as an endogenous model transformation over the metamodel in [Fig. 6.9](#). Note the elements shaded in red, which are added to the GraphQL SDL metamodel in order to perform the merging: Every schema element stemming from one of the existing local endpoints is considered a **LocalType** or **LocalField** respectively. When creating

⁶<https://gitlab.com/olevonbargen/graphqlintegrator/>.

Implementation

the key values, identifying the objects that should be merged (e.g. utilising a HashMap), and then invoking MERGE recursively on the merge-clusters.

Algorithm 2 Query Resolving (“Divide & Conquer”)

```
function RESOLVE(query, corrSpec)
  localResults ← new List
  for all ep in corrSpec.endpoints do
    localQueryRoot ← new Selection
    SPLIT(query.root, localQueryRoot, ep)
    localQuery ← new Query with root = localQueryRoot
    response ← SENDREQUEST(localQuery, ep)
    append response to localResults

  responseDocRoot ← new Object
  MERGE(localResults, query.root, responseDocRoot)
  response ← new Document with name = responseDocRoot
  return response

procedure SPLIT(globalSelectionParent, localSelectionParent, ep)
  for all localField in globalSelectionParent.field.contains where
localField.owner.endpoint = ep do
    localSelectionChild ← new Selection with field = localField
    for all globalSelectionChild in globalSelectionParent.selection do
      SPLIT(globalSelectionChild, localSelectionChild, ep)
    append localSelectionChild to localSelectionParent.selections

procedure MERGE(localResults, selection, responseDocParent)
  resultsForField ← TRAVERSETOFIELD(localResults, selection)
  if selection.field.basedOn.identification is null then
    subtree ← CONCATRESULTS(resultsForField, querySelection)
  else
    subtree ← MERGEVIAKEYS(resultsForField, querySelection)
  append subtree to responseDocParent with key = selection.field.name
```

6.2 Second Iteration: Model Management Functionality

Simultaneously to the development of GraphQLIntegrator, a prototype⁷ implementation of comprehensive systems had been started in the context of [446]. This implementation was based on Xtext⁸ and EMF. Quickly, it turned out that both development branches shared several similarities: The underlying structures (schemas and metamodels), the DSLs for defining merged types and commonalities, as well as federated endpoints and comprehensive models are conceptually close to each other. Thus, I began to consolidate both implementations and created a unified code base,

⁷<https://github.com/webminz/comprehensivesystems-emf-prototype>

⁸<https://www.eclipse.org/Xtext/>

EMF	GraphQL	Internal Representation
.ecore file .xmi file	GraphQL schema Documents (returned by resolvers)	DiagrammaticGraph GraphMorphism
EClass	ObjectType	Element (vertex)
EDataType	ScalarType	Element (vertex) with Diagram
EReference	Field (object type)	Element (edge)
EAttribute	Field (scalar type)	Element (edge)
Multiplicities (lowerBound & upperBound)	Modifiers (isListValued & isMandatory)	Diagram
-	Argument	Element (2-edge)
eContainment	-	Diagram
eOpposite	-	Diagram

Table 6.1: Representation of modeling concepts

which eventually became CORRLANG.

In the beginning, two conceptual differences between the two development branches had to be mediated. Firstly, both tools address different technical solution domains (i.e. GraphQL web services vs. EMF models). Secondly, both tools were based on different alignment strategies (i.e. colimit vs. comprehensive system), see Sec. 5.2.

6.2.1 Tech Spaces: Integrating EMF

The Eclipse Modeling Framework (EMF) [436] was mentioned in Chap. 4. It was originally developed by IBM as a code generation framework. Since its inception, it has evolved into a de-facto standard in the MDSE community. In essence, EMF defines an XML-based format for serialising metamodels called *Ecore*. There is a plethora of academic and industrial Ecore metamodels available on the internet. Hence, reading and writing .ecore-files can be seen as a mandatory feature of every MDSE tool. The Ecore metamodel is a subset of MOF comprising the central concepts EClasses (i.e. entity types), EDataTypes (i.e. value types), EAttributes (i.e. relationships between entities and values), and EReferences (i.e. relationships between entities) together with other well-known concepts such as *multiplicities*, *inheritance*, and *composition*. In Sec. 5.1.2, it was analysed how these concepts can be interpreted via (generalised) sketches. An instance of an Ecore-model is stored as an XMI-file and can formally be interpreted as a graph morphism (typing). The concepts of the GraphQL SDL (object types, scalar types, fields) are very similar to Ecore, see Sec. 6.1.1. Therefore, one may interpret them analogously. A summary of the univocal formal interpretation of the various EMF and GraphQL concepts is given in Tab. 6.1. Element represents a generic graphical element. This can be a node, an edge, or an edge between edges (“2-edge”).

In [301], Kurtev et al. introduce the concept of a *technological space*, which defines a context of concepts, tools, a body of knowledge and a representation format. A common example is “XMLware”, which describes the ecosystem of tools and methods that evolved around the XML format. Likewise, EMF and GraphQL can be considered to form technological spaces. Therefore, I will adopt this idea for CORRLANG. Concretely, a technological space (short TechSpace) is identified by a unique identifier and must implement a respective *interface*, see Fig. 6.10. The latter perform a translation between the concrete technology-specific representation and the general mathematical representation in terms of graph, morphisms and sketches.

Implementation

Moreover, the concept of *endpoints*, see Sec. 6.1.1, has to be generalised. In terms of my conceptual framework, a `CORRLANG` endpoint corresponds to a *model space*. In `GraphQLIntegrator`, an endpoint is merely a URL behind which one finds a set of callable operations and a schema for them. This type of endpoint can be seen as a “blackbox”, i.e. there is a priori no way to inspect the internal structure of the system. In EMF, one is working with a set of metamodels and models, which are stored as files, i.e. one can inspect their actual content. Hence, this type of endpoints can be seen as a “whitebox”. To work with both types, the syntax for declaring endpoints has to be extended accordingly.

In the following, I will present the concrete syntax of `CORRLANG` step by step in the form of BNF grammar rules while explaining the respective language constructs. The syntax used for denoting these grammar rules follows common conventions: Terminal symbols are enclosed in ticks and non-terminal symbols are enclosed in angle brackets. The pipe symbol denotes an alternative, the question mark symbol denotes an optional part, a star symbol denotes zero or arbitrary many repetitions, and a plus denotes one to arbitrary many repetitions.

```
<endpoint> ::= 'endpoint' <identifier> '{'  
  'type' ( 'FILE' | 'SERVER' )  
  'at' <url>  
  'technology' <identifier>  
  ('schema' <url> )?  
  '}'
```

The definition of an endpoint comprises an identifier, a location (given by a URL), and a TechSpace. An endpoint of type `FILE` is a “whitebox”, whereas a `SERVER`-endpoint is a “blackbox”. Some technologies (e.g. GraphQL) allow to retrieve the metamodel/schema from the URL alone using inspection. Alternatively, the location of the metamodel can be specified explicitly. List. 6.2 and List. 6.3 provide concrete examples for this syntax.

```
1 endpoint sales {  
2   type SERVER  
3   at http://localhost:4011/  
4   technology GRAPH_QL  
5 }
```

Listing 6.2: GraphQL “blackbox” endpoint

```
1 endpoint BPMN {  
2   type FILE  
3   at file://-/models/patient-referral.simple-bpmn  
4   technology EMF  
5   schema file://-/metamodels/simple-bpmn.ecore  
6 }
```

Listing 6.3: EMF “whitebox” endpoint

Fig. 6.10 summarises the relevant concepts discussed above. Note the resemblance between the contents of the `graphlib` package and the formalism presented in Sec. 5.1. The kind of supported Graphs is generic because `Element` allows to represent various “graphical” features. An `Element` can represent a node, which means that `src` and `trg` references are loops and `degree()`=0. Otherwise, the `Element` represents an edge, which means that the object at the other end of `src` or `trg` reference must have a lower `degree()`, i.e. an `Element` with `degree()`=1 represents a regular edge (connecting nodes), an `Element` with `degree()`=2 represents a 2-edge (connecting edges with nodes or edges) and so on. In this way, not only directed multigraphs can be represented but also more complex structures such as E-graphs, see Fig. 5.2c. The latter

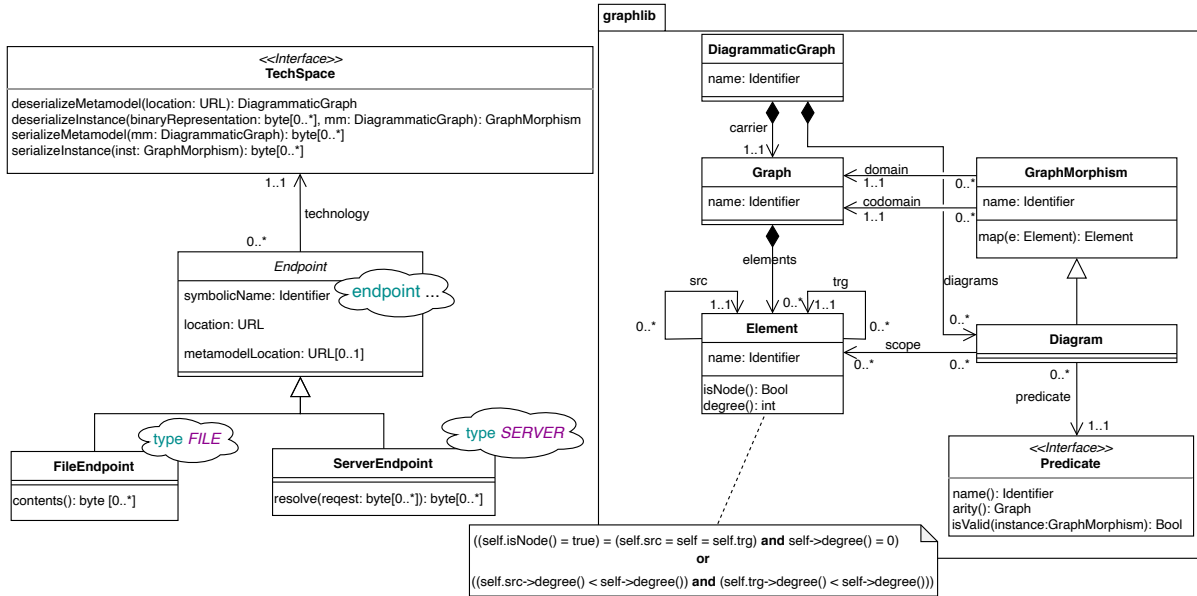


Fig. 6.10: TechSpace-Abstraction and generic Graph library

are, for instance, necessary to model Arguments in GraphQL. The map () operation of a GraphMorphism must ensure that src- and trg-references are preserved in accordance with the generalised homomorphism property, see Def. 5.2.

6.2.2 Comprehensive Systems: Generalising the Federation

Both, the GraphQLIntegrator and the comprehensive system implementation comprise a textual syntax for defining relationships among elements from disparate metamodels. The “only” difference between these languages lies in their intents. The GraphQLIntegrator language is used to specify merging of multiple types, while the comprehensive system DSL is used to establish structural relationships between elements across disparate metamodels. The latter can be presented in an integrated way (see Theorem 8) so that they will appear to the user as “regular” nodes (commonality witnesses) and edges (projections) in the final federated system (global view). This situation is sometimes referred to as “linguistic extension” [109, 393]. Sec. 5.2 showed that the comprehensive system construction is more general than the colimit-based merge. Still, in practice, merging turn out to be useful. Therefore, I decided to support both merging and weaving in CORRLANG. The user can steer this behaviour by using different keywords.

For this, I introduce the language construct Commonality, see Def. 3.4. It generalises the TypeRelation and FieldRelation from Fig. 6.9 since these two GraphQL concepts are treated uniformly as Elements. The resulting class diagram is shown in Fig. 6.11 and the respective grammar rules are given below.

$\langle fullyQualifiedIdentifier \rangle ::= \langle identifier \rangle (\cdot \langle identifier \rangle)_+$

$\langle commonality \rangle ::= (\text{relate} \mid \text{identify}) ((\langle projection \rangle (\cdot \langle projection \rangle)_+) (\text{as} \langle identifier \rangle) ? (\text{with} (\{ (\langle commonality \rangle)^* \} \mid \text{FIELDS}) ?$

Implementation

```
(‘when’ ‘[’ <identificationRule> ‘]’)?
‘;’
```

```
<projection> ::= <fullyQualifiedIdentifier> (‘as’ <identifier>)?
```

```
<identificationRule> ::= <clause> (‘|’ <clause>)*
```

```
<clause> ::= <equation> (‘&&’ <equation>)*
```

```
<equation> ::= <equationLiteral> (‘==’ <equationLiteral> )+
```

```
<equationLiteral> ::= <fullyQualifiedIdentifier> | <stringExpression>
```

The keyword “relate” declares a “weaving” Commonality (comprehensive system) and “identify” declares a “merge” Commonality (colimit). A Commonality has at least two Projections, which refer to Elements. The optional alias of a Commonality is used to give a name for the commonality witness respectively merged element in the resulting global metamodel. The definition of Commonalities can be nested with the keyword “with” in order to relate nested features such as fields (GraphQL) or attributes (Ecore). There is a syntactic sugar (“with FIELDS”), which automatically declares commonalities for all outgoing edges (features) with the same name, compare line 40 in List. 6.1. Moreover, the IdentificationRule concept is inherited from GraphQLIntegrator, i.e. Commonalities can be augmented with such rules acting in the same way as in Sec. 6.1.5.

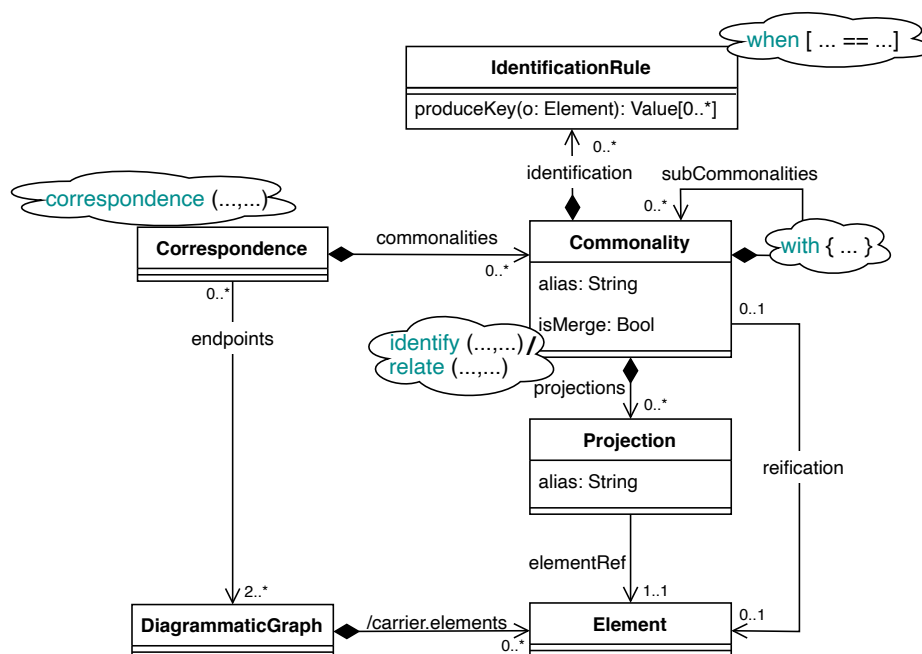


Fig. 6.11: Domain model excerpt for Commonalities

Finally, List. 6.4 contains an example demonstrating how the relationships among BPMN, DMN and UML elements are declared. Note that this is only the the syntactical part, i.e. there are no implicit checks yet.

6.3 Third Iteration: Consistency Management Functionality

```
1 relate (BPMN.Activity as act, DMN.DecisionTable as tab)
2   as DecisionTableDef
3   with {
4     relate (BPMN.Activity.consumes,DMN.DecisionTable.inputSideColumns) as input;
5     relate (BPMN.Activity.produces,DMN.DecisionTable.outputSideColumns) as output;
6   };
7
8 relate (BPMN.DataObject as do,
9   UML.Class as cls,
10  UML.Attribute as att,
11  DMN.Column as clmn)
12  as DataObjectClassAttrColumnImpl
13  with {
14    relate (UML.Attribute.type,DMN.Column.type) as type;
15  }
16
17 identify (UML.DataType,DMN.ColumnType) as BaseType
18   with FIELDS
19   when [UML.DataType.name == DMN.ColumnType.name];
```

Listing 6.4: Commonalities syntax example

6.3 Third Iteration: Consistency Management Functionality

6.3.1 Integration of existing verification tools

Chap. 4 showed that there are several formalisms, tools and languages for expressing consistency rules. One of the central assumptions in Chap. 5 was that existing technical solutions for “local” consistency verification can be reused and therefore the first development goal is to establish an integration with those tools.

The formal framework from Chap. 5 treats consistency rules abstractly as diagrams labelled with a predicate, which defines the semantics. Technically, this is realised as an interface, see Predicate in Fig. 6.10, which allows to implement arbitrary predicate semantics. For bridging the gap between a concrete technology and the generic Predicate-representation, I will employ the TechSpace-concept again, see Sec. 6.2.1. Concretely, the respective interface will receive a new method `parseConsistencyRule()`, which translates a technology-specific textual formulation into a Predicate-implementation. For the time being CORRLANG implements an Epsilon-integration, which parses EVL-constraints.

To support the definition of rules in the CORRLANG-syntax, a new grammar rule is needed, which is specified below. A Rule has a name, a reference to the TechSpace that is used and the concrete implementation of the Rule, which can either be provided as a file-reference (URL) or written directly in the specification.

```
<rule> ::= 'rule' <identifier> '{'
         'technology' <identifier>
         'impl' ( <url> | <externalCode> )
         '}'
```

```
<externalCode> ::= '""' <char>* '""'
```

A Rule is attached to a Commonality (via the new keyword “check”). Per default the scope of the respective Predicate will span the complete comprehensive metamodel,

Implementation

which allows to define arbitrary consistency rules exploiting all features of the local metamodels as well as the “linguistic extension” provided by the comprehensive system.

```
1 relate (BPMN.Activity as act, DMN.DecisionTable as tab)
2   as DecisionTableDef
3   with {
4     relate (BPMN.Activity.consumes,DMN.DecisionTable.inputSideColumns) as input;
5     relate (BPMN.Activity.produces,DMN.DecisionTable.outputSideColumns) as output;
6   } check BusinessRuleIsDefined;
7
8 rule BusinessRuleIsDefined {
9   technology EVL
10  impl """
11    context BPMN!Activity {
12      constraint {
13        check : self.type = ActivityType::BUSINESS_RULE implies
14              DecisionTableDef.allInstances.exists(trace|trace.act = self)
15      }
16    }
17    """
18 }
```

Listing 6.5: CORRLANG Consistency Rule via EVL

List. 6.5 gives an example of the Rule-syntax: Here, **CR₅** is implemented as an EVL-constraint. `DecisionTableDef` acts as a type for commonality witnesses that can be accessed by Epsilons `allInstances()` method. The named projection “act” (see line 1) is rendered as a regular edge and can thus be used like a regular metamodel feature in the formulation of the EVL-constraint (see line 12).

6.3.2 Common Constraints: **INTEGRITY & FORALL**

The mechanism described in the previous section is very flexible and allows to define most kinds of consistency rules since it is as expressive as the underlying technology – EVL in this case. However, the translation between the different technical representation causes some runtime overhead. Therefore, it will be worthwhile to offer built-in support for the most common forms of consistency rules. They are realised as concrete Predicate-implementations called **INTEGRITY** and **FORALL**, see Example 5.3 in Sec. 5.2.2.

The **INTEGRITY**-constraint implements rules of the form of **CR₁**, i.e. the values for all attributes of a distributed element must be equal everywhere this element occurs. List. 6.6 demonstrates how this constraint can be used to check that name and address-fields of identified Customer- and Client-objects from Sec. 6.1 are consistent.

```
1 identify (sales.Customer, invoices.Client)
2   as Partner
3   with FIELDS
4   when [sales.Customer.id == invoices.Client.id]
5   check INTEGRITY;
```

Listing 6.6: INTEGRITY-constraint of Partner

The **FORALL**-constraint is known under many different names. Klare and Gleitze [279] call it “Participation” and in [331] it is called “pattern-matching”. It is a core part of QVTr’s execution semantics [327] and it is effectively the type of rule that is (indirectly) specified by a TGG rule [422]. In its most simple form, it appears as “For every A there must be a B and vice versa”.

6.3 Third Iteration: Consistency Management Functionality

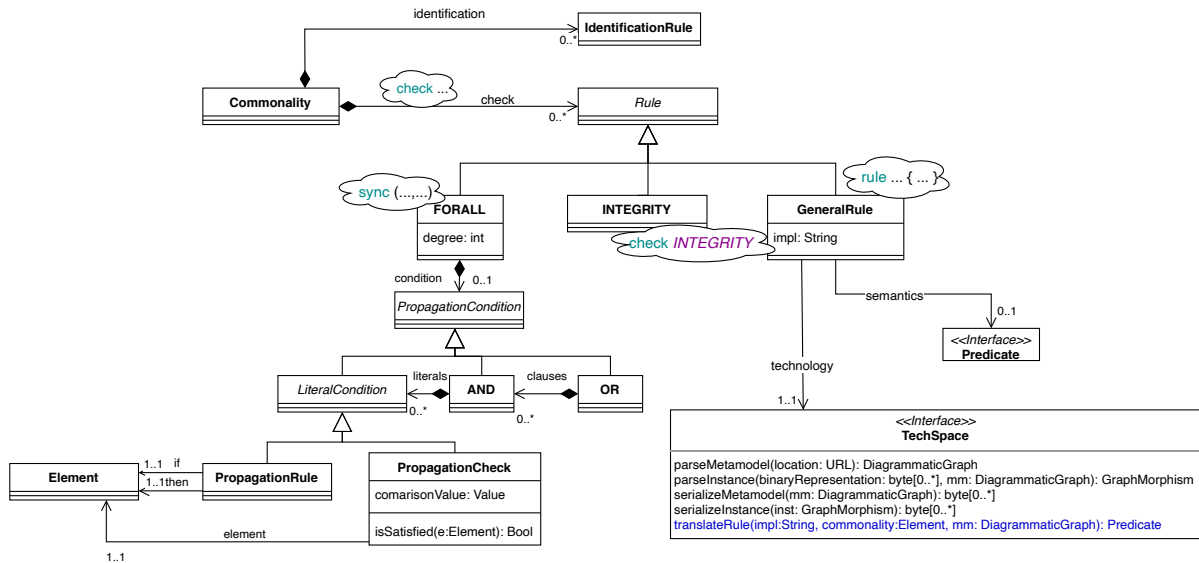


Fig. 6.12: Domain model excerpt for Rules

Since, FORALL type constraints are so common, I will offer a built-in support and special syntax⁹ for it: The new keyword “sync” can be used instead of “relate” or “identify”. A Commonality defined this way behaves basically as a “relate” Commonality but it implicitly has the FORALL constraint attached to it. CORRLANG verifies this constraint by looking for tuples of Elements w.r.t. the types mentioned in the Projections of this Commonality. By default, this check is performed in a *symmetric*-manner (“if and only if”) and invoked for all node-type Elements ($\text{degree}() = 0$). For edge-type Elements ($\text{degree}() > 0$), it is only invoked when src- and trg-Elements are already related by a Commonality.

Sometimes, the FORALL-constraint is only required to hold in one direction and under certain preconditions. For example, the relation between an Activity in BPMN and DecisionTable is required to exist only for one direction (Activity implies DecisionTable but not vice versa) and only if the Activity is of type “BUSINESS_RULE”. To specify this kind of conditions, the the syntax that was originally used for IdentificationRules will be slightly abused: I introduce the new language construct PropagationCondition, which is effectively a superset of the IdentificationRule construct. PropagationConditions can only be attached to “sync” Commonalities and allow to specify the *direction* of the FORALL check in the form of a hierarchy, see the class diagram in Fig. 6.12 and the extended grammar rules below:

```

<commonality> ::= ('relate' | 'identify' | 'sync')
                (' <projection> (' , ' <projection>)+ ')
                ('as' <identifier>)?
                ('with' ('{' (<commonality>)* '}' | 'FIELDS')?
                ('when' '[' <identificationRule> |
                <propagationCondition> // only if sync
    
```

⁹Arguably, this constraint could have been realised in a similar way as INTEGRITY (e.g. “check FORALL”). But, due to the PropagationRule language construct, I decided to introduce a new keyword.

Implementation

```
'?']?
('check' ('INTEGRITY' | <identifier>)) (',' <identifier>)*
';'
```

```
<propagationRule> ::= <propClause> ('||' <propClause>)*
```

```
<propClause> ::= <propClauseLiteral> ('&&' <propClauseLiteral>)*
```

```
<propClauseLiteral> ::= <equation> | <propagationRule>
```

```
<propagationRule> ::= <fullyQualifiedIdentifier> '~~>' <fullyQualifiedIdentifier>
```

To give an example, let us revisit the well known scenario featuring BPMN, UML, and DMN (Fig. 1.9) and the consistency rules CR5-CR8. List. 6.7 shows a technical realisation of these rules utilising the FORALL constraint. Comparing this Listing to List. 6.4, the relate-statements have been replaced by sync-statements and propagation-conditions have been added. For instance, line 8 encodes the rule that every Activity of type BUSINESS_RULE must have an associated DecisionTable (CR5). The built-in “sync”-notation makes the EVL-constraint from List. 6.5 obsolete. Also note that the FORALL-constraints on the nested Commonalities in lines 4-5, 6-7, and 16 contain Projections onto edge-type Elements and are thus only invoked when there is Commonality instance among the respective container Elements, i.e. Activity/DecisionTable and Attribute/Column respectively.

```
1 sync (BPMN.Activity as act, DMN.DecisionTable as tab)
2   as DecisionTableDef
3   with {
4     sync (BPMN.Activity.consumes, DMN.DecisionTable.inputSideColumns) as input
5     when [DMN.DecisionTable.inputSideColumns ~-> BPMN.Activity.consumes];
6     sync (BPMN.Activity.produces, DMN.DecisionTable.outputSideColumns) as output
7     when [DMN.DecisionTable.outputSideColumns ~-> BPMN.Activity.produces];
8   } when [act.type == "ActivityType::BUSINESS_RULE" && act ~-> tab];
9
10 sync (BPMN.DataObject as do,
11   UML.Class as cls,
12   UML.Attribute as att,
13   DMN.Column as clmn)
14 as DataObjectClassAttrColumnImpl
15 with {
16   sync (UML.Attribute.type, DMN.Column.type) as type;
17 } when [do ~-> cls && clm ~-> att
18   ||
19   do ~-> att && clm ~-> att];
```

Listing 6.7: Implementing CR5-CR8 with the FORALL-constraint

6.3.3 Model Management via Goals

Finally, in order to execute consistency verification, CORRLANG must be extended to support several execution modes. The CORRLANG version after the second iteration supported only one execution mode: FEDERATION, which varies based on the type of the underlying Endpoints. If all of them are ServerEndpoints, CORRLANG emulates a federated server interface. If all¹⁰ of them are FileEndpoints, CORRLANG creates

¹⁰Mixing both types is currently not possible.

6.3 Third Iteration: Consistency Management Functionality

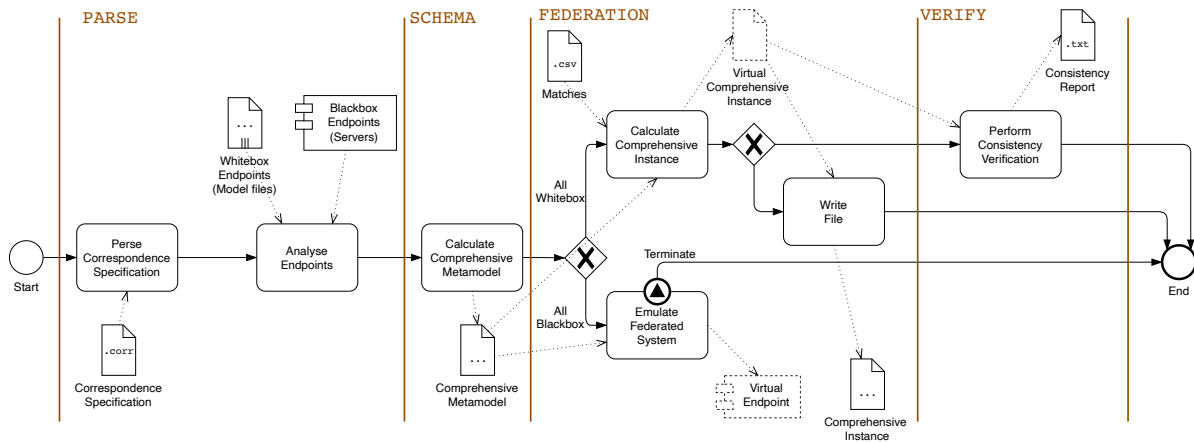


Fig. 6.13: CORRLANG execution process

a file representing a global view of all models. In addition to that, a comprehensive schema/metamodel is created beforehand. The execution process is visualised in Fig. 6.13, which also illustrates how I am going to build consistency verification on top of the existing FEDERATION concept.

A user should be able to decide “what” he wants to do with CORRLANG, i.e. if she or he wants to perform consistency verification, create a comprehensive model, or simply calculate the comprehensive metamodel alone. Thus, I introduce the language construct of Goals, which allow to specify how far the process in Fig. 6.13 shall be executed (note the vertical bars) and to configure technical details. In this context, the possibility to import *external matches* when working with FileEndpoints is introduced as well. CORRLANG supports matching of instance-elements via keys. But, when working with design models, this is not always possible since elements may be named differently. Thus, in order to support a more flexible approach, a CSV file can be imported during the creation of a global view. The CSV file contains an arbitrary number of tuples. The first field has to be the name of a Commonality (type), and the remaining fields are Element identifiers that constitute the concrete Commonality tuple on the instance level. This file can be created manually or may be produced by an existing model matching solution, see Sec. 4.2.5.

When working with ServerEndpoints, a distinct verification cannot be executed since it is a priori not possible to inspect the current database state. However, CORRLANG has a possibility to do “local”-checks in the FEDERATION-execution mode. This means that consistency rules are checked on the response documents returned by a request. The user can retrieve the result of this check by reading newly synthetic field that is introduced in the global schema by CORRLANG. The grammar-rules for defining Goals is given below and their class diagram is shown in Fig. 6.14.

```

<goal> ::= 'goal' <identifier> '{'
  'correspondence' <identifier>
  'action' ( 'PARSE' | 'SCHEMA' | 'FEDERATION' ('with check')? | 'VERIFY' )
  ('matches' <url>)?
  ('technology' <identifier>)?
  ('target' ( 'FILE' '{' 'at' <url> '}' | 'SERVER' '{' 'port' <number> 'path' <string> '}'
  ))? '}'

```

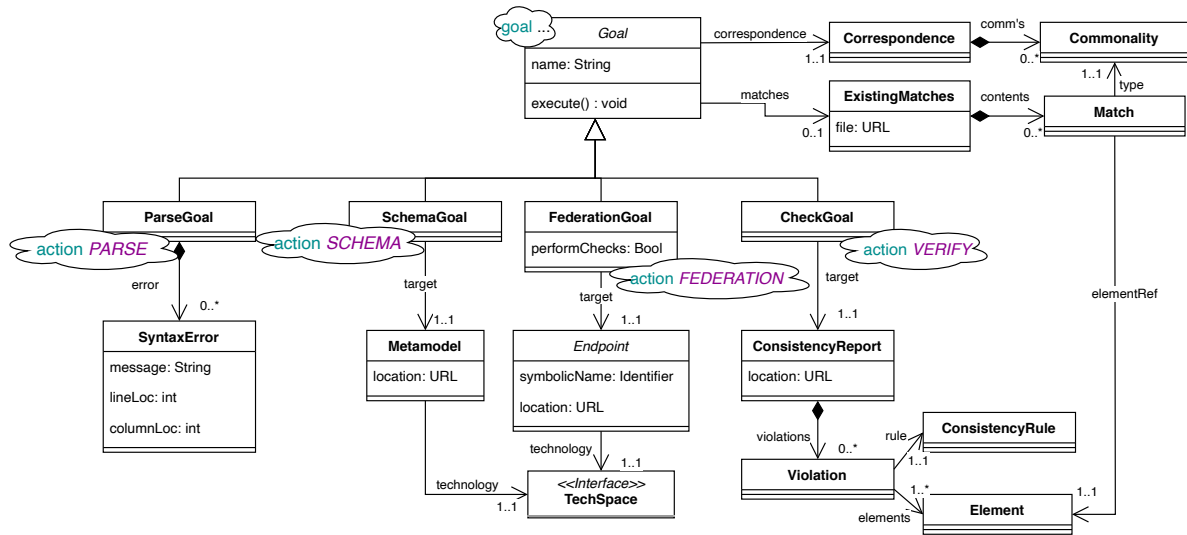



Fig. 6.14: Domain model excerpt for Goals

6.4 Summary & Future Directions

The result of third iteration corresponds to the current version of CORRLANG which is found on *github*. It supports calculating comprehensive metamodels, comprehensive models (federation) and performing consistency verification with support for the technologies GraphQL, EMF, and Epsilon EVL. Thus, allowing to mediate between heterogeneous technologies and metamodels. Hence, the research questions **RQ₁**, **RQ₂**, and **RQ₄** are addressed while safeguarding support for general multi-ary correspondence relationships among models.

An obvious direction for future work is to extend the list of supported technologies. Promising first candidates are the “XMLware” ecosystem (i.e. XML-schema for metamodel-definition and WSDL for server interface description), the semantic web ecosystem (i.e. OWL schemas and RDF instances), OpenAPI (i.e. schemas for REST-interfaces), or a programming language such as Java because annotated Java-classes can act as metamodels as well, see Fig. 1.10. Programming these integrations is expected to be more or less straightforward. Beyond that, there are open issues, which require additional conceptual work:

MORE FLEXIBLE MODEL MATCHING Currently, elements of an instance can be matched via keys (*IdentificationRule*) or by using already identified matches that are given from the outside (CSV-file). For the future, it will be useful to support more flexible matching. This is necessary in situations where elements cannot be identified by “hard”-criteria such as keys but instead are matched based on “soft”-criteria such as (structural) similarity, metrics, ontologies or thesauri. Thus, the result of the matching procedure may become ambiguous. This and the fact that model matching is a vast research domain on its own, compare Sec. 4.2.5, requires more conceptual work before this feature can be implemented.

MIXING ServerEndpoints AND FileEndpoints Currently, CORRLANG can work with model files and server interfaces. However, it is not yet supported to mix both types

in one Correspondence. The central idea to address this limitation in the future is that a FileEndpoint can always be considered as a ServerEndpoint by adding a CRUD-interface for each type in the metamodel. Still, there are some open questions on how consistency verification and repair can be performed on top of ServerEndpoints. Interestingly, a large portion of the literature on model repair considers the “whitebox”-situation only and does not consider access methods explicitly.

MODEL REPAIR Goal In terms of the conceptual multi-model consistency management framework, CORRLANG only supports 2 out of the 3 central operations, i.e. matching and verification. Consistency restoration could be performed by exporting the comprehensive instance into a specific TechSpace and utilise existing model repair tools for this technology. But, to support all aspects of multi-model consistency management CORRLANG must support model repair by adding a respective REPAIR Goal. This requires further work on the conceptual and also the formal side, see Sec. 5.4. Furthermore, it would be interesting to investigate whether and how existing rule-based (e.g. *Henshin* [48] or *emoflon* [482]) and search-based (e.g. *PARMOREL* [35]) solutions can be integrated internally.

INFORMATION HIDING AND A MORE POWERFUL PROJECTION-LANGUAGE At present, the Projections of a Commonality are given by fully qualified element references. In some scenarios, this may not suffice. For example, if there is no relationship between elements of separate model immediately, but only after additional computation of *derived* elements. Another aspect is *privacy*, i.e. there is some information that must not be shared, hence requiring additional steps of filtering and/or *obfuscation*. The formal underpinning of this feature are the *diagrammatic operations* mentioned in Sec. 5.4.

“If you tell the truth you don’t have to remember anything.”

—Mark Twain

CHAPTER 7

VALIDATION

An important characteristic of constructive technology research such as software engineering (see Sec. 2.3) is a final evaluation of the results, i.e. to check whether the constructed artefacts address the initial problem and to what extent these artefacts represent an improvement compared to existing solutions. In [379], Petersen et al. distinguish between *evaluation* and *validation* studies. The difference between them is that evaluation happens in an (uncontrolled) industrial environment whereas validation happens in a (controllable) academic environment. In my case, the latter category applies. In this chapter, I will present a validation of my three research artefacts (scientific contributions): **A1** the *conceptual framework* (Sec. 7.1), **(A2)** the *comprehensive system formalism* (7.2), and **A3** the *tool* (Sec. 7.3).

7.1 Validation of the Conceptual Framework

The utility of the conceptualisation in Chap. 3 must be evaluated in terms of its ability to abstractly capture the two motivating scenarios presented in Sec. 1.3.

SEMANTIC INTEROPERABILITY OF SOFTWARE SYSTEMS Fig. 7.1 depicts the situation described in Sec. 1.3.1 (Fig. 1.3) employing the visual language of the conceptual framework from Chap. 3. Every system forms a model space, see Sec. 3.3.1. The “models” in these model spaces represent *database states* of the respective systems. The change-relation models database updates. Each model space is defined by a metamodel, i.e. the public schema describing the system interface. Systems may have identical schemas. For instance, the GP journal system, the hospital journal system, and the scheduling system are all based on the FHIR metamodel. Still, they are represented by different model spaces because their databases’ contents will differ at various points in time. Among the models in this scenario there are many possible correspondences, which are induced by shared data. For example, Patient-records shared among the GP journal system, the hospital journal system, and the scheduling system or Observation-records shared among the hospital journal system, the laboratory system and the imaging system. This information represents the commonality data, see Sec. 3.3.2. This data is subject to **CR1**. This rule can be rendered as a technical rule definition based on the concepts of the FHIR Metamodel, the Laboratory System Schema, the Imaging System Schema, and the Payment System Schema, see Sec. 3.3.3. Because records are shared across more than two systems, verifying this global consistency rule is best performed utilis-

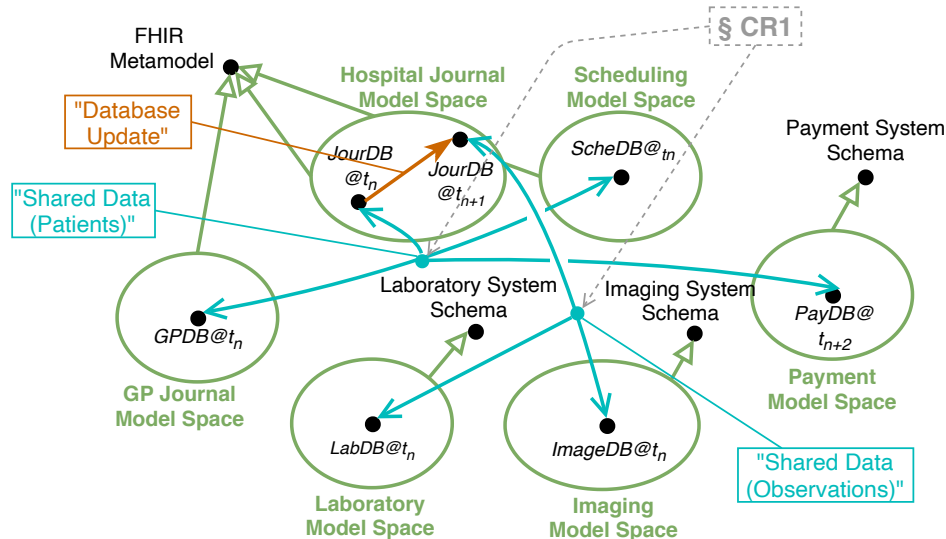


Fig. 7.1: Semantic Interoperability of Software Systems: Conceptual Picture

ing the global view architecture, see Sec. 3.3.5. Thus, a tuple of database states together with information about shared records is turned into a single global database state, which contains all records and highlights shared ones, which will enable consistency verification. Model repair, see Sec. 3.3.4, in this scenario allows to implement automatic synchronisation of inconsistent records. It is important to note that this conceptualisation assumes that the relevant part of the current state of a system database is accessible at any given time. Also the technical aspects of inter-system communication are abstracted away.

CONSISTENCY OF SOFTWARE DESIGN DOCUMENTS Fig. 7.2 depicts the conceptual picture of the design artefact scenario from Sec. 1.3.2 (Fig. 1.7). Each stakeholder shown in Fig. 1.7 is turned into a model space, which is defined by a metamodel that represents the respective modeling language, see Sec. 3.3.1. The models in this case are the versions of the individual design documents while the change relation traces the evolution of these documents. The correspondences are given by the traceability relationships which are subject to the consistency rules CR2-CR14. They are divided into three groups called “Refinement”, “Dependency”, and “Implementation”, which correspond to the three aspects of this scenario discussed alongside Fig. 1.8, Fig. 1.9, and Fig. 1.10 respectively. One can interpret them as three multi-models or consider them altogether. The commonality data in these multi-models are links between model elements that are required for the consistency rules, see Sec. 3.3.2. Some of the consistency rules only involve binary relationships, e.g. CR2 or CR3, and could therefore be represented as a *consistency network*. Other rules, CR8, however, require proper multi-ary relationships for their representation and must therefore be realised with the help of global views, see Sec. 3.3.5. The latter can be interpreted as one global design model, which contains the contents of every model including the traceability relationships in order to perform multi-model consistency management with the help of a single artefact.

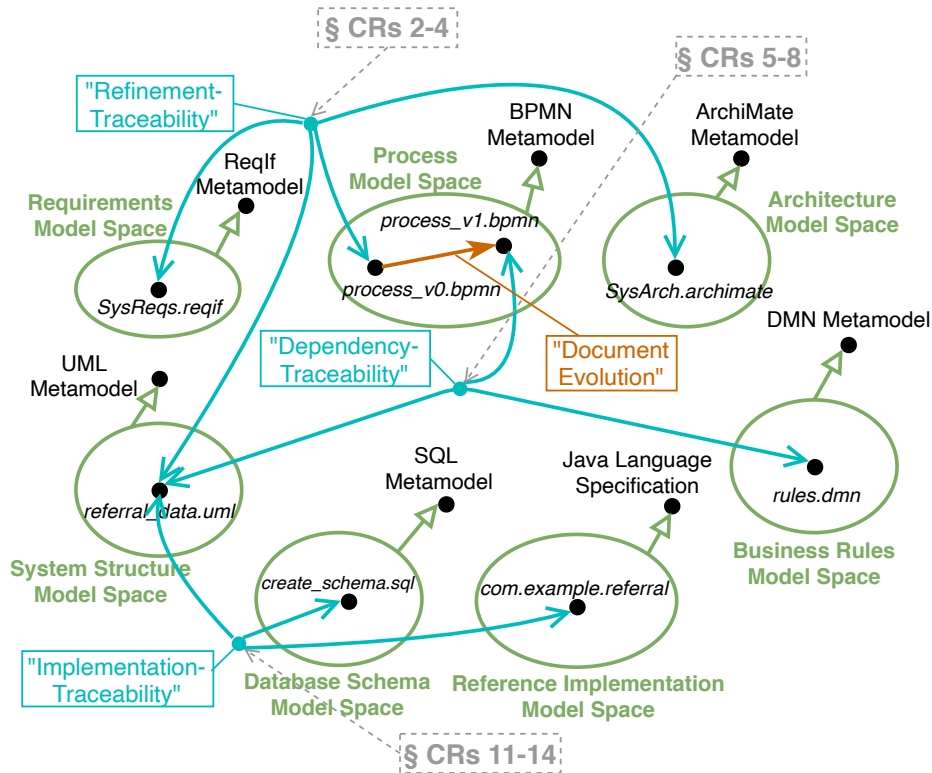


Fig. 7.2: Consistency of Software Design Models: Conceptual Picture

7.2 Validation of the Formalism

The central contribution of Chap. 5 is a novel formalism called *comprehensive systems*. The latter describes a formal representation of a collection of software models and structural relationships among their elements in terms of a single integrated artefact (global view). Internal validity was assured by the proofs of the respective propositions and theorems, where the majority of these proofs are found in Appendix B. Regarding the external validity, I want to illustrate the concrete significance of the theoretical results in practice.

First, it is important to mention that comprehensive systems represent a straightforward formalisation of what many practitioners are already doing intuitively: Drawing connections (trace-links) between formerly unconnected elements. Thus, comprehensive systems provide the formal underpinning for generic and domain specific trace models. The latter represents one of the two main limitations identified in Sec. 4.5.

Theorem 9 states that *comprehensive systems are a suitable carrier structure for diagrammatic constraints*. The latter have been shown to capture model constraints in software engineering [408, 409]. Moreover, it could be shown that the resulting framework of diagrammatic constraints over comprehensive systems forms a *semi-institution*, see Corr. 10. Institutions are heavily used by researchers, who investigate the integration of heterogeneous formal verification approaches [347, 348]. Thus, opening the door for integrating my formalism with established formal means for consistency verification in the future.

Theorem 12 states that *comprehensive systems form a weak adhesive HLR category with respect to a special class of "reflective" monomorphisms*. Weak adhesive HLR categories are

Validation

the foundation of *algebraic graph transformation*, a mature formalism for investigating rule-based rewriting of graph-based structures. Most of the results in this extensive research domain are formulated for this kind of categories and hence all the major results discovered in this research domain apply for comprehensive systems as well [146]. Moreover, this allows comprehensive systems to be implemented in proven graph transformation tools such as *Henshin* [48] or *eMoflon* [482].

Theorem 18 shows that *comprehensive systems generalise the underlying categories of graph diagrams (which include triple graphs)*. This means that every consistency rule that can be represented via triple graphs and graph diagrams can also be represented using comprehensive systems. However, comprehensive systems are more general and can express situations that cannot be captured with triple graphs or graph diagrams, e.g. general multi-ary model relationships where the final arities of these relationships are unknown in the beginning or may change over time (assuming an arbitrary but fixed maximal number of models under consideration). The ability to represent multi-ary correspondence relationships directly addresses a main limitation with current approaches, see Sec. 4.5.

7.3 Validation of the Tool

Finally, I will assess the capabilities of CORRLANG, which represents the practical contribution of this PhD project. Concretely, I will demonstrate its ability to address practical multi-model consistency management problems (*feasibility*), analyse its current *features*, and evaluate its performance when dealing with large models and big collections of elements (*scalability*).

7.3.1 Feasibility

I am investigating two use cases, which are based on the practical scenarios introduced in Sec. 1.3 and modelled on an abstract level in Sec. 7.1. Both use cases are (for the sake of presentation) slightly simplified compared to their real-world pendants. The respective software artefacts are publicly available in a source code repository¹. This section contains several code snippets and the complete code is found in the repository.

SEMANTIC INTEROPERABILITY OF SOFTWARE SYSTEMS The first case features three systems: the journal system of the general practitioner, the journal system of the hospital and the laboratory system storing blood test results. I will refer to them by GP, HOSPITAL, and LAB. GP and HOSPITAL are based on the FHIR data model whereas LAB is based a custom-made schema. The goal, in accordance with the technical challenges mentioned in Sec. 1.3.1, is (i) to make the data of all systems accessible behind a homogenous interface, (ii) to identify and represent shared data, and (iii) check shared data for inconsistencies, see CR1. CORRLANG currently supports the technologies GraphQL and EMF. Let us therefore assume that all three systems have a GraphQL interface². The first

¹<https://github.com/webminz/corrlang-scenarios>

²This is only a minor technical limitation. Support for more technologies is planned in the future. Moreover, GraphQL support for FHIR is actually under development, see [236, 349]

step when developing a solution with CORRLANG is to define the relevant Endpoints (model spaces), see lines 1–15 in List. 7.1.

```

1 endpoint GP {
2   type SERVER
3   at https://... /* URL of the GP journal GraphQL endpoint */
4   technology GRAPH_QL
5 }
6 endpoint HOSPITAL {
7   type SERVER
8   at https://... /* URL of the Hospital journal GraphQL endpoint */
9   technology GRAPH_QL
10 }
11 endpoint LAB {
12   type SERVER
13   at https://... /* URL of the Laboratory GraphQL endpoint */
14   technology GRAPH_QL
15 }
16
17 correspondence DataInteg (GP,HOSPITAL,LAB) {
18   /* Commonalities */
19   [...]
20 }
21
22 /* Creates a federated GraphQL endpoint at http://localhost:8080/graphql */
23 goal GraphQLEndpoint {
24   correspondence DataInteg
25   action FEDERATION with check
26   technology GRAPH_QL
27   target SERVER {
28     port 8080
29     path /graphql
30   }
31 }

```

Listing 7.1: First step: Identifying the involved model spaces

Lines 23–31 in List. 7.1 declare a Goal, which emulates a federated GraphQL endpoint accessible under the following URL: `http://localhost:8080/graphql`. This address provides a uniform interface to access the three systems. Without further ado, this declaration will result in a “parallel composition” of the three GraphQL endpoints GP, HOSPITAL, and LAB, i.e. one can interact with the individual systems but there is no further coordination concerning shared data. Thus, concrete Commonalities have to be defined, see line 19 in List. 7.1. First, let us have a look at the data models behind the three systems.

Fig. 7.3 depicts (a simplified³ extract of) the schema of the FHIR data model and Fig. 7.4 depicts the custom-made schema of the laboratory system. I am using class diagrams as a visual syntax for GraphQL schemas, see Sec. 6.1.1. Note also that excerpts of these models were already shown in Chap. 1 (Fig. 1.6). GP and HOSPITAL are both using FHIR and therefore they have identical schemas. CORRLANG cannot detect this fact a priori and treats them as they would be completely different. The user has to manually define their entity types as identical, see lines 2–5 in List. 7.2. The syntactical sugar “with FIELDS” alleviates writing these statements as it automatically also identifies all related fields with same name and target type.

```

1 /* Identification of identical entity types */
2 identify (GP.Encounter,HOSPITAL.Encounter) as Encounter with FIELDS;
3 identify (GP.Observation,HOSPITAL.Observation) as Period with FIELDS;

```

³Compared to the official definition, some fields and resource types are left out and some hierarchical content is collapsed to ease presentation.

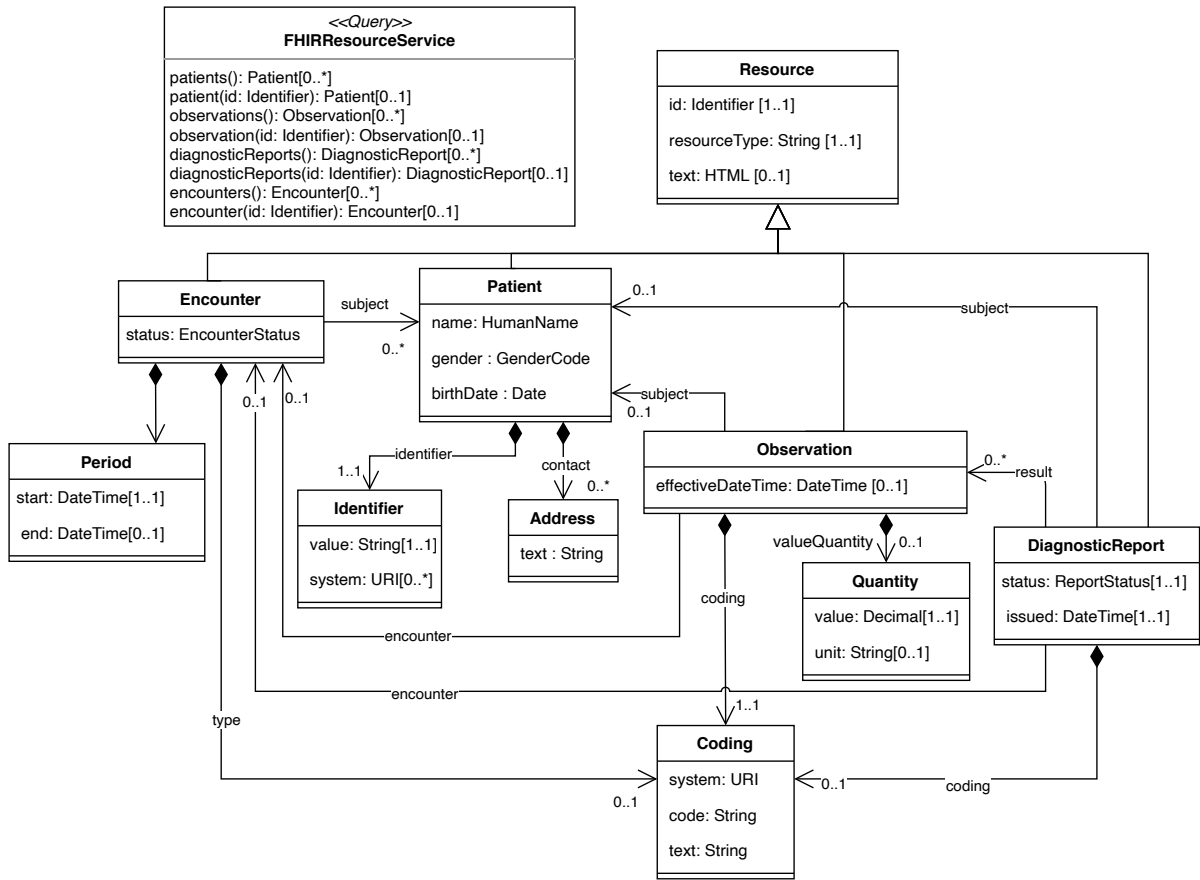


Fig. 7.3: FHIR schema (simplified)

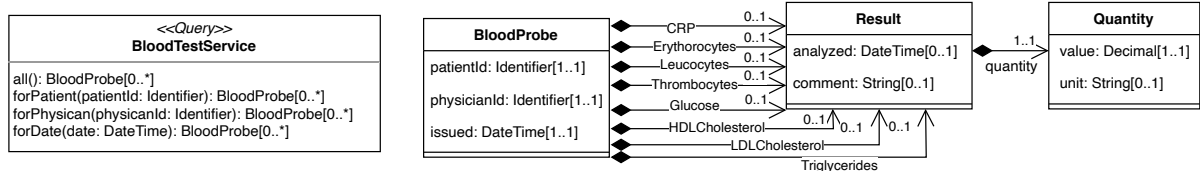


Fig. 7.4: Laboratory schema

```

4 identify (GP.Quantity,HOSPITAL.Quantity) as Period with FIELDS;
5 ... /* analogous for the other types in Fig. 7.3 */
6
7 /* Identification of the Patient entity type and declaring a rule that matched Patient records based on their national ids */
8 identify (GP.Patient,HOSPITAL.Patient) as Patient with FIELDS
9   when [ GP.Patient.identifier.system == "https://www.skatteetaten.no"
10         HOSPITAL.Patient.identifier.system == "https://www.skatteetaten.no" &&
11         GP.Patient.identifier.value == HOSPITAL.Patient.identifier.value ]
12   check INTEGRITY; /* Ensuring CR1 */

```

Listing 7.2: Second step: Aligning FHIR schemas

Special attention goes to the `identify` statement concerning the Patient entity (lines 8–12), where I define a domain specific IdentificationRule. In this use case, patients are “globally” identified through a *national id* (= the Norwegian “*personnummer*” assigned by the “*Folkeregisteret*”, which has the top-level URI <https://www.skatteetaten.no>). The “`check INTEGRITY`” statement will allow us to verify the structural integrity of Patient-records in the end, see Sec. 6.3.2.

In the next step, the custom-made schema of LAB has to be integrated. In practice, this will require to consult the domain experts responsible for the individual systems. In our case, let us assume that this consultation uncovered that the BloodProbe type can be mapped to a FHIR DiagnosticReport, the Result-type maps to an Observation and the Quantity-types are already identically designed. The definition of the respective Commonalities is shown in List. 7.3.

```

1 /* Integrating the Laboratory schema types*/
2 identify (HOSPITAL.Observation,LAB.Result) as Observation with {
3   identify (HOSPITAL.Observation.effectiveDateTime,LAB.Result.analyzed) as effectiveDateTime;
4   identify (HOSPITAL.Observation.valueQuantity,LAB.Result.quantity) as valueQuantity;
5 } check INTEGRITY;
6 identify (GP.Quantity,HOSPITAL.Quantity,LAB.Quantity) as Quantity with fields;
7 identify (HOSPITAL.DiagnosticReport,LAB.BloodProbe) as DiagnosticReport with {
8   /* the specific blood test values are treated as Observation results with respective LOINC codes */
9   identify (HOSPITAL.DiagnosticReport.result,LAB.BloodProbe.CRP) as result
10  when [ HOSPITAL.DiagnosticReport.result.coding.system == "https://loinc.org/" &&
11         HOSPITAL.DiagnosticReport.result.coding.code == "1988-5" ];
12  identify (HOSPITAL.DiagnosticReport.result,LAB.BloodProbe.Erythrocytes) as result
13  when [ HOSPITAL.DiagnosticReport.result.coding.system == "https://loinc.org/" &&
14         HOSPITAL.DiagnosticReport.result.coding.code == "5161-5" ];
15  ... /* analogous for the other fields in Fig. 7.4 such as Leucocytes etc. */
16 /* identification rule treating BloodProbe as a FHIR Diagnostic Report with the respective SNOMED-CT term
17 and matching records if they refer to the same patient */
18 } when [ HOSPITAL.DiagnosticReport.coding.system == "http://snomed.info/sct" &&
19         HOSPITAL.DiagnosticReport.coding.code == "396550006" &&
20         HOSPITAL.DiagnosticReport.subject.identifier.system == "https://www.skatteetaten.no"
21         HOSPITAL.DiagnosticReport.subject.identifier.value == LAB.BloodProbe.patientId ];

```

Listing 7.3: Third step: Integrating the Laboratory schema

Note that `identify` statements are “additive”, see Sec. 6.2.2: Since line 3 in List. 7.2 already identified the Observation types from GP and HOSPITAL, the `identify` statement in line 2 in List. 7.3 adds up to the same Commonality such that GP Observation, HOSPITAL Observation and LAB Result are merged into the same type. The same applies for the respective fields `effectiveDateTime` and `valueQuantity`. The special IdentificationRules that are applied to fields of BloodProbe (lines 10–11, 13–14, ...) make sure that the outgoing references of BloodProbe-instances are mapped to Observation-instances with the correct coding. The FHIR data model is much more generic than the LAB schema. Thus, in order to represent the concrete type of an observation or a medical procedure it employs semantic coding systems such as LOINC or

Validation

```
1 query {
2   diagnosticReports {
3     subject {
4       name {
5         text
6       }
7       contact {
8         text
9       }
10      CORRLANG_isStructurallyConsistent
11    }
12    result {
13      coding {
14        code
15      }
16      valueQuantity {
17        value
18        unit
19      }
20    }
21  }
22 }
```

```
{
  "data": {
    "diagnosticReports": [
      {
        "subject": {
          "name": {
            "text": "Patrick Stünkel"
          },
          "contact": [
            {
              "text": "Nøstegaten 81, 5011 Bergen"
            },
            {
              "text": "Gjernesvegen 103, 5700 Voss"
            }
          ],
          "CORRLANG_isStructurallyConsistent": false
        },
        "result": [
          {
            "coding": {
              "code": "1988-5"
            },
            "valueQuantity": {
              "value": 8.723,
              "unit": "mg/L"
            }
          }
        ]
      }
    ]
  }
}
```

Fig. 7.5: Resulting GraphQL Endpoint

SNOMED-CT.

```
1 identify (GP.FHIRResourceService,HOSPITAL.FHIRResourceService, LAB.BloodTestService) as
  FHIRResourceService with {
2   identify (GP.FHIRResourceService.diagnosticReports,HOSPITAL.FHIRResourceService.diagnosticReports,
  LAB.BloodTestService.all) as diagnosticReports;
3   };
```

Listing 7.4: Final step: Coordinating Queries

Finally, in order to access the shared records, one has to identify the Query operations, which is shown in List. 7.4. By identifying the query fields `diagnosticReports` (GP/HOSPITAL) and `all` (LAB) under the new name `allDiagnosticReports`, there will be a new Query operation, which calls all systems simultaneously and afterwards consolidates the results according to the IdentificationRules defined in List. 7.2 and List. 7.3. Moreover, due to the “check INTEGRITY” statements, the Patient and Observation types will be augmented with the Boolean field “CORRLANG_isStructurallyConsistent”, whose value shows whether the record is structurally consistent.

Fig. 7.5 shows how resulting GraphQL endpoint appears to clients in the end. The query shown on the left hand side retrieves the Patient and Observation data from all DiagnosticReports. The result shows a report for a patient that has a CRP BloodProbe result. The latter information stems from the LAB system. Moreover, this patient has two

different Addresses stemming from the aggregation of the data in GP and HOSPITAL. Thus, the structural consistency check (CORRLANG_isStructurallyConsistent) is violated. As a conclusion, all of the initially stated goals (i-iii) have been addressed.

CONSISTENCY OF SOFTWARE DESIGN DOCUMENTS The second example comprises five design documents named REQS, ARCH, BPMN, UML, and DMN. They have been introduced by Fig. 1.7, excluding the database schema and the reference implementation because the latter is yet another instance of the well-known object-relational mapping problem. Discussing it here again would not yield new insights. Therefore, I will only discuss the consistency rules CR2–CR8 here. The final goal of this use case is to perform global consistency verification. This requires to translate CR2–CR8 into the CORRLANGDSL and to invoke a VERIFY-type GOAL.

Each document is stored as a file on disk and is denoted in a different modeling language. Thus, I am working with FileEndpoints. Further, I am assuming that each modeling language is defined by an Ecore-model. Hence, the EMF TechSpace is used. Lines 1–34 in List. 7.5 shows the respective Endpoints.

```

1 endpoint REQS {
2     type FILE
3     at models/requirements.xmi
4     technology EMF
5     schema metamodels/reqIf.ecore
6 }
7
8 endpoint ARCH {
9     type FILE
10    at models/businessArchitecture.xmi
11    technology EMF
12    schema metamodels/archimate.ecore
13 }
14
15 endpoint BPMN {
16    type FILE
17    at models/process.xmi
18    technology EMF
19    schema metamodels/bpmn.ecore
20 }
21
22 endpoint UML {
23    type FILE
24    at models/dataModel.xmi
25    technology EMF
26    schema metamodels/uml.ecore
27 }
28
29 endpoint DMN {
30    type FILE
31    at models/decisionModel.xmi
32    technology EMF
33    schema metamodels/dmn.ecore
34 }
35
36 correspondence PatientReferralDesign (REQS,ARCH,BPMN,UML,DMN) {
37
38 }

```

Listing 7.5: First step: Endpoint definition

In the next step, Commonalities have to be defined, which represent the types of the possible traceability-links among the design model element. I begin with the presentation of the CORRLANGportrayal of CR5–CR8, which is shown in List. 7.6. The

Validation

solution utilises the built-in FORALL-constraint and has already been discussed in Sec. 6.3.2, see List. 6.7.

```
1  /* Realises CR5 */
2  sync (BPMN.Activity as act, DMN.DecisionTable as tab)
3    as DecisionTableDef
4    with {
5      /* Realises parts of CR8 */
6      sync (BPMN.Activity.consumes, DMN.DecisionTable.inputSideColumns) as input
7        when [DMN.DecisionTable.inputSideColumns ==> BPMN.Activity.consumes];
8      sync (BPMN.Activity.produces, DMN.DecisionTable.outputSideColumns) as output
9        when [DMN.DecisionTable.outputSideColumns ==> BPMN.Activity.produces];
10     } when [act.type == "ActivityType::BUSINESS_RULE" && act ==> tab];
11
12 /* Needed for CR7 */
13 identify (UML.DataType, DMN.ColumnType) as BaseType
14   with FIELDS
15   when [UML.DataType.name == DMN.ColumnType.name];
16
17
18 /* Realises CR5, CR6, CR7, and CR8 together */
19 sync (BPMN.DataObject as do,
20       UML.Class as cls,
21       UML.Attribute as att,
22       DMN.Column as clmn)
23   as DataObjectClassAttrColumnImpl
24   with {
25     sync (UML.Attribute.type, DMN.Column.type) as type;
26   } when [ do ==> cls && clmn ==> att
27           ||
28           do ==> att && clmn ==> att ];
```

Listing 7.6: Second step: Translating CR5–CR8

Secondly, CR2–CR5 are implemented in a similar fashion. However, some of these rules are slightly more complicated and require traversal over the object graph such that they cannot immediately be realised with FORALL constraints. Thus, I employ Epsilon’s EVL, see Sec. 6.3.1, to implement the rules, which is shown in List. 7.7. The formulation of these rules demonstrates how the linguistic extension (see features `ProcessImpl`, `role` and `pool` in line 33), produced by the `relate/sync` statements, can be used.

```
1  /* Realises CR2 */
2  identify (REQS.SpecObject, ARCH.Requirement) as Requirement
3    when [REQS.SpecObject.identifier, ARCH.Requirement.name]
4    check FunctionalRequirementIsImplemented;
5
6  rule FunctionalRequirementIsImplemented {
7    technology EVL
8    impl ""
9      context Requirement {
10       constraint C1 {
11         guard : self.type = "Functional Requirement"
12         check : self.attributes.exists(a|a.type="Status" and a.value="accepted") implies
13               ARCH!Association.allInstances.exists(i|i.label= "implements" and i.target = self)
14       }
15     ""
16   }
17 /* Realises CR3 */
18 sync (ARCH.DataEntity, UML.Class) when [ARCH.DataEntity ==> UML.Class];
19
20 /* Realises CR5 */
21 sync (ARCH.Process as proc, BPMN.BPMNDiagram as diag) as ProcessImpl when [ARCH.Process ==> BPMN.
22   BPMNDiagram] check RolesParticipateAsPools;
23
24 relate (ARCH.Role as role, BPMN.Pool as pool) as Party;
```

```

25 rule RolesParticpateAsPools {
26   technology EVL
27   impl """
28     context ARCH!Process {
29       constraint C2 {
30         check : ARCH!Association
31         .select(a|a.source = self and a.label = "participation" and a.target.isTypeOf(ARCH!
32         Role))
33         .forall(a|Party.allInstances.exists(p|
34         p.role = a.target and self.ProcessImpl.diag.pool.contains(p.pool)))
35       }
36     }
37   """
}

```

Listing 7.7: Third step: Translating CR₂–CR₅

Finally, the verification VERIFY-type GOAL has to be defined (lines 20–27 in List. 7.8). List. 7.8 also comprises two other Goals, which produce the global metamodel (lines 1–8) and the global instance (10–18) using the EMF TechSpace. The latter may be fed into other tools based on EMF, e.g. in order to do model repair.

```

1 goal CreateGlobalMetamodel {
2   correspondence PatientReferralDesign
3   action SCHEMA
4   technology ECORE
5   target FILE {
6     at globalMetamodel.ecore
7   }
8 }
9
10 goal CreateGlobalInstance {
11   correspondence PatientReferralDesign
12   action FEDERATION
13   matches traces.csv
14   technology ECORE
15   target FILE {
16     at output/globalInstance.xmi
17   }
18 }
19
20 goal CheckGlobalConsistency {
21   correspondence PatientReferralDesign
22   action VERIFY
23   matches traces.csv
24   target FILE {
25     at output/checkResult.txt
26   }
27 }

```

Listing 7.8: Final step: Goal definition

List. 7.6 and List. 7.7 only comprise two IdentificationRules (one for Requirements and one for BaseTypeS). This is because that in this scenario elements cannot easily be identified via their names or properties. Instead, this information has to be provided from the outside, e.g. by the domain experts. Therefore, the Goals in List. 7.8 are provided with a CSV file containing element matches, see Sec. 6.3.3. List. 7.9 shows the contents of this file. Every line represents an instance of a Commonality. The first slot mentions the name of the Commonality and the remaining slots contain element identifiers (i.e. Projections). The concrete format of the element identifiers depend on the respective TechSpace. In case of EMF, elements within an XMI-file (model) are identified via XQuery expressions.

Validation

```
"DecisionTableDef";"//@activites.2";"//@tables.0"  
"DataObjectClassAttrColumnImpl";"//@dataObjects.0";"//@packages.0/@classes.2";;  
"DataObjectClassAttrColumnImpl";"//@dataObjects.1";"//@packages.0/@classes.5/@attributes.1";"//@tables  
.0/@column.0"  
"DataObjectClassAttrColumnImpl";"//@dataObjects.2";"//@packages.0/@classes.4/@attributes.0";"//@tables  
.0/@column.2"  
"DataObjectClassAttrColumnImpl";;"//@packages.0/@classes.1/@attributes.0";"//@tables.0/@column.1"  
"DataObjectClassAttrColumnImpl";;"//@packages.0/@classes.4/@attributes.0";"//@tables.0/@column.3"  
"ProcessImpl";"//@elements.2";"/@bpmnDiagram"  
"Party";"//@elements.1";"//@pools.0"
```

Listing 7.9: traces.csv

Comparing the content of List. 7.9 with the links (dotted lines) in Fig. 1.8 and Fig. 1.9, it turns out that the file is missing a link between the Referrer-Role and the Referrer-Pool (link with number 5 in Fig. 1.8). Therefore, the consistency verification will result in an error. List. 7.10 shows how the output on the command line after invoking CORRLANG looks like. Adding the missing link to the CSV file would fix the inconsistency.

```
$> java -jar corrlang.jar patientReferralCase.corrlang g:CheckGlobalConsistency  
Executing Goal "CheckGlobalConsistency":  
  
Rule "RolesParticpateAsPools" is violated! Detailed message: "Role(s) not implemented as pools:  
Sequence {Referrer}"  
Total: 1 Inconsistencies
```

Listing 7.10: Command Line Output

As a conclusion, the initially stated goal of performing consistency verification over heterogeneous design documents has been achieved.

7.3.2 Features

To summarise the current features and limitations of CORRLANG, I will apply the feature model from Chap. 4. The result of this classification is shown in Tab. 7.1.

Currently, the implementation supports two concrete technologies GraphQL and EMF, which are implemented by respective technology adapters that translate meta-models (GraphQL schema definition language files or Ecore files) and their instances (JSON files or XMI files) into the formal representation used by the tool. The latter is given by generalised sketches, a formalism based on Category Theory. It is important to note that more Tech Spaces can be integrated by implementing a respective adapter, see Sec. 6.2.1.

CORRLANG considers Changes in a State-Based way and does not consider specific types of Allowed Updates. This is because, CORRLANG currently does not support Repair. For the next iteration it may be worthwhile to reconsider this aspect.

The number and structure of metamodels is Freely Customisable. Internally, the metamodels are interpreted as Type Graphs and the models are typed over them. CORRLANG is able to check Well-Formedness by validating Typing and local Constraints.

Extensive support for Multi-ary Correspondences was the central design goal behind the tool. The architecture is based on Global Views, which are constructed automatically. Commonalities are represented using a Weaving approach but CORRLANG also supports aspects of a Merging approach. The Definition of Commonalities is based on Structural

CORRLANG	
Models	
Tech Space	XMI, Other
Formalism	Category Theory
Change	
Representation	State-based
Recording	-
Allowed Updates	<i>not considered</i>
Conformance	
Well-Formedness	Typing, Constraints
Metamodels	
Definition	Customisable::Freely
Tech Space	MOF, DSL
Formalism	Type "Graph"
Correspondence	
Components	Heterogeneous::(Metamodels, Tech Spaces)
Commonalities	
Representation	Weaving
Definition	Customisable::(Structural Properties, Element Tuples)
Properties	-
Arity	Multi-ary
General Architecture	Global View::Constructed
Information Content	Symmetric
Authority	-
Concurrency	-
Privacy	-
Matching	
Invocation	Steady
Storage	In-Memory
Implementation	Keys
Consistency	
Inconsistency Report	Elements
Rules	
Definition	Builtin, Customisable: OCL++, ...
Nature	Structural
Scope	Inter-Model
Severity Levels	-
Repair Hints	-
Verification	
Execution	Automatic
Invocation	Manual
Implementation	Specialised Automated Analysis
Repair	<i>not considered</i>

Table 7.1: Feature Model Classification of CORRLANG

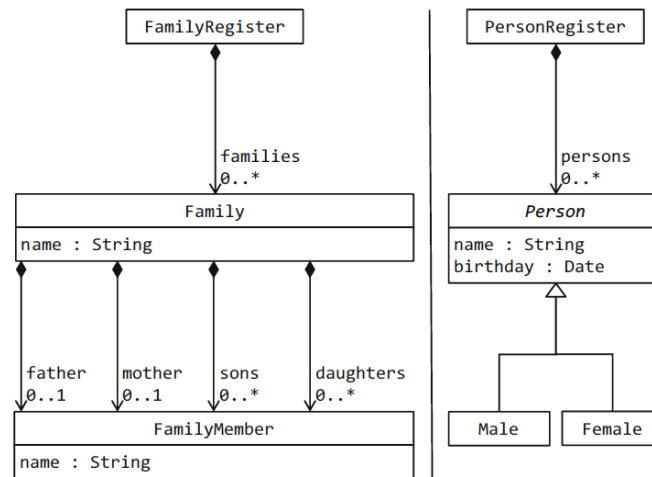


Fig. 7.6: Families2Persons [18]

Features (Key-based matching) or Element Tuples that are directly provided. Identified Commonalities are stored persistently on the Hard Drive.

The notion of Consistency is induced by consistency rules. There is Built-in support for FORALL and INTEGRITY constraints. Beyond that, consistency rules can be implemented Externally by utilising existing tools. Currently, Epsilon's OCL-like EVL language is supported but others can be included by implementing a respective adapter for the tool. The scope of these rules is Inter-Model and the focus lies on Structural rules. Verification is invoked Manually by calling a respective *Goal* and performed Automatically (i.e. without user interaction), and is executed On-Demand. The Implementation of the built-in constraints is based on a Specialised Automated Analysis, in case of the external rules it depends on the respective tool. The inconsistency report contains references to the Elements that violate a consistency rule.

The Repair feature is currently not supported directly and therefore has to be performed externally by exporting the global view on the multi-model instance into a specific TechSpace. The immediate next goal for the future is to support model repair more seamlessly.

7.3.3 Scalability

To test whether CORRLANG is able to scale up to real-world use-cases comprising models with several hundreds or thousands of elements, I will run a benchmark analysis and compare the run time of several model management tools and CORRLANG in a sample consistency verification scenario. This benchmark, including all related artefacts and instructions for reproducing it, is publicly available in a source code repository⁴. The scenario is based on the well-known *Families2Persons* case, which was first introduced as a showcase for ATL tool and has recently been used as a benchmark of BX tools [18]. It comprises two model spaces called FAMILIES and PERSONS, whose metamodels are depicted in Fig. 7.6.

The idea is that FamilyMember-elements in FAMILIES correspond to Person-

⁴<https://github.com/webminz/corrlang-performance>

elements in PERSONS and vice versa. Hence, a tuple consisting of a FAMILIES-model and a PERSONS-model is considered consistent if and only if for every FamilyMember exists a Male or Female with a matching name (Person.name is the concatenation of FamilyMember.name and Family.name) and sex (father/sons \Leftrightarrow Male and mother/daughters \Leftrightarrow Female respectively). The benchmark comprises a generator, which creates a pair of instances w.r.t. the metamodels in Fig. 7.6 and takes two parameters. The first parameter specifies how many (FamilyMember, Person) element pairs it should create. The second parameter specifies an “inconsistency percentage”, i.e. normally the generator populates the FAMILIES-model and a PERSONS-model with consistent pairs of FamilyMember and Person elements. The parameter specifies the probability for creating inconsistent elements. There are the following types of inconsistencies, which can be introduced.

- There exists no corresponding Person for a FamilyMember.
- There exists no corresponding FamilyMember for a Person.
- There exists the corresponding element but the sex does not match.
- There exists the corresponding element but the names do not match.

The choice of participants for this benchmark follows the selection of tools in Sec. 4.4. Hence, I am comparing CORRLANG with two solutions based on *Epsilon* and a fourth solution implemented with *eMoflon::Neo*. It was not possible to develop a solution with *Echo* since the *Families2Persons* case requires string concatenation, which is not supported by *Echo*. The principal ideas behind each solution are sketched below.

EPSILON SOLUTION (MERGE) The first Epsilon solution is based on the “consistency checking via merging” idea, compare Sec. 4.4.2. The workflow comprises three steps: First, the elements in the FAMILIES and PERSONS models are compared to identify the MatchTraces. Afterwards, the MatchTraces are used to create a merged model before, finally, constraints are verified on the merged models. Thus, one has to write an ECL program (matching), an EML program (merging) and EVL constraints (validation). This further necessitates to create a metamodel for the merge result (manually), which comprises all features shown in Fig. 7.6 and wherein the types FamilyMember and Person have been identified.

```

1 rule MatchRegistries
2   match l : FAMILIES!FamilyRegister
3   with r : PERSONS!PersonRegister {
4     compare : true
5   }
6
7 rule MatchMales
8   match l : Fam!FamilyMember
9   with r : Pers!Male {
10    guard : l.isMale()
11    compare : r.name = l.name + " " + l.getFamily().name
12  }
13
14 ...

```

Listing 7.11: ECL Matching Rules

Validation

List. 7.11 shows an excerpt of the ECL program, which shows the rule that matches male FamilyMembers (fathers or sons) with Males. The check tests whether the names match. The operations `isMale()` and `getFamily()` are helper methods (not shown) that retrieve the respective information from the containing Family element. The respective rule for Females looks similar and the rule that compares the two root objects (FamilyRegistry and PersonRegistry) always returns true since there will always be one instance of each type that always should be identified.

```
1 rule MergeRegistries
2   merge l : FAMILIES!FamilyRegister
3   with r : PERSONS!PersonRegister
4   into m : MERGE!Register {
5     m.families = l.families.equivalent();
6     m.persons = r.persons.equivalent();
7   }
8
9 rule MergeFemales
10  merge l : FAMILIES!FamilyMember
11  with r : PERSONS!Female
12  into m : MERGE!Female {
13    m.isMatched = true;
14    m.birthday = r.birthday;
15    m.fullname = r.name;
16    m.firstname = l.name;
17    m.motherInverse = l.motherInverse;
18    m.daughtersInverse = l.daughtersInverse;
19  }
20
21 ...
22
23 rule CopyUnmatchedFemaleFamilyMember
24  transform l : FAMILIES!FamilyMember
25  to m : Merge!Male {
26    guard : l.isFemale()
27    m.isMatched = false;
28    m.firstname = l.name;
29    m.motherInverse = l.motherInverse;
30    m.daughtersInverse = l.daughtersInverse;
31  }
32
33 rule CopyUnmatchedFemalePerson
34  transform r : Pers!Female
35  to m : Merge!Female {
36    m.isMatched = false;
37    m.birthday = r.birthday;
38    m.fullname = r.name;
39  }
40
41 ...
```

Listing 7.12: EML merge rules

List. 7.12 presents an excerpt of the EML program showing both merge and transform (copy) rules. The merge-rule is invoked for each found match. The rule creates a new Female element in the merged model. By default, EML does not transfer any feature values automatically. This must be defined manually within the body of the respective rule, which gives the designer full flexibility. In this case, the information present from FAMILIES and PERSONS simply gets copied. Additionally, a new attribute, called `isMatched`, is set to `true`, which will be used later in EVL program. Besides the merge rules, there are transform rules, which are invoked on all unmatched elements. These rules simply copy the available information into the merged model but set `isMatched` to “false”.

```

1 context MERGE!Person {
2   constraint isConsistent {
3     check : self.isMatched;
4   }
5 }

```

Listing 7.13: EVL constraint

Finally, List. 7.13 shows the EVL constraint, which is defined on the merged metamodel and checks that all elements in that model are the result of a correct match.

EPSILON SOLUTION (NO MERGE) *Epsilon* is not strictly tied to the “consistency verification via merging” workflow [134]. Since it is more a library comprising various DSLs supporting different model management task, various approaches can be implemented, see e.g. [168, 215, 413]. The second solution that I am presenting uses EVL alone. EVL can be used to define local and global consistency rules because the implementation of constraints allows arbitrary EOL statements querying multiple models simultaneously [288].

```

1 context PERSONS!Person {
2   constraint hasFamilyMember {
3     check : FAMILIES!FamilyMember.all()
4       .exists(fm|fm.name + " " + fm.getFamily().name = self.name and self.isTypeOf(Pers!Male) = fm.
5         isMale())
6     message : self.name + ' has no matching Family Member'
7   }
8 }
9 context FAMILIES!FamilyMember {
10  constraint hasPerson {
11    check : PERSONS!Person.all()
12      .exists(p|p.name = self.name + " " + self.getFamily().name and self.isMale() = p.isTypeOf(
13        Pers!Male))
14    message : self.name + ' has no matching Person'
15  }
16 }

```

Listing 7.14: EVL rules (No Merge)

List. 7.14 shows the respective solution (the helper-functions `isMale()`, `isFemale()`, `getFamily()` are now shown). It comprises two constraints attached to `FamilyMember` and `Person` respectively, which search for the matching partner element. For this, all elements of the other model are iterated, see `FamilyMember.all()/Person.all()`. This solution is simpler in design compare to the merge-based attempt. However, it does not store the matches persistently, which means that it operates in an incremental way.

EMOFLON SOLUTION The *eMoflon* solution is directly taken from the BX benchmark [18]. The consistency rules in this case are defined in a more indirect way, i.e. the developer has to define a triple graph grammar whose language corresponds to the semantic extension of the consistency rules of the *Families2Persons* case. The grammar is specified in a DSL (file ending `.msl`), see Sec. 4.4.3.

```

1 tripleGrammar FamiliesToPersons {
2   source {
3     Families
4   }
5 }

```

Validation

```
5
6 target {
7   Persons
8 }
9
10 correspondence {
11   FamilyRegister <-FamiliesToPersons->PersonRegister
12   FamilyMember<-FamilyMemberToPerson->Person
13 }
14
15 rules {
16   Families2Persons
17   MotherToFemale
18   MotherOfExistingFamilyToFemale
19   FatherToMale
20   FatherOfExistingFamilyToMale
21   DaughterToFemale
22   DaughterOfExistingFamilyToFemale
23   SonToMale
24   SonOfExistingFamilyToMale
25 }
26 }
```

Listing 7.15: *eMoflon* Schema.msl

List. 7.15 shows the content of the root definition file `Schema.msl`, which declares the source and target metamodels, the “corr”-types, and the rules that are part of the grammar. The listing shows that there are multiple rules to cover the various constellations of fathers, mothers, sons, and daughters. Emoflon avoids redundancy case via (multiple) rule-inheritance.

```
1 abstract tripleRule FamilyMember2Person : FamiliesToPersons {
2   source {
3     families : FamilyRegister {
4       ++ -families->f
5     }
6     ++ f : Family {
7       .name := <familyName>
8     }
9     ++ fm : FamilyMember {
10      .name := <memberName>
11    }
12  }
13
14  target {
15    persons : PersonRegister {
16      ++ -persons->p
17    }
18    ++ p : Person {
19      .name := <personName>
20    }
21  }
22
23  correspondence {
24    families <- :FamiliesToPersons -> persons
25    ++ fm <- :FamilyMemberToPerson -> p
26  }
27
28  attributeConstraints {
29    concat(
30      separator=" ",
31      left=<memberName>,
32      right=<familyName>,
33      combined=<personName>
34    )
35  }
36 }
```

Listing 7.16: Abstract FamilyMember2Person-rule

List. 7.16 shows the definition of the most important rule: FamilyMember2Person. This rule is marked as abstract and the other rules such as MotherOfExistingFamilyToFemale or Father2Male are concretisations of this rule.

CORRLANG SOLUTION

```

1 correspondence Fam2Pers (FAMILIES, PERSONS) {
2   sync (FAMILIES.FamilyMember as fm , PERSONS.Male as m) as syncMale
3     when [ fm.name ++ " " ++ fm.fatherInverse.name == m.name ||
4            fm.name ++ " " ++ fm.sonsInverse.name == m.name ];
5   sync (FAMILIES.FamilyMember as fm, PERSONS.Female as f) as syncFemale
6     when [ fm.name ++ " " ++ fm.motherInverse.name == f.name ||
7            fm.name ++ " " ++ fm.daughtersInverse.name == f.name ];
8 }

```

Listing 7.17: CORRLANG solution

The “heart” of the CORRLANG-solution, i.e. the Commonality-declaration, is shown in List. 7.17. This solution utilises the FORALL-constraint, see Sec. 6.3.2, which makes sure that every FamilyMember has a matching Person of the correct subtype. The matching is facilitated via IdentificationRules.

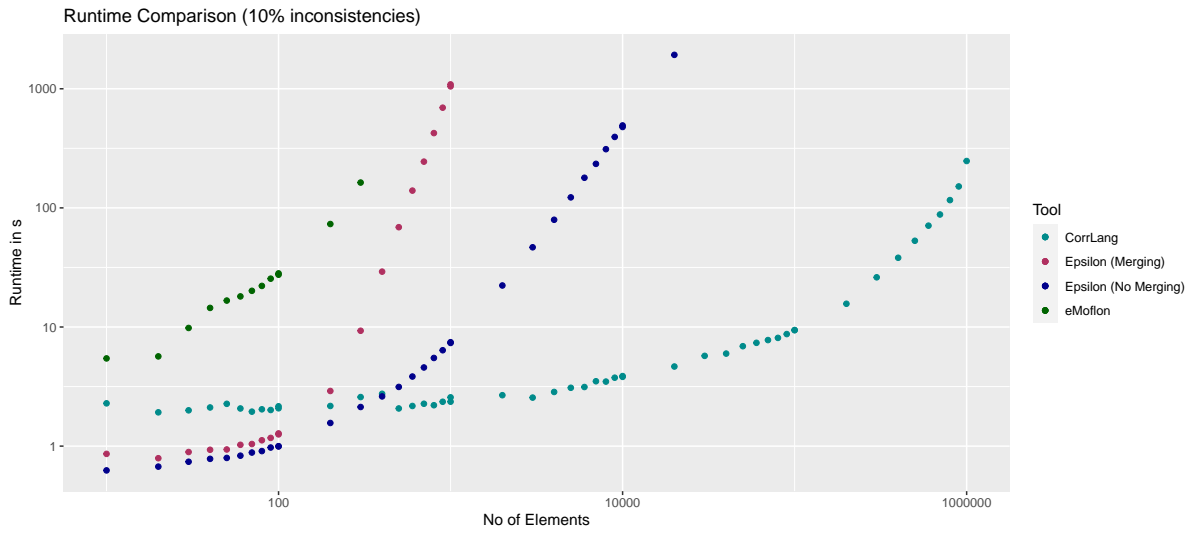
COMPARISON Before running the benchmark, one may compare the “complexity” of the four solution attempts. Tab. 7.2 summarises the number of files, lines, words, and characters. Tab. 7.2 gives a superficial view on the complexity of each approach and experts that are more proficient with one tool would assess the complexity differently.

	# Files	# Lines	# Words	# Chars
Epsilon (Merge)	3	116	367	2759
Epsilon (No Merge)	1	40	119	995
Emoflon	13	268	532	6535
CORRLANG	1	33	102	773

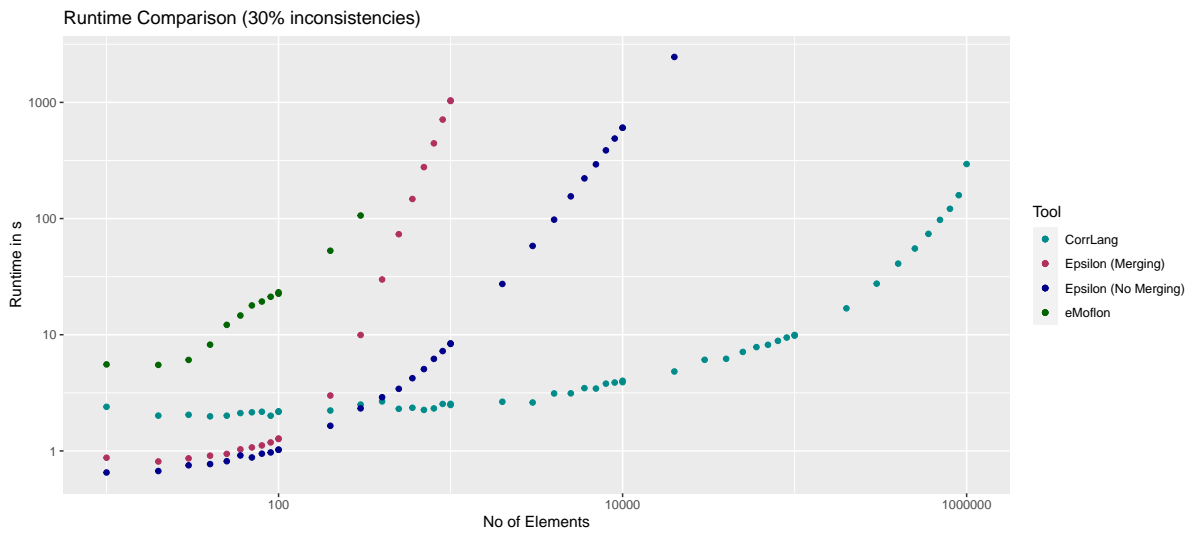
Table 7.2: Families2Persons: “Solution Complexity”

The result of running the benchmark for various model’ sizes and three different inconsistency-probabilities is shown in Fig. 7.7. The results are averaged over running the benchmark 10 times on a 2017 model of a MacBook Pro 2,3 GHz Dual-Core Intel Core i5 with 8GB RAM. The X-axis shows the number of model elements (FamilyMember, Person) and the Y-axis shows the runtime in seconds. Note that both axis have a logarithmic (\log_{10}) scale in order to better visualise the results for small and very large model sizes. Moreover, it is important to note that not all tools could be fed with equally big models: During my experiments, I experienced deadlock in the Neo4J graph database when running *emoflon::Neo* with models containing more than 400 elements. The *Epsilon* solutions showed very long run times at a certain point. Thus, in order to finish the benchmark in a reasonable time, I set the cut-off for the merge-based *Epsilon* solution to 1000 and for the other *Epsilon* solution to 20000.

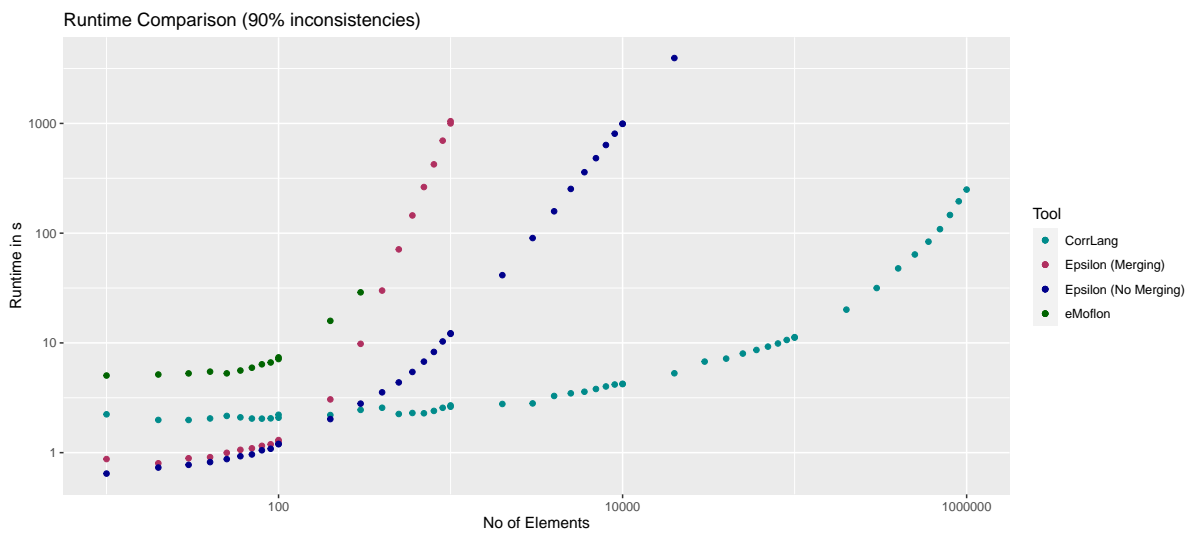
Validation



(a) 10% inconsistencies



(b) 30% inconsistencies



(c) 90% inconsistencies

Fig. 7.7: Runtime comparison

All three graphs show a similar (quadratic) trend. The slightly shorter execution times for a high probability of inconsistencies may be due to the possibility to abort the execution of the concrete “check”-function after the first inconsistency has been found. The data points show a quadratic growth for each tool, which makes sense concerning that an ad-hoc implementation of the *Families2Persons* consistency rules will involve two nested loops for the comparison in both directions. This benchmark gives a first impression of the scalability capabilities of CORR_{LANG}. Yet, further experiments with more realistic examples remain to be executed in the future. The main takeaway is that CORR_{LANG} is able to compete with established tools.

Part III
OUTRO

“We seldom think of what we have, but always of what we lack.”

—Arthur Schopenhauer

CHAPTER 8

CONCLUSION

8.1 Summary

The starting point of this whole research endeavour was given by four research questions, see Sec. 1.4: The first research question **RQ1** asked how multi-models and consistency rules among them can be represented. The second research question **RQ2** asked how consistency verification can be performed. The third research question **RQ3** asked how the consistency restoration can be performed. The fourth research question **RQ4** asked for an overarching architecture and workflow for coordinating these three aspects. Each question refers to specific sub-problems of multi-model consistency management. Their formulation embodies the constructive nature of SE research, i.e. asking for a problem solution (artefact) and not an explanation of reality (empirical theory). Three such artefacts have been developed throughout this thesis, see Sec. 2.4: **A1** provides a solution on the most abstract conceptual level, **A2** provides a solution on a formal theoretical level, and **A3** provides a concrete technical solution.

The grid in Tab. 8.1 provides an abstract overview over the artefacts, research questions, and the extent to which an artefact answers a research question, which is expressed through the symbolism “0, (✓), ✓” representing a three-element total order. The value n/a indicates that the respective research question is not applicable to that particular artefact.

The conceptual model (**A1**) addresses all aspects of multi-model consistency management on a high level. It provides a language for talking about problem scenarios and a feature list for comparing tools and approaches. Thus, the results of the literature study in Chap. 4 must be counted towards the contributions of this artefact. It may serve as a decision support for engineers developing multi-model consistency management tools as it offers a condensed overview of the various implementation strategies and available solutions. An accompanying literature study, which is pre-

	A1 (Conceptualisation)	A2 (Formalism)	A3 (Tool)
RQ1 (Representation)	✓	✓	✓
RQ2 (Verification)	✓	(✓)	(✓)
RQ3 (Restoration)	✓	(✓)	0
RQ4 (Architecture)	✓	n/a	✓

Table 8.1: Contributions and Research Questions

Conclusion

sented in Chap. 4, identified several limitations of those existing solutions, which are summarised in Sec. 4.5. The most serious limitation is the lack of support for *multi-ary* multi-model correspondence relationships and rules over them. My other two more concrete artefact contributions have been created to address this and other limitations stated in Sec. 4.5.

Comprehensive System (A₂) present a formalism for representing multi-models. The aspects of verification and restoration are addressed by proving that comprehensive systems show all the formal properties that are necessary to re-use and apply existing formal approaches for verification and restoration. However, I did not develop a custom formal verification or restoration procedure, therefore the symbol “(✓)”.

The design of CORRLANG (A₃) is based on the conceptual framework and the comprehensive system formalism. It implements all features necessary for representing multi-model consistency management including rudimentary support for model matching using keys, the possibility to integrate existing (formal) verification tools, and built-in support for the most common constraints concerning the consistency of multi-models. Consistency restoration is not supported yet. However, CORRLANG allows to export the global view of a multi-model as a single artefact to one of the supported technologies, which allows to apply model repair tools for that technology on the exported artefact. The latter represents a more general feature of CORRLANG namely to translate among heterogeneous technological spaces utilising the formalism of Chap. 5 (generalised sketches over graph-like structures).

In conclusion, this thesis provides a self-contained framework to “attack” multi-model consistency management problems, where the focus certainly lies on the formal theoretical side. Nonetheless, a demonstration of an (efficient) implementability of these theoretical ideas is given in form of a purposeful tool, see Chap. 7.

8.2 Threats to Validity

One of the final steps of each scientific inquiry is a reflection over the threats to the validity of the obtained results. The literature on research methodology [333, 492] distinguishes four aspects of validity: Construct validity, internal validity, external validity, and conclusion validity. Yet, their definition is based on the classical understanding of empirical research methods and must therefore be adjusted for my concrete case of constructive research.

Internal validity looks into the application of the research methods itself, e.g. in empirical research: “Has the experiment been conducted as described or are there outside factors that could have influenced the results?”. In the case of this thesis, this aspect can be interpreted as follows: “Has the process always followed the respective state-of-the-art and is it reproducible?” Regarding the development of the conceptual model, this question is actually difficult to answer as this is mainly a creative process. However, the conceptual model must be linked to the literature study. The latter followed the best practices identified by systematic literature reviews and systematic mapping studies, even though it did not follow those processes in all rigour. There is one threat to consider here. The selection of the initial survey papers that were used to aggregate the total list of studies resulted from search queries on *Google Scholar* and *DBLP* using terms I introduced in the conceptual framework. There is a chance

that I miss a study, which addresses multi-model consistency management. The formal investigation in Chap. 5 was conducted with mathematical rigour and all proofs have been reviewed by experts in theoretical computer science and category theory. The development of the prototype was conducted following state-of-the-art software engineering practices utilising unit tests (Test Driven Development) and code quality metrics. The implementation followed the formal foundation closely and all source code is publicly available for further assessment.

Construct validity looks into the correspondence between theory and the real problem domain. The construction of each of the three artefacts always started with a practical problem or requirement.

Conclusion validity looks into the actual *effect* of the scientific procedure. In quantitative studies, this is usually validated by ensuring statistical significance. In my case, it is of interest whether the created artefacts actually represent novel or improved solutions. This particular discussion is the content of Chap. 7.

Finally, *external validity* refers to the ability to generalise the results. The conceptual framework and the formalism are able to describe many specific approaches. The prototype tool was designed in a modular way and comprises a plug-in mechanism, which shall make it extendable in the future,

8.3 Related Work

The introductory section of Chap. 3 mentions all research domains related to multi-model consistency management and Chap. 4 provides a high-level summary about them. Furthermore, Chap. 4 contains an in-depth analysis of three tools, which act as representatives for the “primary” approaches to the multi-model consistency management problem, i.e. *Echo* as a representative for approaches based on formal logic, *Epsilon* as a representative for (language) engineering-based approaches, and *Emoflon* as a representative for formal graph-based approaches. The current limitations of these tools served as the motivation for my solution and have been discussed in detail. In the following, I want to mention industrial solutions and academic approaches that have received less attention in this thesis but also classify as related work and would be worthwhile to investigate more closely in the future.

8.3.1 Industrial Solutions

ENTERPRISE DEVELOPMENT FRAMEWORKS Contemporary software libraries and frameworks that facilitate developing enterprise software applications comprise an abstraction layer for data access. This abstraction layer generalises the concept of object-relational mappings in a way that it mediates the representation based on classes in an object-oriented programming language with heterogeneous data sources such as SQL databases or XML documents. Examples are given by *Spring Data* or Microsoft’s *Language Integrated Queries (LINQ)* embedded in the *.NET*-platform. The mapping between the classes in the application and the data types of the respective schema is declaratively specified by *annotations* that are placed in the program code. Moreover, the aforementioned frameworks feature an SQL-like query language for accessing data in a uniform way. Thus, there are conceptual similarities between these enterprise

Conclusion

frameworks and CORR`LANG`. The main difference is that enterprise application frameworks operate on a lower abstraction level than CORR`LANG`. It would be worthwhile to elucidate whether CORR`LANG` and enterprise frameworks can be applied in combination. For example, CORR`LANG` could be used to check the annotation-definitions, which are scattered across multiple fields, against high-level design documents.

EXTRACT TRANSFORMATION LOAD Enterprise data warehouses have to collect the data stored across various information systems of an organisation in order to enable decision makers to create reports about their business. This requires to extract the heterogeneous data from the individual information systems and to copy it into a single database, which is optimised for aggregating queries. The respective workflow is called *Extract-Transform-Load (ETL)* and is commonly facilitated by commercial tools such as *Informatica* and *Talend*. These tools offer adapters for a wide range of technologies (e.g. SQL databases, XML, Excel sheets, web services, ...) and graphical editors for defining mappings between the source schemas and the data warehouse schema. Those mappings are similar to the definition of commonalities in the CORR`LANG`DSL. However, the primary use cases for ETL tools and CORR`LANG` differ: The data-flow direction in ETL goes one way from multiple sources to a single target. CORR`LANG` considers multi-ary and multi-directional relationships among data elements.

8.3.2 Academic Approaches

VITRUVIUS *Vitruvius* [280] is an approach to view-based system development utilising the idea of orthographic software modeling [25]. The latter is centred around the idea of a *single underlying model (SUM)* representing the system, which is synthetically assembled from several artefacts and modified solely through (projective) views. In *Vitruvius*, the SUM exists only virtually, i.e. it is derived from so-called *consistency preservation rules*, which are established between the metamodels of the concrete artefacts that make up the SUM. CORR`LANG` and *Vitruvius* address the same problem domain and there are several conceptual similarities. The concept of a federation can be compared to a virtual SUM and the consistency preservation rules are closely related to the commonalities that make up a CORR`LANG` specification. *Vitruvius* features automatic consistency restoration, with the possibility to define views on top of the SUM, it comprises an extra layer, and also appears more mature since it has been around for longer. However, the consistency preservation rules that synchronise the concrete artefacts at the bottom are limited to binary relations. A more detailed comparison of both approaches in the future is desirable.

CATEGORICAL DATABASES (CQL) In [420, 421], Schultz et al. present an approach for data representation and integration which is based on concepts from category theory and implemented in a tool called *Categorical Query Language (CQL)*. Database schemas are formally interpreted as a category and database instances are interpreted as functors. Thus, the tool can be seen as the conceptual successor of the *Easik*¹ tool. The main application domain are SQL databases but integration for other technologies also exist. CQL supports automatic data-migration, which is formally underpinned by the

¹<https://www.mta.ca/~rrosebru/project/Easik/index.html>

change-of-base functor and its left and right adjoints. These theoretical constructions give rise to an SQL-like query language. Similarly, CORR_{LANG} addresses data integration and is conceptually based on comprehensive systems, which are formulated with the help of category theory as well. However, CQL only considers binary relationships. Moreover, it relies on the existence of left- and right-adjoints to the change-of-base functor, which is not generally given in the category that I am considering in my framework.

THE HETEROGENEOUS TOOL SET (HETS) The *heterogeneous toolset (Hets)* [347] was shortly mentioned in Chap. 5. It is a “meta-tool” for formal verification, which allows to combine solvers and theorem provers that are based heterogeneous formalisms. Theoretically, it is based on the formal concept of *institutions* where various logic-systems such as propositional logic, first-order logic, temporal logic etc. can be related via institution co-morphisms. It is further integrated with the *Distributed Ontology, Model and Specification Language (DOL)*, which is standardised by the OMG and targeted especially towards semantic web technologies. Comprehensive systems, which are the formal underpinning for CORR_{LANG}, can be embedded into the abstract framework of institutions. In the future, it is worthwhile to work towards an integration with Hets, such that complex consistency rules can be checked utilising various formal tools.

8.4 Future Work

The contributions of thesis are only one “small step” into the direction of “solving” the multi-model consistency management problem once and for all. There are several *open issues* and *unfinished ideas* that arise immediately from the presentation in this thesis, e.g. the open questions in the theoretical framework (Sec. 5.4) and the missing features for CORR_{LANG} (Sec. 6.4). Here, I want to mention the most important topics for future work.

EMPIRICAL EVALUATION The practical validation of the artefacts was conducted in an academic setting only, see Chap. 7. In the future it will be important to evaluate the conceptual, formal, and practical contributions of this thesis in an industrial setting. This usually requires substantial additional effort, but the insights from a bigger case study will provide important evidence of my contributions and uncover open issues. Ideally, the practical evaluation would be performed in a cross-disciplinary project (e.g. including social scientists) to collect qualitative data from the end users (software engineers).

MULTI-MODEL REPAIR The topic of multi-model repair was dealt with on an abstract level and under the assumption that tools for the “classic” model repair problem exist which can be reused. In particular, I put a focus on the algebraic graph transformation framework in Chap. 5. In general, model repair is a vast topic and an ideal universal solution is unlikely to exist, such that for each concrete case, domain specific expertise is required to build efficient model repair solutions that fulfil given quality requirements such as *least change* or *least surprise*. Multi-model repair is an interesting special case of

Conclusion

model repair since it often comprises a special kind of consistency rules, e.g. FORALL or INTEGRITY, see Sec. 6.3.2. For these constraints, general repair strategies can be formulated. It will be worthwhile to pursue this topic further both on the formal level (*comprehensive systems*) and the implementation level (CORRLANG).

BEHAVIOURAL SEMANTICS The software models encountered throughout the thesis were all *structural*. Thus, I formally conceived them as some sort of graph-like structure (category \mathbb{G}). The semantics of a given (meta-)model were interpreted by the set of all possible instances, i.e. graph-like structures that can be typed over the given model (formally given by a *slice category*). This conception of semantics is called *static*. In the future, it will be interesting to also consider models with behavioural semantics such as state charts. Their semantics is usually expressed differently (e.g. via *Kripke structures*), the rules are formulated in a *temporal* way and their composition is based on a different concept (limit-based instead of colimit-based).

NETWORKS OF GLOBAL VIEWS Comprehensive systems and their practical reification in the CORRLANG-tool are able to model an integration of multiple models (systems). This architecture is centralised, i.e. each comprehensive system induces a single global view that is used for coordination. Now, assuming that multiple organisations apply this idea and afterwards decide to integrate their models or systems with each other, forcing them to integrate their system landscapes via a single hub is certainly unfeasible. Thus, it will be important to investigate how comprehensive systems can be integrated themselves such that one can create a *layered architecture*, similar to the architecture of the internet, i.e. mediating the network and the centralised approach with each other. In this context, approaches for information hiding must be investigated as well.

8.5 Conclusion

This thesis just presented a *framework* (= concepts, theory, and tool) for multi-model consistency management that embeds itself into the existing body of knowledge. Notwithstanding, challenging theoretical and practical questions remain. Hence, “the contents of this thesis should rather be considered a starting point . . . than the final document of this research issue”².

²This is a direct citation of the final sentence in [318]

BIBLIOGRAPHY

- [1] F. Abou-Saleh, J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. Introduction to Bidirectional Transformations. In J. Gibbons and P. Stevens, editors, *Bidirectional Transformations: International Summer School, 2016*, LNCS, pages 1–28. Springer International Publishing, 2018.
- [2] J.-R. Abrial. Formal methods in industry: achievements, problems, future. In *Proceedings of the 28th international conference on Software engineering, ICSE '06*, pages 761–768, New York, NY, USA, May 2006. Association for Computing Machinery.
- [3] J. Adámek, H. Herrlich, and G. Strecker. *Abstract and concrete categories: the joy of cats*. Pure and applied mathematics. Wiley, 1990.
- [4] J. Adamek and J. Rosicky. *Locally Presentable and Accessible Categories*. London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge, 1994.
- [5] J. Adámek and H. Herrlich. Cartesian closed categories, quasitopoi and topological universes. *Commentationes Mathematicae Universitatis Carolinae*, 27(2):235–257, 1986. Publisher: Charles University in Prague, Faculty of Mathematics and Physics.
- [6] M. A. Ahmad and A. Nadeem. Consistency checking of UML models using Description Logics: A critical review. In *2010 6th International Conference on Emerging Technologies (ICET)*, pages 310–315, Oct. 2010.
- [7] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3):515–526, 2006.
- [8] D. H. Akehurst and S. Kent. A Relational Approach to Defining Transformations in a Metamodel. In *Proceedings of the 5th International Conference on The Unified Modeling Language, UML '02*, pages 243–258, Berlin, Heidelberg, Sept. 2002. Springer-Verlag.
- [9] J. E. v. Aken. Management Research Based on the Paradigm of the Design Sciences: The Quest for Field-Tested and Grounded Technological Rules. *Journal of Management Studies*, 41(2):219–246, 2004.
- [10] M. Alanen and I. Porres. Difference and union of models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference*, volume 2863 of LNCS, pages 2–17. Springer, 2003.
- [11] F. Allilaire, J. Bézinvin, H. Bruneliere, and F. Jouault. Global Model Management in Eclipse GMT/AM3. In *Eclipse Technology eXchange Workshop (eTX) - a ECOOP 2006 Satellite Event*, Nantes, France, July 2006.

BIBLIOGRAPHY

- [12] N. Alshuqayran, N. Ali, and R. Evans. A Systematic Mapping Study in Microservice Architecture. In *SOCA 2016*, pages 44–51, Nov. 2016.
- [13] M. Alvarez-Picallo and C.-H. L. Ong. Change Actions: Models of Generalised Differentiation. In M. Bojańczyk and A. Simpson, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 45–61. Springer International Publishing, 2019.
- [14] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture – Foundations and Applications*, Lecture Notes in Computer Science, pages 361–375, Berlin, Heidelberg, 2006. Springer.
- [15] S. Ananieva, S. Greiner, T. Kühn, J. Krüger, L. Linsbauer, S. Grüner, T. Kehrer, H. Klare, A. Koziolok, H. Lönn, S. Krieter, C. Seidl, S. Ramesh, R. Reussner, and B. Westfechtel. A conceptual model for unifying variability in space and time. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, SPLC '20, pages 1–12, New York, NY, USA, Oct. 2020. Association for Computing Machinery.
- [16] M. Andreessen. Why Software Is Eating The World. *Wall Street Journal*, Aug. 2011.
- [17] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, Apr. 1999.
- [18] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H.-S. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, and A. Zündorf. Benchmarking bidirectional transformations: theory, implementation, application, and assessment. *Software and Systems Modeling*, Sept. 2019.
- [19] M. Antkiewicz and K. Czarnecki. Design Space of Heterogeneous Synchronization. In R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE 2007*, Lecture Notes in Computer Science, pages 3–46. Springer, Berlin, Heidelberg, 2008.
- [20] M. A. Arbib and E. G. Manes. *Arrows, Structures, and Functors: The Categorical Imperative*. Academic Pr., New York, 1st edition, Feb. 1975.
- [21] M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Foundations of Data Exchange*. Cambridge University Press, Cambridge, 2014.
- [22] C. H. Asuncion and M. J. van Sinderen. Pragmatic Interoperability: A Systematic Review of Published Definitions. In P. Bernus, G. Doumeingts, and M. Fox, editors, *Enterprise Architecture, Integration and Interoperability*, IFIP Advances in Information and Communication Technology, pages 164–175, Berlin, Heidelberg, 2010. Springer.

- [23] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, Sept. 2003. Conference Name: IEEE Software.
- [24] C. Atkinson and T. Kühne. The Essence of Multilevel Metamodeling. In M. Gogolla and C. Kobryn, editors, *UML 2001*, LNCS, pages 19–33, Berlin, Heidelberg, 2001. Springer.
- [25] C. Atkinson, D. Stoll, and P. Bostan. Orthographic Software Modeling: A Practical Approach to View-Based Development. In L. A. Maciaszek, C. González-Pérez, and S. Jablonski, editors, *ENASE 2009*, Communications in Computer and Information Science, pages 206–219. Springer Berlin Heidelberg, 2010.
- [26] U. Aßmann, S. Zschaler, and G. Wagner. Ontologies, Meta-models, and the Model-Driven Paradigm. In C. Calero, F. Ruiz, and M. Piattini, editors, *Ontologies for Software Engineering and Software Technology*, pages 249–273. Springer, Berlin, Heidelberg, 2006.
- [27] J. Backus. The history of Fortran I, II, and III. *IEEE Annals of the History of Computing*, 20(4):68–78, Oct. 1998. Conference Name: IEEE Annals of the History of Computing.
- [28] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger, and P. Naur. Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 3(5):299–314, May 1960.
- [29] J. Bailey, D. Budgen, and M. Turner. Evidence relating to Object-Oriented software design: A survey. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2007.
- [30] R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th international conference on Software engineering, ICSE '91*, pages 158–165, Washington, DC, USA, May 1991. IEEE Computer Society Press.
- [31] F. Bancilhon and N. Spyrtos. Update Semantics of Relational Views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.
- [32] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching Lenses: Alignment and View Update. In *ICFP '10*, pages 193–204, New York, NY, USA, 2010. ACM.
- [33] N. Barnickel and M. Fluegge. Towards a conceptual framework for semantic interoperability in service oriented architectures. In *Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications, ISWSA '10*, pages 1–7, New York, NY, USA, June 2010. Association for Computing Machinery.
- [34] M. Barr and C. Wells. *Category theory for computing science*. Prentice Hall, 1990.

BIBLIOGRAPHY

- [35] A. Barriga, R. Heldal, L. Iovino, M. Marthinsen, and A. Rutle. An extensible framework for customizable model repair. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '20*, pages 24–34, New York, NY, USA, Oct. 2020. Association for Computing Machinery.
- [36] A. Barriga, A. Rutle, and R. Heldal. Automatic model repair using reinforcement learning. In *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018.*, pages 781–786, 2018.
- [37] R. S. Bashir, S. P. Lee, S. U. R. Khan, V. Chang, and S. Farid. UML models consistency management: Guidelines for software quality manager. *International Journal of Information Management*, 36(6, Part A):883–899, Dec. 2016.
- [38] V. Basili, L. Briand, D. Bianculli, S. Nejati, F. Pastore, and M. Sabetzadeh. Software Engineering Research and Industry: A Symbiotic Relationship to Foster Impact. *IEEE Software*, 35(5):44–49, Sept. 2018. Conference Name: IEEE Software.
- [39] A. Bastiani and C. Ehresmann. Categories of sketched structures. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 13(2):104–214, 1972.
- [40] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.
- [41] S. Becker, H. Koziolk, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, Jan. 2009.
- [42] S. Bennani, S. Ebersold, M. El Hamlaoui, B. Coulette, and M. Nassar. A Collaborative Decision Approach for Alignment of Heterogeneous Models. In *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 112–117, June 2019. ISSN: 2641-8169.
- [43] J. Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, Aug. 1986.
- [44] G. Bergmann, I. Ráth, G. Varró, and D. Varró. Change-driven model transformations. *Software & Systems Modeling*, 11(3):431–461, July 2012.
- [45] P. A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *CIDR*, 2003.
- [46] P. A. Bernstein, A. Y. Halevy, and R. A. Pottinger. A vision for management of complex models. *ACM SIGMOD Record*, 29(4):55–63, Dec. 2000.
- [47] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In U. Aßmann, M. Aksit, and A. Rensink, editors,

- Model Driven Architecture: European MDA Workshops: Foundations and Applications, MDFA 2003 and MDFA 2004*, pages 33–46, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [48] E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *MODEL 2008*, pages 53–67, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [49] N. Bikakis, C. Tsinaraki, N. Gioldasis, I. Stavrakantonakis, and S. Christodoulakis. The XML and Semantic Web Worlds: Technologies, Interoperability and Integration: A Survey of the State of the Art. In I. E. Anagnostopoulos, M. Bieliková, P. Mylonas, and N. Tsapatsoulis, editors, *Semantic Hyper/Multimedia Adaptation: Schemes and Applications*, Studies in Computational Intelligence, pages 319–360. Springer, Berlin, Heidelberg, 2013.
- [50] D. Bjørner and K. Havelund. 40 Years of Formal Methods. In C. Jones, P. Pihlajasaari, and J. Sun, editors, *FM 2014: Formal Methods*, Lecture Notes in Computer Science, pages 42–61. Springer International Publishing, 2014.
- [51] J. Boardman and B. Sauser. System of Systems - the meaning of of. In *2006 IEEE/SMC International Conference on System of Systems Engineering*, pages 6 pp.–, Apr. 2006.
- [52] B. W. Boehm. Software Engineering. *IEEE Trans. Comput.*, 25(12):1226–1241, Dec. 1976.
- [53] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. *ACM SIGPLAN Notices*, 43(1):407–419, Jan. 2008.
- [54] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, PODS '06*, pages 338–347, New York, NY, USA, June 2006. Association for Computing Machinery.
- [55] F. Bonchi, F. Gadducci, A. Kissinger, P. Sobociński, and F. Zanasi. Confluence of Graph Rewriting with Interfaces. In H. Yang, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 141–169, Berlin, Heidelberg, 2017. Springer.
- [56] G. Booch. The History of Software Engineering. *IEEE Software*, 35(5):108–114, Sept. 2018. Conference Name: IEEE Software.
- [57] A. Boronat, A. Knapp, J. Meseguer, and M. Wirsing. What Is a Multi-modeling Language? In *WADT 2008*, pages 71–87, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [58] A. Boronat and J. Meseguer. Algebraic Semantics of OCL-Constrained Metamodel Specifications. In M. Oriol and B. Meyer, editors, *Objects, Components, Models and*

BIBLIOGRAPHY

- Patterns*, Lecture Notes in Business Information Processing, pages 96–115, Berlin, Heidelberg, 2009. Springer.
- [59] A. Boronat and J. Meseguer. An algebraic semantics for MOF. *Formal Aspects of Computing*, 22(3):269–296, May 2010.
- [60] P. Bourque, R. E. Fairley, and I. C. Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, Washington, DC, USA, 3rd edition, 2014.
- [61] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2nd edition, 2017.
- [62] Brooks. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, Apr. 1987. Conference Name: Computer.
- [63] F. P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley Professional, Reading, Mass, 2 edition edition, Aug. 1995.
- [64] M. Broy. Yesterday, Today, and Tomorrow: 50 Years of Software Engineering. *IEEE Software*, 35(5):38–43, Sept. 2018. Conference Name: IEEE Software.
- [65] M. Broy, M. V. Cengarle, H. Grönniger, and B. Rumpe. Considerations and Rationale for a UML System Model. In *UML 2 Semantics and Applications*, pages 43–60. John Wiley & Sons, Ltd, 2009.
- [66] M. Broy, M. V. Cengarle, H. Grönniger, and B. Rumpe. Definition of the System Model. In K. Lano, editor, *UML 2 Semantics and Applications*, pages 63–91. John Wiley & Sons, Ltd, 2009.
- [67] C. Brun and A. Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.
- [68] H. Bruneliere, E. Burger, J. Cabot, and M. Wimmer. A feature-based survey of model view approaches. *Software & Systems Modeling*, 18(3):1931–1952, June 2019.
- [69] H. Bruneliere, J. G. Perez, M. Wimmer, and J. Cabot. EMF Views: A View Mechanism for Integrating Heterogeneous Models. In P. Johannesson, M. L. Lee, S. W. Liddle, A. L. Opdahl, and O. Pastor López, editors, *Conceptual Modeling, Lecture Notes in Computer Science*, pages 317–325, Cham, 2015. Springer International Publishing.
- [70] G. Brunet, M. Chechik, S. Easterbrook, S. Nejati, N. Niu, and M. Sabetzadeh. A Manifesto for Model Merging. In *GaMMa '06*, pages 5–12, New York, NY, USA, 2006. ACM.
- [71] R. Bruni, J. Meseguer, and U. Montanari. Symmetric monoidal and cartesian double categories as a semantic framework for tile logic. *Mathematical Structures in Computer Science*, 12(1):53–90, Feb. 2002.

- [72] A. Bucchiarone, J. Cabot, R. F. Paige, and A. Pierantonio. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*, 19(1):5–13, Jan. 2020.
- [73] T. Buchmann. BXtend - A Framework for (Bidirectional) Incremental Model Transformations. In *MODELSWARD 2019 Proceedings*, pages 336–345, Oct. 2019.
- [74] D. Budgen, M. Turner, P. Brereton, and B. Kitchenham. Using Mapping Studies in Software Engineering. In *Proceedings of PPIG*, volume 8, 2008.
- [75] J. Bénabou. Introduction to bicategories. In J. Bénabou, R. Davis, A. Dold, J. Isbell, S. MacLane, U. Oberst, and J. E. Roos, editors, *Reports of the Midwest Category Seminar*, Lecture Notes in Mathematics, pages 1–77, Berlin, Heidelberg, 1967. Springer.
- [76] J. Bézivin. In search of a Basic Principle for Model-Driven Engineering. *Novatica – Special Issue on UML (Unified Modeling Language)*, 5(2):21–24, 2004.
- [77] J. Bézivin, S. Bouzitouna, M. D. Del Fabro, M.-P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R. F. Paige. A Canonical Scheme for Model Composition. In A. Rensink and J. Warmer, editors, *Model Driven Architecture – Foundations and Applications*, Lecture Notes in Computer Science, pages 346–360. Springer Berlin Heidelberg, 2006.
- [78] J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J. E. Rougui. First experiments with the ATL model transformation language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.
- [79] J. Bézivin, F. Jouault, and P. Valduriez. On the Need for Megamodels. In *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications,(2004)*, Vancouver, Canada, Oct. 2004.
- [80] J. Cabot and M. Gogolla. Object Constraint Language (OCL): A Definitive Guide. In M. Bernardo, V. Cortellessa, and A. Pierantonio, editors, *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, Lecture Notes in Computer Science, pages 58–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [81] A. Carboni, S. Lack, and R. F. C. Walters. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, 84(2):145–158, Feb. 1993.
- [82] R. Carnap. The elimination of metaphysics through logical analysis of language. *Erkenntnis*, pages 60–81, 1932.
- [83] M. Castells. *The Information Age, Volumes 1-3: Economy, Society and Culture*. Wiley, July 1999.

BIBLIOGRAPHY

- [84] M. Chechik, S. Nejati, and M. Sabetzadeh. A relationship-based approach to model integration. *Innovations in Systems and Software Engineering*, 8(1):3–18, Mar. 2012.
- [85] P. P.-S. Chen. The Entity-relationship Model—Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, Mar. 1976.
- [86] J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. Towards a Principle of Least Surprise for Bidirectional Transformations. *Proceedings of the 4th International Workshop on Bidirectional Transformations co-located with Software Technologies: Applications and Foundations (STAF 2015)*, 1396:66–80, 2015.
- [87] A. Cicchetti, F. Ciccozzi, and A. Pierantonio. Multi-view approaches for software and system modelling: a systematic literature review. *Software and Systems Modeling*, 18(6):3207–3233, Dec. 2019.
- [88] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. JTL: A Bidirectional and Change Propagating Transformation Language. In B. Malloy, S. Staab, and M. van den Brand, editors, *SLE 2011*, Lecture Notes in Computer Science, pages 183–202. Springer Berlin Heidelberg, 2011.
- [89] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Dependent Changes in Coupled Evolution. In R. F. Paige, editor, *Theory and Practice of Model Transformations*, Lecture Notes in Computer Science, pages 35–51, Berlin, Heidelberg, 2009. Springer.
- [90] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, Sept. 2008.
- [91] A. Cleve, E. Kindler, P. Stevens, and V. Zaytsev. Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491). *Dagstuhl Reports*, 8(12):1–48, 2019.
- [92] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM*, 13(6):377–387, June 1970.
- [93] B. Coecke and A. Kissinger. *Picturing Quantum Processes*. Cambridge University Press, Mar. 2017.
- [94] E. Commission. The New European Interoperability Framework, Feb. 2017.
- [95] I. C. S. S. C. Committee. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, 610. ANSI / IEEE Std. IEEE, 1990.
- [96] S. Cook. The UML Family: Profiles, Prefaces and Packages. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 — The Unified Modeling Language*, Lecture Notes in Computer Science, pages 255–264, Berlin, Heidelberg, 2000. Springer.
- [97] J. M. Corbin and A. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, Mar. 1990.

- [98] A. Corradini, D. Duval, R. Echahed, F. Prost, and L. Ribeiro. AGREE – Algebraic Graph Rewriting with Controlled Embedding. In F. Parisi-Presicce and B. Westfechtel, editors, *Graph Transformation*, Lecture Notes in Computer Science, pages 35–51. Springer International Publishing, 2015.
- [99] A. Corradini, T. Heindel, F. Hermann, and B. König. Sesqui-pushout rewriting. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations*, pages 30–45, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [100] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of graph grammars and computing by graph transformation*, pages 313–400. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.
- [101] J. W. Creswell and J. D. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications, Nov. 2017.
- [102] K. Czarnecki. Overview of Generative Software Development. In J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, editors, *Unconventional Programming Paradigms*, Lecture Notes in Computer Science, pages 326–341, Berlin, Heidelberg, 2005. Springer.
- [103] K. Czarnecki, N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *ICMT 2009*, pages 193–204, 2009.
- [104] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.
- [105] H. K. Dam and M. Winikoff. Supporting change propagation in UML models. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Sept. 2010. ISSN: 1063-6773.
- [106] S. Damodaran. B2B integration over the Internet with XML: RosettaNet successes and challenges. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, pages 188–195, New York, NY, USA, May 2004. Association for Computing Machinery.
- [107] J. Davies, J. Gibbons, S. Harris, and C. Crichton. The CancerGrid experience: Metadata-based model-driven engineering for clinical trials. *Science of Computer Programming*, 89:126–143, Sept. 2014.
- [108] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.
- [109] J. de Lara and E. Guerra. Deep Meta-modelling with MetaDepth. In J. Vitek, editor, *Objects, Models, Components, Patterns*, Lecture Notes in Computer Science, pages 1–20, Berlin, Heidelberg, 2010. Springer.

BIBLIOGRAPHY

- [110] J. de Lara, E. Guerra, J. Kienzle, and Y. Hattab. Facet-oriented modelling: open objects for model-driven engineering. In *SLE 2018*, pages 147–159, Boston, MA, USA, Oct. 2018. Association for Computing Machinery.
- [111] J. De Lara and H. Vangheluwe. Atom 3: A tool for multi-formalism and meta-modelling. In *International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer, 2002.
- [112] A. Demuth, M. Riedl-Ehrenleitner, R. E. Lopez-Herrejon, and A. Egyed. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software*, 111:281–297, Jan. 2016.
- [113] S. DeRose, E. Maler, D. Orchard, and N. Walsh. XML Linking Language (XLink) Version 1.1, May 2010.
- [114] M. Didonet Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: a generic model weaver. In *1 ere Journées sur l'Ingénierie Dirigée par les Modèles (IDMo5)*, pages 105–114, France, 2005.
- [115] E. W. Dijkstra. On the cruelty of really teaching computing science. *Comm. ACM*, 32(12):1398–1404, 1989.
- [116] J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *Software & Systems Modeling*, 7(4):443–467, Oct. 2008.
- [117] Z. Diskin. Towards Algebraic Graph-Based Model Theory for Computer Science. *Bulletin of Symbolic Logic*, 3:144–145, 1997.
- [118] Z. Diskin. Mathematics of UML. In *Practical Foundations of Business System Specifications*, pages 145–178. Springer Netherlands, Dordrecht, 2003.
- [119] Z. Diskin. Model Synchronization: Mappings, Tiles, and Categories. In J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE 2009*, Lecture Notes in Computer Science, pages 92–165. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [120] Z. Diskin. General Supervised Learning as Change Propagation with Delta Lenses. In *Foundations of Software Science and Computation Structures*, pages 177–197. Springer, Cham, Apr. 2020.
- [121] Z. Diskin, K. Czarnecki, and M. Antkiewicz. Model-versioning-in-the-large: Algebraic foundations and the tile notation. In *2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 7–12, May 2009. ISSN: null.
- [122] Z. Diskin, R. Eramo, A. Pierantonio, and K. Czarnecki. Incorporating Uncertainty into Bidirectional Model Transformations and their Delta-Lens Formalization. In *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016*, pages 15–31, 2016.

- [123] Z. Diskin, H. Gholizadeh, A. Wider, and K. Czarnecki. A three-dimensional taxonomy for bidirectional model synchronization. *J. Syst. Softw*, 111:298–322, Jan. 2016.
- [124] Z. Diskin, A. Gómez, and J. Cabot. Traceability Mappings as a Fundamental Instrument in Model Transformations. In M. Huisman and J. Rubin, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 247–263, Berlin, Heidelberg, 2017. Springer.
- [125] Z. Diskin, S. Kokaly, and T. Maibaum. Mapping-aware megamodeling: Design patterns and laws. *LNCS*, 8225:322–343, 2013.
- [126] Z. Diskin, H. König, and M. Lawford. Multiple model synchronization with multiary delta lenses with amendment andK-Putput. *Formal Aspects of Computing*, 31(5):611–640, Nov. 2019.
- [127] Z. Diskin, H. König, M. Lawford, and T. Maibaum. Toward Product Lines of Mathematical Models for Software Model Management. In M. Seidl and S. Zschaler, editors, *Software Technologies: Applications and Foundations*, Lecture Notes in Computer Science, pages 200–216, Cham, 2018. Springer International Publishing.
- [128] Z. Diskin, T. Maibaum, and K. Czarnecki. Intermodeling, Queries, and Kleisli Categories. In J. de Lara and A. Zisman, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 163–177, Berlin, Heidelberg, 2012. Springer.
- [129] Z. Diskin, T. Maibaum, and K. Czarnecki. Intermodeling, Queries, and Kleisli Categories. In J. de Lara and A. Zisman, editors, *Fundamental Approaches to Software Engineering*, LNCS, pages 163–177. Springer Berlin Heidelberg, 2012.
- [130] Z. Diskin, T. Maibaum, and K. Czarnecki. A Model Management Imperative: Being Graphical Is Not Sufficient, You Have to Be Categorical. In G. Taentzer and F. Bordeleau, editors, *Modelling Foundations and Applications*, Lecture Notes in Computer Science, pages 154–170, Cham, 2015. Springer International Publishing.
- [131] Z. Diskin and P. Stünkel. Sketches, Queries, and Views: From Term-Graphs to Diagrammatic Term-Sketches (Revision of TR 2020-33). Technical Report 34, McMaster Centre for Software Certification, May 2021.
- [132] Z. Diskin and U. Wolter. A diagrammatic logic for object-oriented visual modeling. In *ACCAT '07*, pages 19–41, 2007.
- [133] Z. Diskin, Y. Xiong, and K. Czarnecki. From State- to Delta-Based Bidirectional Model Transformations. In L. Tratt and M. Gogolla, editors, *Theory and Practice of Model Transformations*, Lecture Notes in Computer Science, pages 61–76. Springer Berlin Heidelberg, 2010.
- [134] Z. Diskin, Y. Xiong, and K. Czarnecki. Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In *MDI@MODELS 2010*, pages 165–179, 2011.

BIBLIOGRAPHY

- [135] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From State- to Delta-based Bidirectional Model Transformations: The Symmetric Case. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems, MODELS'11*, pages 304–318, Berlin, Heidelberg, 2011. Springer-Verlag. event-place: Wellington, New Zealand.
- [136] A. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Elsevier, June 2012.
- [137] M. Dowson. The Ariane 5 Software Failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, Mar. 1997.
- [138] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. J. Fernandes. Engineering a DSL for Software Traceability. In D. Gašević, R. Lämmel, and E. Van Wyk, editors, *Software Language Engineering*, Lecture Notes in Computer Science, pages 151–167, Berlin, Heidelberg, 2009. Springer.
- [139] T. Dyba, B. A. Kitchenham, and M. Jorgensen. Evidence-based software engineering for practitioners. *IEEE Software*, 22(1):58–65, Jan. 2005. Conference Name: IEEE Software.
- [140] C. L. Dym. *Engineering Design: A Project-Based Introduction*. Wiley, 2013.
- [141] S. Easterbrook and B. Nuseibeh. Using ViewPoints for inconsistency management. *Software Engineering Journal*, 11(1):31–43, Jan. 1996.
- [142] A. Egyed. Scalable consistency checking between diagrams - the VIEWINTEGRA approach. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 387–390, Nov. 2001. ISSN: 1938-4300.
- [143] A. Egyed, E. Letier, and A. Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 99–108, Sept. 2008.
- [144] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information Preserving Bidirectional Model Transformations. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 72–86. Springer Berlin Heidelberg, 2007.
- [145] H. Ehrig, K. Ehrig, and F. Hermann. From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars. *Electronic Communications of the EASST*, 10(0), June 2008.
- [146] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation*. Springer-Verlag Berlin Heidelberg, 1 edition, 2006.
- [147] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. From graph grammars to high level replacement systems. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, Lecture Notes in Computer Science, pages 269–291, Berlin, Heidelberg, 1991. Springer.

- [148] H. Ehrig and M. Löwe. Categorical principles, techniques and results for high-level-replacement systems in computer science. *Applied Categorical Structures*, 1(1):21–50, Mar. 1993.
- [149] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 167–180, Oct. 1973.
- [150] H. Ehrig and U. Prange. Weak Adhesive High-Level Replacement Categories and Systems: A Unifying Framework for Graph and Petri Net Transformations. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, Lecture Notes in Computer Science, pages 235–251. Springer, Berlin, Heidelberg, 2006.
- [151] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Graph Transformations*, pages 161–177, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [152] S. Eilenberg and S. MacLane. General Theory of Natural Equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.
- [153] M. Elaasar and L. Briand. An overview of UML consistency management. Technical Report SCE-04-18, Carleton University, 2004.
- [154] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 — The Unified Modeling Language*, Lecture Notes in Computer Science, pages 323–337, Berlin, Heidelberg, 2000. Springer.
- [155] G. Engels, J. M. Küster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. *ACM SIGSOFT Software Engineering Notes*, 26(5):186–195, Sept. 2001.
- [156] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. A model-driven approach to automate the propagation of changes among Architecture Description Languages. *Software & Systems Modeling*, 11(1):29–53, Feb. 2012.
- [157] R. Eramo, R. Marinelli, and A. Pierantonio. Towards a Taxonomy for Bidirectional Transformation. In *SATToSE*, 2014.
- [158] H. Erdogmus, N. Medvidović, and F. Paulisch. 50 Years of Software Engineering. *IEEE Software*, 35(5):20–24, Sept. 2018. Conference Name: IEEE Software.
- [159] EU Comission. Horizon 2020 calls for eHealth projects. <http://ec.europa.eu/digital-single-market/en/news/first-horizon-2020-calls-ehealth-projects-launched>, Last Accessed: 08.11.2017, 2015.

BIBLIOGRAPHY

- [160] E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Boston, first edition edition, Aug. 2003.
- [161] Facebook Inc. GraphQL Specification, June 2018.
- [162] J.-M. Favre. Meta-Model and Model Co-evolution within the 3D Software Space. In *International Workshop on Evolution of Large-scale Industrial Software Applications, ELISA 2003, Amsterdam*, pages 98–109, 2003.
- [163] J.-m. Favre. Towards a basic theory to model model driven engineering. In *In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004*, 2004.
- [164] J.-M. Favre. Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon¹. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
- [165] J.-M. Favre. Foundations of Model (Driven) (Reverse) Engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
- [166] J.-M. Favre. Megamodelling and Etymology. In J. R. Cordy, R. Lämmel, and A. Winter, editors, *Transformation Techniques in Software Engineering*, Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. ISSN: 1862-4405 Issue: 05161.
- [167] J.-M. Favre and T. NGuyen. Towards a Megamodel to Model Software Evolution Through Transformations. *Electronic Notes in Theoretical Computer Science*, 127(3):59–74, 2005.
- [168] S. Feldmann, K. Kernschmidt, M. Wimmer, and B. Vogel-Heuser. Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering. *Journal of Systems and Software*, 153:105–134, July 2019.
- [169] C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
- [170] P. Feyerabend. *Against Method*. Verso, 1993.
- [171] J. L. Fiadeiro. *Categories for Software Engineering*. Springer, 2005.
- [172] R. T. Fielding and R. N. Taylor. Principled design of the modern Web architecture. In *ICSE '00*, pages 407–416, Limerick, Ireland, June 2000. ACM.

- [173] A. Finkelsteiin, G. Spanoudakis, and D. Till. Managing Interference. In *ISAW '96 and Viewpoints '96 on SIGSOFT '96 Workshops*, pages 172–174, New York, NY, USA, 1996. ACM.
- [174] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In I. Sommerville and M. Paul, editors, *Software Engineering — ESEC '93*, Lecture Notes in Computer Science, pages 84–99, Berlin, Heidelberg, 1993. Springer.
- [175] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 02(01):31–57, Mar. 1992. Publisher: World Scientific Publishing Co.
- [176] A. Finkelstein, G. Spanoudakis, and D. Till. Managing Interference. In *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints '96) on SIGSOFT '96 Workshops*, ISAW '96, pages 172–174, New York, NY, USA, 1996. ACM.
- [177] B. Fong and D. I. Spivak. *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*. Cambridge University Press, 1 edition edition, Aug. 2019.
- [178] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-update Problem. *ACM Trans. Program. Lang. Syst.*, 29(3), may 2007.
- [179] N. Foster, K. Matsuda, and J. Voigtländer. Three Complementary Approaches to Bidirectional Programming. In J. Gibbons, editor, *Generic and Indexed Programming: International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, Lecture Notes in Computer Science, pages 1–46. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.
- [180] M. Fowler. *Domain-Specific Languages*. Pearson Education, Sept. 2010.
- [181] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012.
- [182] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Mar. 2012.
- [183] M. Fowler and J. Lewis. Microservices: a definition of this new architectural term. <https://martinfowler.com/articles/microservices.html>, Last Accessed: 07.02.2020, Mar. 2014.
- [184] J. R. Frade, D. Di Giacomo, S. Goedertier, N. Loutas, and V. Peristeras. Building semantic interoperability through the federation of semantic asset repositories. In *Proceedings of the 8th International Conference on Semantic Systems, I-SEMANTICS '12*, pages 185–188, New York, NY, USA, Sept. 2012. Association for Computing Machinery.

BIBLIOGRAPHY

- [185] P. D. Francesco, I. Malavolta, and P. Lago. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In *ICSA 2017*, pages 21–30, Apr. 2017.
- [186] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, pages 1433–1445, New York, NY, USA, May 2018. Association for Computing Machinery.
- [187] M. Franzago, D. D. Ruscio, I. Malavolta, and H. Muccini. Collaborative Model-Driven Software Engineering: A Classification Framework and a Research Map. *IEEE Transactions on Software Engineering*, 44(12):1146–1175, Dec. 2018. Conference Name: IEEE Transactions on Software Engineering.
- [188] G. Frege. *Begriffsschrift*. 1879.
- [189] L. Fritsche, J. Kosiol, A. Möller, A. Schürr, and G. Taentzer. A precedence-driven approach for concurrent model synchronization scenarios using triple graph grammars. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, pages 39–55. Association for Computing Machinery, New York, NY, USA, Nov. 2020.
- [190] I. Galvao and A. Goknil. Survey of Traceability Approaches in Model-Driven Engineering. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 313–313, Oct. 2007. ISSN: 1541-7719.
- [191] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [192] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The Missing Link of MDA. In A. Corradini, H. Ehrig, H. J. Kreowski, and G. Rozenberg, editors, *Graph Transformation*, Lecture Notes in Computer Science, pages 90–105, Berlin, Heidelberg, 2002. Springer.
- [193] H. Giese, S. Hildebrandt, and L. Lambers. Toward Bridging the Gap between Formal Semantics and Implementation of Triple Graph Grammars. In *and Validation 2010 Workshop on Model-Driven Engineering, Verification*, pages 19–24, Oct. 2010. ISSN: null.
- [194] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8(1):21–43, Feb. 2009.
- [195] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Transaction Publishers, 1967.
- [196] J. Gleitze. *A Declarative Language for Preserving Consistency of Multiple Models*. Bachelor Thesis, Karlsruhe Institute of Technology, 2017.

- [197] J. Gleitze, H. Klare, and E. Burger. Finding a Universal Execution Strategy for Model Transformation Networks. In E. Guerra and M. Stoelinga, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 87–107, Cham, 2021. Springer International Publishing.
- [198] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Science of Computer Programming*, 69(1):27–34, Dec. 2007.
- [199] M. Gogolla and J. Cabot. Continuing a Benchmark for UML and OCL Design and Analysis Tools. In P. Milazzo, D. Varró, and M. Wimmer, editors, *Software Technologies: Applications and Foundations*, Lecture Notes in Computer Science, pages 289–302, Cham, 2016. Springer International Publishing.
- [200] M. Gogolla, L. Hamann, F. Hilken, M. Kuhlmann, and R. France. From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In *Modellierung*, pages 273–288. Gesellschaft für Informatik e.V., Bonn, 2014. Accepted: 2019-03-19T14:06:56Z ISSN: 1617-5468.
- [201] J. Goguen. Tossing algebraic flowers down the great divide. In *In People and Ideas in Theoretical Computer Science*, pages 93–129. Springer, 1999.
- [202] J. A. Goguen. Categorical foundations for general systems theory. In F. Pichler and R. Trappl, editors, *Advances in Cybernetics and Systems Research*, pages 121–130. Transcripta Books, 1973.
- [203] J. A. Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, Mar. 1991. Publisher: Cambridge University Press.
- [204] J. A. Goguen. Sheaf Semantics for Concurrent Interacting Objects. In *Mathematical Structures in Computer Science*, pages 159–191, 1992.
- [205] J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, Jan. 1992.
- [206] U. Golas, A. Habel, and H. Ehrig. Multi-amalgamation of rules with application conditions in ω -adhesive categories. *Mathematical Structures in Computer Science*, 24(4), Aug. 2014. Publisher: Cambridge University Press.
- [207] R. Goldblatt. *Topoi: The Categorical Analysis of Logic*. Dover Publications, revised edition, Apr. 2006.
- [208] G. Goldkuhl. Design Theories in Information Systems - A Need for Multi-Grounding. *Journal of Information Technology Theory and Application (JITTA)*, 6(2), July 2004.
- [209] T. Goldschmidt, S. Becker, and E. Burger. Towards a tool-oriented taxonomy of view-based modelling. In E. Sinz and A. Schürr, editors, *Modellierung 2012*, pages 59–74. Gesellschaft für Informatik e.V., 2012. Accepted: 2018-11-14T09:41:29Z ISSN: 1617-5468.

BIBLIOGRAPHY

- [210] H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. Technical Report Part II, Volume 1-3, Institute for Advanced Study, Princeton, New Jersey, 1947.
- [211] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101, Apr. 1994.
- [212] A. Grothendieck and M. Raynaud. Revêtements étales et groupe fondamental (SGA 1). *arXiv:math/0206203*, Jan. 1971. arXiv: math/0206203.
- [213] B. Gruschko, D. Kolovos, and R. Paige. Towards Synchronizing Models with Evolving Metamodels. In *MODSE 2007*, 2007.
- [214] H. Grönniger, J. O. Ringert, and B. Rumpe. System Model-Based Definition of Modeling Language Semantics. In D. Lee, A. Lopes, and A. Poetzsch-Heffter, editors, *Formal Techniques for Distributed Systems*, Lecture Notes in Computer Science, pages 152–166, Berlin, Heidelberg, 2009. Springer.
- [215] E. Guerra, J. de Lara, D. S. Kolovos, and R. F. Paige. Inter-modelling: From Theory to Practice. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *MODELS '10*, Lecture Notes in Computer Science, pages 376–391. Springer Berlin Heidelberg, 2010.
- [216] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, and M. Roth. Clio Grows Up: From Research Prototype to Industrial Tool. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 805–810, New York, NY, USA, 2005. ACM. event-place: Baltimore, Maryland.
- [217] A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested condition†. *Mathematical Structures in Computer Science*, 19(2):245–296, Apr. 2009.
- [218] A. Habel and C. Sandmann. Graph Repair by Graph Programs. In M. Mazzara, I. Ober, and G. Salaün, editors, *Software Technologies: Applications and Foundations*, Lecture Notes in Computer Science, pages 431–446, Cham, 2018. Springer International Publishing.
- [219] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [220] V. Haren. *TOGAF Version 9.1*. Van Haren Publishing, 10th edition, 2011.
- [221] J. Haungs, M. Fowler, R. Johnson, S. McConnell, and R. Gabriel. Software Development: Arts & Crafts or Math & Science? In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 141–142, New York, NY, USA, 2004. ACM.
- [222] W. He and L. D. Xu. Integration of Distributed Enterprise Applications: A Survey. *IEEE Transactions on Industrial Informatics*, 10(1):35–42, Feb. 2014.

- [223] I. Healthcare Information and Management Systems Society. Interoperability in Healthcare, July 2020.
- [224] R. Hebig, D. E. Khelladi, and R. Bendraou. Approaches to Co-Evolution of Metamodels and Models: A Survey. *IEEE Transactions on Software Engineering*, 43(5):396–414, May 2017.
- [225] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. Closing the Gap between Modelling and Java. In M. van den Brand, D. Gašević, and J. Gray, editors, *Software Language Engineering*, Lecture Notes in Computer Science, pages 374–383, Berlin, Heidelberg, 2010. Springer.
- [226] T. Heindel. *A Category Theoretical Approach to the Concurrent Semantics of Rewriting: Adhesive Categories and Related Concepts*. PhD thesis, University of Duisburg-Essen, Apr 2010.
- [227] T. Heindel. Hereditary Pushouts Reconsidered. In H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr, editors, *Graph Transformations*, Lecture Notes in Computer Science, pages 250–265, Berlin, Heidelberg, 2010. Springer.
- [228] F. Hermann, H. Ehrig, and C. Ermel. Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks. In M. Chechik and M. Wirsing, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 325–339. Springer Berlin Heidelberg, 2009.
- [229] F. Hermann, H. Ehrig, C. Ermel, and F. Orejas. Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars. In J. de Lara and A. Zisman, editors, *FASE 2012*, Lecture Notes in Computer Science, pages 178–193. Springer Berlin Heidelberg, 2012.
- [230] F. Hermann, H. Ehrig, F. Orejas, K. Czarnecki, Z. Diskin, and Y. Xiong. Correctness of Model Synchronization Based on Triple Graph Grammar. In J. Whittle, T. Clark, and T. Kühne, editors, *MODELS 2011*, pages 668–682, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [231] A. Hevner. A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems*, 19(2), Jan. 2007.
- [232] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–105, 2004. Publisher: Management Information Systems Research Center, University of Minnesota.
- [233] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 480–483, Nov. 2011.
- [234] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. Feature-based Classification of Bidirectional Transformation Approaches. *Softw. Syst. Model.*, 15(3):907–928, jul 2016.

BIBLIOGRAPHY

- [235] G. Hinkel and E. Burger. Change Propagation and Bidirectionality in Internal Transformation DSLs. *Softw. Syst. Model.*, 18(1):249–278, Feb. 2019.
- [236] HL7.org. Graphql - FHIR v4.0.1.
- [237] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.
- [238] M. Hofmann, B. Pierce, and D. Wagner. Symmetric Lenses. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 371–384, New York, NY, USA, 2011. ACM. event-place: Austin, Texas, USA.
- [239] M. Hofmann, B. Pierce, and D. Wagner. Edit Lenses. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 495–508, New York, NY, USA, 2012. ACM.
- [240] G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012.
- [241] Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger. Dagstuhl seminar on bidirectional transformations (BX). *ACM SIGMOD Record*, 40(1):35, 2011.
- [242] D. Hume. *A Treatise of Human Nature*. 1740.
- [243] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.
- [244] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, 2 edition, 2004.
- [245] Z. Huzar, L. Kuzniarz, G. Reggio, and J. L. Sourrouille. Consistency Problems in UML-based Software Development. In *Proceedings of the 2004 International Conference on UML Modeling Languages and Applications*, UML'04, pages 1–12, Berlin, Heidelberg, 2005. Springer-Verlag.
- [246] ISO/IEC JTC 1/SC 7 Software and systems engineering. Iso/iec/ieee 42010:2011 - systems and software engineering — architecture description. <https://www.iso.org/standard/50508.html>, Dec. 2011.
- [247] ITSMF UK. *ITIL Foundation Handbook*. The Stationery Office, GBR, 3rd edition, 2012.
- [248] J. Ivari. A Paradigmatic Analysis of Information Systems As a Design Science. *Scandinavian Journal of Information Systems*, 19(2), Jan. 2007.
- [249] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2016.

- [250] I. Jacobson, H. B. Lawson, P.-W. Ng, P. E. McMahon, and M. Goedicke. *The Essentials of Modern Software Engineering: Free the Practices from the Method Prisons!* Association for Computing Machinery and Morgan & Claypool, 2019.
- [251] K. Jensen and L. M. Kristensen. Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems. *Communications of the ACM*, 58(6):61–70, May 2015.
- [252] M. Johnson and R. Rosebrugh. Sketch Data Models, Relational Schema and Data Specifications. *Electronic Notes in Theoretical Computer Science*, 61:51–63, Jan. 2002.
- [253] M. Johnson and R. Rosebrugh. Fibrations and universal view updatability. *Theoretical Computer Science*, 388(1):109–129, Dec. 2007.
- [254] M. Johnson and R. Rosebrugh. Implementing a Categorical Information System. In J. Meseguer and G. Roşu, editors, *Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, pages 232–237, Berlin, Heidelberg, 2008. Springer.
- [255] M. Johnson and R. Rosebrugh. Symmetric delta lenses and spans of asymmetric delta lenses. *The Journal of Object Technology*, 16(1):2:1, 2017.
- [256] M. Johnson, R. Rosebrugh, and R. Wood. Algebras and Update Strategies. *j-jucs*, 16(5):729–748, Mar. 2010.
- [257] M. Johnson, R. Rosebrugh, and R. J. Wood. Entity-relationship-attribute designs and sketches. *Theory and Applications of Categories*, 10(1):94–112, 2002.
- [258] M. Johnson and R. D. Rosebrugh. Unifying Set-Based, Delta-Based and Edit-Based Lenses. In A. Anjorin and J. Gibbons, editors, *Bx@ETAPS 2016*, volume 1571 of *CEUR Workshop Proceedings*, pages 1–13. CEUR-WS.org, 2016.
- [259] M. Johnson and P. Stevens. Confidentiality in the process of (model-driven) software development. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, Programming’18 Companion, pages 1–8, Nice, France, Apr. 2018. Association for Computing Machinery.
- [260] M. Jørgensen and D. Sjøberg. Generalization and theory-building in software engineering research. *Empirical Assessment in Software Eng. Proc*, pages 29–36, 2004.
- [261] A. Joshi and M. P. E. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In R. Winther, B. A. Gran, and G. Dahll, editors, *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 122–135, Berlin, Heidelberg, 2005. Springer.
- [262] N. M. Josuttis. *SOA in Practice*. "O’Reilly Media, Inc.", 2007.
- [263] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA ’06*, page 719, New York, New York, USA, 2006. ACM Press.

BIBLIOGRAPHY

- [264] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró. Survey and classification of model transformation tools. *Software & Systems Modeling*, Mar. 2018.
- [265] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
- [266] I. Kant. *Kritik der Reinen Vernunft*. 1787.
- [267] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach. Understanding model evolution through semantically lifting model differences with SiLift. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 638–641, Sept. 2012. ISSN: 1063-6773.
- [268] T. Kehrer, U. Kelter, and G. Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 163–172, Nov. 2011. ISSN: 1938-4300.
- [269] T. Kehrer, U. Kelter, and G. Taentzer. Consistency-preserving edit scripts in model versioning. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 191–201, Nov. 2013.
- [270] T. Kehrer, C. Pietsch, U. Kelter, D. Strüber, and S. Vaupel. An adaptable tool environment for high-level differencing of textual models. In *OCL'15: International Workshop on OCL and Textual Modeling*, pages 62–72. CEUR-WS.org, 2015.
- [271] T. Kehrer, G. Taentzer, M. Rindt, and U. Kelter. Automatically Deriving the Specification of Model Editing Operations from Meta-Models. In P. Van Gorp and G. Engels, editors, *Theory and Practice of Model Transformations*, Lecture Notes in Computer Science, pages 173–188, Cham, 2016. Springer International Publishing.
- [272] R. Kennaway. Graph rewriting in some categories of partial morphisms. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Grammars and Their Application to Computer Science*, Lecture Notes in Computer Science, pages 490–504. Springer Berlin Heidelberg, 1991.
- [273] S. Kent. Model Driven Engineering. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods*, Lecture Notes in Computer Science, pages 286–298, Berlin, Heidelberg, 2002. Springer.
- [274] W. Kessentini, H. Sahraoui, and M. Wimmer. Automated metamodel/model co-evolution: A search-based approach. *Information and Software Technology*, 106:49–67, Feb. 2019.
- [275] J. Kienzle, G. Mussbacher, B. Combemale, and J. Deantoni. A unifying framework for homogeneous model composition. *Software & Systems Modeling*, 18(5):3005–3023, Oct. 2019.

- [276] B. Kitchenham, S. Charters, D. Budgen, P. Brereton, M. Turner, S. Linkman, M. Jørgensen, E. Mendes, and G. Visaggio. Guidelines for performing Systematic Literature Reviews in Software Engineering. techreport 1, Evidence Based Software Engineering, Software Engineering Group School of Computer Science and Mathematics Keele University Keele, Staffs ST5 5BG, UK, jul 2007.
- [277] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, Aug. 2002.
- [278] H. Klare. Multi-model Consistency Preservation. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS '18*, pages 156–161, New York, NY, USA, 2018. ACM. event-place: Copenhagen, Denmark.
- [279] H. Klare and J. Gleitze. Commonalities for Preserving Consistency of Multiple Models. In *MODELS 2019 Companion*, pages 371–378, Sept. 2019.
- [280] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner. Enabling consistency in view-based system development — The Vitruvius approach. *Journal of Systems and Software*, 171:110815, Jan. 2021.
- [281] M. Kleiner, M. D. Del Fabro, and P. Albert. Model Search: Formalizing and Automating Constraint Solving in MDE Platforms. In T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, editors, *ECMFA 2010*, Lecture Notes in Computer Science, pages 173–188. Springer Berlin Heidelberg, 2010.
- [282] M. Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. "O'Reilly Media, Inc.", Mar. 2017. Google-Books-ID: p1heDgAAQBAJ.
- [283] W. Kling, F. Jouault, D. Wagelaar, M. Brambilla, and J. Cabot. MoScript: A DSL for Querying and Manipulating Model Repositories. In A. Sloane and U. Aßmann, editors, *Software Language Engineering*, Lecture Notes in Computer Science, pages 180–200, Berlin, Heidelberg, 2012. Springer.
- [284] A. Knapp and T. Mossakowski. Multi-view Consistency in UML: A Survey. In *Graph Transformation, Specifications, and Nets*, LNCS 10800, pages 37–60. Springer, Cham, 2018.
- [285] K. Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, Mar. 1989.
- [286] H.-S. Ko, T. Zan, and Z. Hu. BiGUL: A Formally Verified Core Language for Putback-based Bidirectional Programming. In *PEPM '16*, pages 61–72. ACM, 2016.
- [287] M. Koegel, M. Herrmannsdoerfer, Y. Li, J. Helming, and J. David. Comparing State- and Operation-Based Change Tracking on Models. In *2010 14th IEEE*

BIBLIOGRAPHY

- International Enterprise Distributed Object Computing Conference*, pages 163–172, Oct. 2010. ISSN: 1541-7719.
- [288] D. Kolovos, R. Paige, and F. Polack. Detecting and Repairing Inconsistencies Across Heterogeneous Models. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, ICST '08*, pages 356–364, Washington, DC, USA, 2008. IEEE Computer Society.
- [289] D. S. Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In R. F. Paige, A. Hartman, and A. Rensink, editors, *Model Driven Architecture - Foundations and Applications*, Lecture Notes in Computer Science, pages 146–157, Berlin, Heidelberg, 2009. Springer.
- [290] D. S. Kolovos, D. Di Ruscio, A. Pierantonio, and R. F. Paige. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, CVSM '09*, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society.
- [291] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. The Epsilon Object Language (EOL). In A. Rensink and J. Warmer, editors, *Model Driven Architecture – Foundations and Applications*, Lecture Notes in Computer Science, pages 128–142, Berlin, Heidelberg, 2006. Springer.
- [292] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Merging Models with the Epsilon Merging Language (EML). In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 215–229, Berlin, Heidelberg, 2006. Springer.
- [293] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In J.-R. Abrial and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis: Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, pages 204–218. Springer, Berlin, Heidelberg, 2009.
- [294] H. König and Z. Diskin. Advanced Local Checking of Global Consistency in Heterogeneous Multimodeling. In *ECMFA 2016*, pages 19–35, 2016.
- [295] H. König and Z. Diskin. Efficient Consistency Checking of Interrelated Models. In *ECMFA 2017*, pages 161–178, 2017.
- [296] J. Kosiol, L. Fritsche, A. Schürr, and G. Taentzer. Adhesive Subcategories of Functor Categories with Instantiation to Partial Triple Graphs. In E. Guerra and F. Orejas, editors, *Graph Transformation*, Lecture Notes in Computer Science, pages 38–54. Springer International Publishing, 2019.
- [297] J. Kosiol and H. Radke. Rule-based repair of emf models : Formalization and correctness proof. In *GCM 2017*, 2017.

- [298] R. Krömer. *Tool and Object: A History and Philosophy of Category Theory*. Science Networks. Historical Studies. Birkhäuser Basel, 2007.
- [299] B. Kuechler and V. Vaishnavi. On theory development in design science research: anatomy of a research project. *European Journal of Information Systems*, 17(5):489–504, Oct. 2008.
- [300] T. S. Kuhn. *The structure of scientific revolutions*. University of Chicago Press, Chicago, 1970.
- [301] I. Kurtev, J. Bézivin, and M. Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002.
- [302] H. König and P. Stünkel. Single Pushout Rewriting in Comprehensive Systems. In F. Gadducci and T. Kehrer, editors, *Graph Transformation*, Lecture Notes in Computer Science, pages 91–108, Cham, 2020. Springer International Publishing.
- [303] T. Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4):369–385, Dec. 2006.
- [304] S. Lack and P. Sobociński. Adhesive categories. In I. Walukiewicz, editor, *Foundations of Software Science and Computation Structures*, pages 273–288, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [305] S. Lack and P. Sobociński. Toposes Are Adhesive. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations*, Lecture Notes in Computer Science, pages 184–198. Springer Berlin Heidelberg, 2006.
- [306] J. Lambek. From λ -calculus to cartesian closed categories. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, pages 375–402, 1980.
- [307] R. Lämmel. Coupled software transformations (Extended Abstract). In *Proceedings 1st International Workshop on Software Evolution Transformations*, pages 31–35, 2014.
- [308] J.-L. Lauriere. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, Feb. 1978.
- [309] F. W. Lawvere. *Functorial Semantics of Algebraic Theories*. PhD Thesis, Columbia University, 1963.
- [310] F. W. Lawvere and S. H. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. Cambridge University Press, Cambridge, 2 edition, 2009.
- [311] E. Leblebici, A. Anjorin, L. Fritsche, G. Varró, and A. Schürr. Leveraging Incremental Pattern Matching Techniques for Model Synchronisation. In J. de Lara and D. Plump, editors, *Graph Transformation*, Lecture Notes in Computer Science, pages 179–195. Springer International Publishing, 2017.

BIBLIOGRAPHY

- [312] E. Leblebici, A. Anjorin, and A. Schürr. Inter-model Consistency Checking Using Triple Graph Grammars and Linear Optimization Techniques. In *Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering - Volume 10202*, pages 191–207, New York, NY, USA, 2017. Springer-Verlag New York, Inc.
- [313] E. Leblebici, A. Anjorin, and A. Schürr. Developing eMoflon with eMoflon. In D. Di Ruscio and D. Varró, editors, *Theory and Practice of Model Transformations*, Lecture Notes in Computer Science, pages 138–145. Springer International Publishing, 2014.
- [314] S. K. Lellahi and N. Spyrtos. Towards a categorical data model supporting structured objects and inheritance. In J. W. Schmidt and A. A. Stogny, editors, *Next Generation Information System Technology*, Lecture Notes in Computer Science, pages 86–105, Berlin, Heidelberg, 1991. Springer.
- [315] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.
- [316] M. Lindvall and K. Sandahl. Practical Implications of Traceability. *Software: Practice and Experience*, 26(10):1161–1180, 1996.
- [317] P. Lopes and J. L. Oliveira. A semantic web application framework for health systems interoperability. In *Proceedings of the first international workshop on Managing interoperability and complexity in health systems*, MIXHS '11, pages 87–90, New York, NY, USA, Oct. 2011. Association for Computing Machinery.
- [318] M. Löwe. *Extended algebraic graph transformation*. PhD thesis, Technical University of Berlin, Germany, 1991.
- [319] M. Löwe and M. Tempelmeier. Single-pushout rewriting of partial algebras. In D. Plump, editor, *Proceedings of GCM co-located with ICGT / STAF, L'Aquila, Italy*, volume 1403 of *CEUR Workshop Proceedings*, pages 82–96, 2015.
- [320] F. J. Lucas, F. Molina, and A. Toval. A systematic review of UML model consistency management. *Information and Software Technology*, 51(12):1631–1645, Dec. 2009.
- [321] R. Lämmel. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer International Publishing, 2018.
- [322] M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1):181–224, Mar. 1993.
- [323] M. Löwe. Graph Rewriting in Span-Categories. In H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr, editors, *Graph Transformations*, Lecture Notes in Computer Science, pages 218–233, Berlin, Heidelberg, 2010. Springer.
- [324] M. Löwe. Refined Graph Rewriting in Span-Categories. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Graph Transformations*, Lecture Notes in Computer Science, pages 111–125, Berlin, Heidelberg, 2012. Springer.

- [325] M. Löwe, H. König, C. Schulz, and M. Schultchen. Algebraic graph transformations with inheritance and abstraction. *Science of Computer Programming*, 107-108:2–18, Sept. 2015.
- [326] S. Mac Lane. *Categories for the Working Mathematician*. Springer, 1998.
- [327] N. Macedo and A. Cunha. Implementing QVT-R Bidirectional Model Transformations Using Alloy. In V. Cortellessa and D. Varró, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 297–311. Springer Berlin Heidelberg, 2013.
- [328] N. Macedo and A. Cunha. Least-change bidirectional model transformation with QVT-R and ATL. *Software & Systems Modeling*, 15(3):783–810, July 2016.
- [329] N. Macedo, A. Cunha, and H. Pacheco. Towards a framework for multidirectional model transformations. In *EDBT/ICDT 2014*, pages 71–74, 2014.
- [330] N. Macedo, T. Guimarães, and A. Cunha. Model repair and transformation with Echo. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 694–697, Nov. 2013. ISSN: null.
- [331] N. Macedo, T. Jorge, and A. Cunha. A Feature-Based Classification of Model Repair Approaches. *IEEE Transactions on Software Engineering*, 43(7):615–640, July 2017.
- [332] M. Makkai. Generalized sketches as a framework for completeness theorems. *Journal of Pure and Applied Algebra*, 115:49–79, 179–212, 214–274, Feb. 1997.
- [333] R. Malhotra. *Empirical Research in Software Engineering: Concepts, Analysis, and Applications*. CRC Press, Mar. 2016.
- [334] F. Mantz, G. Taentzer, Y. Lamo, and U. Wolter. Co-evolving meta-models and their instance models: A formal approach based on graph transformation. *Science of Computer Programming*, 104(1):2–43, 2015.
- [335] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251–266, Dec. 1995.
- [336] M. L. Markus, A. Majchrzak, and L. Gasser. A Design Theory for Systems That Support Emergent Knowledge Processes. *MIS Quarterly*, 26(3):179–212, 2002. Publisher: Management Information Systems Research Center, University of Minnesota.
- [337] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In J. Cardoso and A. Sheth, editors, *Semantic Web Services and Web Process Composition*, Lecture Notes in Computer Science, pages 26–42, Berlin, Heidelberg, 2005. Springer.
- [338] S. Martínez, S. Gérard, and J. Cabot. Efficient model similarity estimation with robust hashing. *Software and Systems Modeling*, Aug. 2021.

BIBLIOGRAPHY

- [339] L. Meertens. Designing Constraint Maintainers for User Interaction. Technical report, CWI, Amsterdam, 1998.
- [340] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A Programming Platform for Generic Model Management. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03*, pages 193–204, New York, NY, USA, 2003. ACM. event-place: San Diego, California.
- [341] T. Mens and R. Van Der Straeten. Incremental Resolution of Model Inconsistencies. In J. L. Fiadeiro and P.-Y. Schobbens, editors, *Recent Trends in Algebraic Development Techniques*, Lecture Notes in Computer Science, pages 111–126. Springer Berlin Heidelberg, 2007.
- [342] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, Mar. 2006.
- [343] Meteor Development Group Inc. Apollo federation overview. <https://www.apollographql.com/docs/federation/>, Last Accessed: 07.02.2020, 2021.
- [344] B. Milewski. *Category Theory for Programmers*. Blurb, Incorporated, Aug. 2019.
- [345] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.
- [346] D. Monniaux. A Survey of Satisfiability Modulo Theory. In V. P. Gerdt, W. Koepf, W. M. Seiler, and E. V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing*, Lecture Notes in Computer Science, pages 401–425, Cham, 2016. Springer International Publishing.
- [347] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set, Hets. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 519–522. Springer Berlin Heidelberg, 2007.
- [348] T. Mossakowski and A. Tarlecki. Heterogeneous Logical Environments for Distributed Specifications. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques*, Lecture Notes in Computer Science, pages 266–289, Berlin, Heidelberg, 2009. Springer.
- [349] S. K. Mukhiya, F. Rabbi, V. Ka I Pun, A. Rutle, and Y. Lamo. A GraphQL approach to Healthcare Information Exchange with HL7 FHIR. *Procedia Computer Science*, 160:338–345, Jan. 2019.
- [350] F. u. Muram, H. Tran, and U. Zdun. Systematic Review of Software Behavioral Model Consistency Checking. *ACM Computing Surveys*, 50(2):17:1–17:39, Apr. 2017.
- [351] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. H. C. Cheng, P. Collet, B. Combemale, R. B. France, R. Heldal, J. Hill, J. Kienzle, M. Schöttle, F. Steimann, D. Stikkolorum, and J. Whittle. The Relevance of Model-Driven Engineering

- Thirty Years from Now. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 183–200, Cham, 2014. Springer International Publishing.
- [352] N. Mustafa and Y. Labiche. The Need for Traceability in Heterogeneous Systems: A Systematic Literature Review. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 305–310, July 2017. ISSN: 0730-3157.
- [353] N. Nassar, J. Kosiol, T. Arendt, and G. Taentzer. OCL2AC: Automatic Translation of OCL Constraints to Graph Constraints and Application Conditions for Transformation Rules. In L. Lambers and J. Weber, editors, *Graph Transformation*, Lecture Notes in Computer Science, pages 171–177, Cham, 2018. Springer International Publishing.
- [354] N. Nassar, H. Radke, and T. Arendt. Rule-Based Repair of EMF Models: An Automated Interactive Approach. In E. Guerra and M. van den Brand, editors, *Theory and Practice of Model Transformation*, Lecture Notes in Computer Science, pages 171–181. Springer International Publishing, 2017.
- [355] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, Aug. 1973.
- [356] P. Naur and B. Randell. Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968. Technical report, Scientific Affairs Division, NATO, Brussels, 1969.
- [357] C. Nentwich, W. Emmerich, and A. Finkelsteiin. Consistency Management with Repair Actions. In *ICSE '03*, pages 455–464, 2003.
- [358] P. G. Neumann. *Computer Related Risks*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.
- [359] I. Newton. *Philosophiæ Naturalis Principia Mathematica*. 1687.
- [360] J. F. Nunamaker and M. Chen. Systems development in information systems research. In *Twenty-Third Annual Hawaii International Conference on System Sciences*, volume 3, pages 631–640 vol.3, Jan. 1990.
- [361] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *Computer*, 33(4):24–29, Apr. 2000.
- [362] B. Nuseibeh, J. Kramer, and A. Finkelsteiin. Expressing the relationships between multiple views in requirements specification. In *Proceedings of the 15th international conference on Software Engineering, ICSE '93*, pages 187–196, Washington, DC, USA, May 1993. IEEE Computer Society Press.
- [363] Object Management Group. Object Constraint Language (OCL) v.2.3.1, 2012.

BIBLIOGRAPHY

- [364] Object Management Group. Business Process Model And Notation (BPMN) v.2.0.2, 2014.
- [365] Object Management Group. Unified Modeling Language (UML) v.2.4.1, 2015.
- [366] Object Management Group. XML Metadata Interchange (XMI) v.2.5.1, 2015.
- [367] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) v.1.3. <http://www.omg.org/spec/QVT/1.3>, 2016.
- [368] Object Management Group. Meta Object Facility (MOF) Core Specification v. 2.4.1, 2016.
- [369] Object Management Group. Decision Model and Notation (DMN) v.1.2, 2019.
- [370] F. Orejas, E. Pino, and M. Navarro. Incremental Concurrent Model Synchronization using Triple Graph Grammars. In H. Wehrheim and J. Cabot, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 273–293, Cham, 2020. Springer International Publishing.
- [371] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Drivalos, and F. A. C. Polack. The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In *Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS '09*, pages 162–171, Washington, DC, USA, 2009. IEEE Computer Society.
- [372] R. F. Paige, D. S. Kolovos, L. M. Rose, N. Matragkas, and J. R. Williams. Model Management in the Wild. In R. Lämmel, J. Saraiva, and J. Visser, editors, *GTTSE 2011*, LNCS, pages 197–218. Springer, Berlin, Heidelberg, 2013.
- [373] R. F. Paige, N. Matragkas, and L. M. Rose. Evolving models in Model-Driven Engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272–280, Jan. 2016.
- [374] P. Y. Papalambros. Design Science: Why, What and How. *Design Science*, 1, July 2015. Publisher: Cambridge University Press.
- [375] D. L. Parnas. Software Engineering Principles. *INFOR: Information Systems and Operational Research*, 22(4):303–316, Nov. 1984.
- [376] T. Parr. *The Definitive ANTLR Reference: Building Domain-specific Languages*. Pragmatic Bookshelf, 2007.
- [377] J. Pearl. Heuristic Search Theory: Survey of Recent Results. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI'81*, pages 554–562, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.
- [378] K.-H. Pennemann. An Algorithm for Approximating the Satisfiability Problem of High-level Conditions. *Electronic Notes in Theoretical Computer Science*, 213(1):75–94, May 2008.

- [379] K. Petersen, S. Vakkalanka, and L. Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, Aug. 2015.
- [380] B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, MA, USA, 1991.
- [381] F. Piessens and E. Steegmans. Categorical data-specifications. *Theory and Applications of Categories*, 1:156–173, 1995.
- [382] S. F. Pileggi and C. Fernandez-Llatas. *Semantic Interoperability: Issues, Solutions, Challenges*. River Publishers, Mar. 2012.
- [383] J. Pinna Puissant. *Resolving Inconsistencies in Model-Driven Engineering using Automated Planning*. PhD Thesis, University of Mons, Sept. 2012.
- [384] J. Pinna Puissant, R. Van Der Straeten, and T. Mens. Badger: A Regression Planner to Resolve Design Model Inconsistencies. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. Kolovos, editors, *Modelling Foundations and Applications*, Lecture Notes in Computer Science, pages 146–161, Berlin, Heidelberg, 2012. Springer.
- [385] J. Pinna Puissant, R. Van Der Straeten, and T. Mens. Resolving model inconsistencies using automated regression planning. *Software & Systems Modeling*, 14(1):461–481, Feb. 2015.
- [386] D. Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In R. Sleep, R. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting*, pages 201–213. John Wiley, 1993.
- [387] S. Pokarev, M. Reichert, M. W. A. Stehen, and R. Wieringa. Semantic and Pragmatic Interoperability: A Model for Understanding. In *Proceedings of the Open Interop Workshop on Enterprise Modelling and Ontologies for Interoperability*, volume 160. CEUR-WS.org, 2005.
- [388] J. D. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.
- [389] K. Popper. *Logik der Forschung*. 1935.
- [390] R. A. Pottinger and P. A. Bernstein. Merging Models Based on Given Correspondences. In J.-C. Freytag, P. Lockemann, S. Abiteboul, M. Carey, P. Selinger, and A. Heuer, editors, *Proceedings 2003 VLDB Conference*, pages 862–873. Morgan Kaufmann, San Francisco, Jan. 2003.
- [391] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, Apr. 2005.
- [392] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *J. Comput. Syst. Sci.*, 5(6):560–595, Dec. 1971.

BIBLIOGRAPHY

- [393] F. Rabbi, Y. Lamo, and W. MacCaull. Co-ordination of multiple metamodels, with application to healthcare systems. *Procedia Computer Science*, 37(1877):473–480, 2014.
- [394] J. C. Raoult. On graph rewritings. *Theoretical Computer Science*, 32(1):1–24, Jan. 1984.
- [395] A. Reder and A. Egyed. Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML. In *Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE '10*, pages 347–348, Antwerp, Belgium, Sept. 2010. Association for Computing Machinery.
- [396] A. Reder and A. Egyed. Computing repair trees for resolving inconsistencies in design models. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 220–229, Sept. 2012.
- [397] A. Rensink. Representing First-Order Logic Using Graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Graph Transformations*, Lecture Notes in Computer Science, pages 319–335. Springer Berlin Heidelberg, 2004.
- [398] A. Rensink and A. Kleppe. On a Graph-Based Semantics for UML Class and Object Diagrams. *Electronic Communications of the EASST*, 10(o), July 2008.
- [399] J. E. Rivera and A. Vallecillo. Representing and Operating with Model Differences. In R. F. Paige and B. Meyer, editors, *Objects, Components, Models and Patterns*, Lecture Notes in Business Information Processing, pages 141–160. Springer Berlin Heidelberg, 2008.
- [400] E. Robinson and G. Rosolini. Categories of partial maps. *Information and Computation*, 79(2):95–130, Nov. 1988.
- [401] J. D. Rocco, D. D. Ruscio, H. Narayanankutty, and A. Pierantonio. Resilience in sirius editors: Understanding the impact of metamodel changes. In *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, Hu-FaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018*, pages 620–630, 2018.
- [402] J. F. Roddick. Schema Evolution in Database Systems: An Annotated Bibliography. *SIGMOD Rec.*, 21(4):35–40, Dec. 1992.
- [403] G. Rozenberg. *Handbook of graph grammars and computing by graph transformation, Volume 1*. World Scientific, 1997.
- [404] J. Rubin and M. Chechik. N-way Model Merging. In *ESEC/FSE 2013*, pages 301–311, New York, NY, USA, 2013. ACM.
- [405] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 2nd edition, 2004.

- [406] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.
- [407] A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, University of Bergen, 2010.
- [408] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A Diagrammatic Formalisation of MOF-Based Modelling Languages. In *TOOLS EUROPE 2009*, pages 37–56. Springer, Berlin, Heidelberg, 2009.
- [409] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A formal approach to the specification and transformation of constraints in MDE. *JLAMP*, 81(4):422–457, 2012.
- [410] M. Sabetzadeh and S. Easterbrook. Analysis of inconsistency in graph-based viewpoints: a category-theoretical approach. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 12–21, Oct. 2003. ISSN: 1938-4300.
- [411] M. Sabetzadeh and S. Easterbrook. An Algebraic Framework for Merging Incomplete and Inconsistent Views. In *RE 2005*, pages 306–315, 2005.
- [412] M. Sabetzadeh, S. Nejati, S. Liaskos, S. Easterbrook, and M. Chechik. Consistency Checking of Conceptual Models via Model Merging. In *RE 2007*, pages 221–230, oct 2007.
- [413] L. Samimi-Dehkordi, B. Zamani, and S. Kolahdouz-Rahimi. EVL+Strace: a novel bidirectional model transformation approach. *Information and Software Technology*, 100:47–72, Aug. 2018.
- [414] J. E. Sammet. The real creators of Cobol. *IEEE Software*, 17(2):30–32, Mar. 2000. Conference Name: IEEE Software.
- [415] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin Heidelberg, 2012.
- [416] I. Santiago, A. Jiménez, J. M. Vara, V. De Castro, V. A. Bollati, and E. Marcos. Model-Driven Engineering as a new landscape for traceability management: A systematic literature review. *Information and Software Technology*, 54(12):1340–1356, Dec. 2012.
- [417] H. Schichl. Models and the History of Modeling. In J. Kallrath, editor, *Modeling Languages in Mathematical Optimization*, Applied Optimization, pages 25–36. Springer US, Boston, MA, 2004.
- [418] S. Schneider, L. Lambers, and F. Orejas. Automated reasoning for attributed graph properties. *International Journal on Software Tools for Technology Transfer*, 20(6):705–737, Nov. 2018.

BIBLIOGRAPHY

- [419] S. Schneider, L. Lambers, and F. Orejas. A Logic-Based Incremental Approach to Graph Repair. In R. Hähnle and W. van der Aalst, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 151–167. Springer International Publishing, 2019.
- [420] P. Schultz, D. I. Spivak, C. Vasilakopoulou, and R. Wisnesky. Algebraic databases, 2016.
- [421] P. Schultz and R. Wisnesky. Algebraic data integration*. *Journal of Functional Programming*, 27, 2017.
- [422] A. Schürr. Specification of graph translators with triple graph grammars. In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, pages 151–163, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [423] K. Schwab. *The Fourth Industrial Revolution*. Penguin UK, Jan. 2017.
- [424] J. C. Segen. *The Dictionary of Modern Medicine*. CRC Press, Feb. 1992.
- [425] P. Selinger. A Survey of Graphical Languages for Monoidal Categories. In B. Coecke, editor, *New Structures for Physics*, Lecture Notes in Physics, pages 289–355. Springer, Berlin, Heidelberg, 2011.
- [426] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Softw.*, 20(5):42–45, sep 2003.
- [427] A. P. Sheth. Changing Focus on Interoperability in Information Systems: From System, Syntax, Structure to Semantics. In M. Goodchild, M. Egenhofer, R. Fegeas, and C. Kottman, editors, *Interoperating Geographic Information Systems*, The Springer International Series in Engineering and Computer Science, pages 5–29. Springer US, Boston, MA, 1999.
- [428] M. A. A. d. Silva, A. Mougnot, X. Blanc, and R. Bendraou. Towards Automated Inconsistency Handling in Design Models. In *Advanced Information Systems Engineering*, Lecture Notes in Computer Science, pages 348–362. Springer, Berlin, Heidelberg, June 2010.
- [429] H. A. Simon. *The Sciences of the Artificial*. MIT Press, third edition, Jan. 1969.
- [430] I. Solheim and K. Stølen. Technology Research Explained. Technical Report A313, SINTEF, Mar. 2007.
- [431] I. Sommerville. *Software Engineering*. Pearson/Addison-Wesley, 2004.
- [432] G. Spanoudakis, A. Finkelstein, and D. Till. Overlaps in Requirements Engineering. *Automated Software Engineering*, 6(2):171–198, Apr. 1999.
- [433] G. Spanoudakis and A. Zisman. Inconsistency Management in Software Engineering: Survey and Open Research Issues. In *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380, 2001.

- [434] G. Spanoudakis and A. Zisman. Software traceability: a roadmap. In *Handbook of Software Engineering and Knowledge Engineering*, pages 395–428. WORLD SCIENTIFIC, Aug. 2005.
- [435] S. Staab, T. Walter, G. Gröner, and F. S. Parreiras. Model Driven Engineering with Ontology Technologies. In U. Aßmann, A. Bartho, and C. Wende, editors, *Reasoning Web. Semantic Technologies for Software Engineering: 6th International Summer School 2010, Dresden, Germany, August 30 - September 3, 2010. Tutorial Lectures*, Lecture Notes in Computer Science, pages 62–98. Springer, Berlin, Heidelberg, 2010.
- [436] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, Dec. 2008.
- [437] A. G. Stephenson, D. R. Mulville, F. H. Bauer, G. A. Dukeman, P. Norvig, L. S. LaPiana, and R. Sackheim. Mars climate orbiter mishap investigation board phase i report. Technical report, NASA, Washington D.C., 1999.
- [438] P. Stevens. A Landscape of Bidirectional Model Transformations. In *Generative and Transformational Techniques in Software Engineering II*, Lecture Notes in Computer Science, pages 408–424. Springer, Berlin, Heidelberg, July 2007.
- [439] P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software & Systems Modeling*, 9(1):7, dec 2008.
- [440] P. Stevens. Bidirectional transformations in the large. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 1–11, 2017.
- [441] P. Stevens. Towards Sound, Optimal, and Flexible Building from Megamodels. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS '18*, pages 301–311, New York, NY, USA, 2018. ACM.
- [442] P. Stevens. Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels. *Software and Systems Modeling*, Mar. 2020.
- [443] R. V. D. Straeten, J. P. Puissant, and T. Mens. Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In *ECMFA*, 2011.
- [444] R. Studer, V. R. Benjamins, and D. Fensel. Knowledge engineering: Principles and methods. *Data & Knowledge Engineering*, 25(1):161–197, Mar. 1998.
- [445] P. Stünkel, H. König, Y. Lamo, and A. Rutle. Multimodel correspondence through inter-model constraints. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, Programming'18 Companion*, page 9–17, New York, NY, USA, 2018. Association for Computing Machinery.

BIBLIOGRAPHY

- [446] P. Stünkel, H. König, Y. Lamo, and A. Rutle. Towards multiple model synchronization with comprehensive systems. In H. Wehrheim and J. Cabot, editors, *Fundamental Approaches to Software Engineering*, volume 12076 of *Lecture Notes in Computer Science*, pages 335–356. Springer, Cham, Apr. 2020.
- [447] P. Stünkel, O. v. Bargaen, A. Rutle, and Y. Lamo. GraphQL Federation: A Model-Based Approach. *Journal of Object Technology*, 19(2):18:1–21, July 2020.
- [448] P. Stünkel and H. König. Single pushout rewriting in comprehensive systems of graph-like structures. *Theoretical Computer Science*, 884:23–43, 2021.
- [449] P. Stünkel and H. König. Single pushout rewriting in comprehensive systems of graph-like structures. *Theoretical Computer Science*, July 2021.
- [450] P. Stünkel, H. König, Y. Lamo, and A. Rutle. Comprehensive Systems: A formal foundation for Multi-Model Consistency Management. *Formal Aspects of Computing*, July 2021.
- [451] P. Stünkel, H. König, A. Rutle, and Y. Lamo. Multi-Model Evolution through Model Repair. *Journal of Object Technology*, 20(1):1:1–25, Jan. 2021.
- [452] G. Taentzer, K. Ehrig, E. Guerra, J. D. Lara, T. Levendovszky, U. Prange, D. Varro, and S. Varro-Gyapay. Model Transformation by Graph Transformation : A Comparative Study. *Model Transformations in Practice Workshop at MODELS 2005, Montego*, 2005.
- [453] G. Taentzer, M. Ohrndorf, Y. Lamo, and A. Rutle. Change-Preserving Model Repair. In M. Huisman and J. Rubin, editors, *Fundamental Approaches to Software Engineering*, *Lecture Notes in Computer Science*, pages 283–299, Berlin, Heidelberg, 2017. Springer.
- [454] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007.
- [455] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, Jan. 2009.
- [456] J. P. Team. Jakarta ee: Jakarta persistence 3.0, 2020.
- [457] I. J. . I. technology. ISO/IEC 7498-1:1994, 1994.
- [458] T. J. Teorey, D. Yang, and J. P. Fry. A Logical Design Methodology for Relational Databases Using the Extended Entity-relationship Model. *ACM Comput. Surv.*, 18(2):197–222, 1986.
- [459] R. Torkar, T. Gorschek, R. Feldt, M. Svahnberg, U. A. Raja, and K. Kamran. Requirements traceability: a systematic review and industry case study. *International Journal of Software Engineering and Knowledge Engineering*, 22(03):385–433, May 2012. Publisher: World Scientific Publishing Co.

- [460] D. Torre, Y. Labiche, and M. Genero. UML Consistency Rules: A Systematic Mapping Study. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14*, pages 6:1–6:10, New York, NY, USA, 2014. ACM.
- [461] W. Torres, M. v. d. Brand, and A. Serebrenik. Model Management Tools for Models of Different Domains: A Systematic Literature Review. In *2019 IEEE International Systems Conference (SysCon)*, pages 1–8, Apr. 2019. ISSN: 2472-9647.
- [462] W. Torres, M. G. J. van den Brand, and A. Serebrenik. A systematic literature review of cross-domain model consistency checking by model management tools. *Software and Systems Modeling*, Oct. 2020.
- [463] F. Trollmann and S. Albayrak. Extending model to model transformation results from triple graph grammars to multiple models. In D. Kolovos and M. Wimmer, editors, *Theory and Practice of Model Transformations*, pages 214–229, Cham, 2015. Springer International Publishing.
- [464] F. Trollmann and S. Albayrak. Extending model synchronization results from triple graph grammars to multiple models. In P. Van Gorp and G. Engels, editors, *Theory and Practice of Model Transformations*, pages 91–106, Cham, 2016. Springer International Publishing.
- [465] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1937.
- [466] H. Ulrich, J. Kern, D. Tas, A. K. Kock-Schoppenhauer, F. Ückert, J. Ingenerf, and M. Lablans. QL4mdr: a GraphQL query language for ISO 11179-based metadata repositories. *BMC Medical Informatics and Decision Making*, 19(1):45, Mar. 2019.
- [467] U.S. Government Accountability. Health Care: National Strategy Needed to Accelerate the Implementation of Information Technology. <http://www.gao.gov/products/GAO-04-947T>, Last Accessed: 08.11.2017, 2004.
- [468] U.S. Government Accountability. Health and Human Services' Estimate of Health Care Cost Savings Resulting from the Use of Information Technology. <http://www.gao.gov/products/GAO-05-309R>, Last Accessed: 08.11.2017, 2005.
- [469] M. Usman, A. Nadeem, T.-h. Kim, and E.-s. Cho. A Survey of Consistency Checking Techniques for UML Models. In *Proceedings of the 2008 Advanced Software Engineering and Its Applications, ASEA '08*, pages 57–62, USA, Dec. 2008. IEEE Computer Society.
- [470] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using Description Logic to Maintain Consistency between UML Models. In P. Stevens, J. Whittle, and G. Booch, editors, *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*, Lecture Notes in Computer Science, pages 326–340, Berlin, Heidelberg, 2003. Springer.

BIBLIOGRAPHY

- [471] B. C. van Fraassen. Scientific Representation: Paradoxes of Perspective. *Analysis*, 70(3):511–514, July 2010.
- [472] D. Varró, G. Bergmann, A. Hegedüs, A. Horváth, I. Ráth, and Z. Ujhelyi. Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework. *Software & Systems Modeling*, 15(3):609–629, July 2016.
- [473] K. H. Veltman. Syntactic and semantic interoperability: New approaches to knowledge and the semantic web. *New Review of Information Networking*, 7(1):159–183, Jan. 2001.
- [474] M. Voelter. *DSL Engineering: Designing, Implementing and Using Domain-specific Languages*. CreateSpace Independent Publishing Platform, 2013.
- [475] O. C. von Bargaen. Integration of Web Services and their data models with special regard to GraphQL. Master’s thesis, Høgskulen på Vestlandet, Bergen, Norway, 2020.
- [476] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, June 2013.
- [477] J. G. Walls, G. R. Widmeyer, and O. A. El Sawy. Building an Information System Design Theory for Vigilant EIS. *Information Systems Research*, 3(1):36–59, 1992. Publisher: INFORMS.
- [478] R. F. C. Walters. *Categories and Computer Science*. Cambridge University Press, New York, NY, USA, 1992.
- [479] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [480] A. I. Wasserman. Tool integration in software engineering environments. In F. Long, editor, *Software Engineering Environments*, Lecture Notes in Computer Science, pages 137–149, Berlin, Heidelberg, 1990. Springer.
- [481] J. H. Weber and C. Kuziemsky. Pragmatic Interoperability for Ehealth Systems: The Fallback Workflow Patterns. In *Proceedings of the 1st International Workshop on Software Engineering for Healthcare, SEH ’19*, pages 29–36, Piscataway, NJ, USA, 2019. IEEE Press. event-place: Montreal, Quebec, Canada.
- [482] N. Weidmann, A. Anjorin, L. Fritsche, G. Varró, A. Schürr, and E. Leblebici. Incremental Bidirectional Model Transformation with eMoflon: : IBeX. In J. Cheney and H.-S. Ko, editors, *Proceedings of the 8th International Workshop on Bidirectional Transformations co-located with the Philadelphia Logic Week, Bx@PLW 2019, Philadelphia, PA, USA, June 4, 2019*, volume 2355 of *CEUR Workshop Proceedings*, pages 45–55. CEUR-WS.org, 2019.
- [483] N. Weidmann, L. Fritsche, and A. Anjorin. A search-based and fault-tolerant approach to concurrent model synchronisation. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, pages 56–71. Association for Computing Machinery, New York, NY, USA, Nov. 2020.

- [484] R. L. Wexelblat, editor. *History of Programming Languages*. Association for Computing Machinery, New York, NY, USA, 1978.
- [485] A. N. Whitehead and B. Russell. *Principia Mathematica*. 1910.
- [486] J. Whittle, J. Hutchinson, and M. Rouncefield. The State of Practice in Model-Driven Engineering. *IEEE Software*, 31(3):79–85, may 2014.
- [487] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem? In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, editors, *Model-Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 1–17, Berlin, Heidelberg, 2013. Springer.
- [488] J. R. Williams, R. F. Paige, and F. A. C. Polack. Searching for Model Migration Strategies. In *ME '12*, pages 39–44, New York, NY, USA, 2012. ACM.
- [489] S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software & Systems Modeling*, 9(4):529–565, Sept. 2010.
- [490] G. Winskel. Categories of models for concurrency. In *Seminar on Concurrency, Carnegie-Mellon University*, page 246–267, Berlin, Heidelberg, 1984. Springer-Verlag.
- [491] L. Wittgenstein. *Tractatus Logico-Philosophicus*. 1922.
- [492] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer-Verlag, Berlin Heidelberg, 2012.
- [493] U. Wolter. Logics of First-Order Constraints – A Category Independent Approach. *arXiv:2101.01944 [cs]*, Jan. 2021. arXiv: 2101.01944.
- [494] U. Wolter and Z. Diskin. The Next Hundred Diagrammatic Specification Techniques — An Introduction to Generalized Sketches —. Technical report, Department of Informatics, University of Bergen, Bergen, 2007.
- [495] U. Wolter, Z. Diskin, and H. König. Graph Operations and Free Graph Algebras. In *Graph Transformation, Specifications, and Nets*, Lecture Notes in Computer Science, pages 313–331. Springer, Cham, 2018.
- [496] K. Wong. What Grounded the Airbus A380? *Cadalyst*, Dec. 2006.
- [497] A. T. Wood-Harper, L. Antill, and D. E. Avison. *Information systems definition: the Multiview approach*. Blackwell Scientific Publications, Ltd., GBR, 1985.
- [498] World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) 1.2, 2007.
- [499] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi, and H. Mei. Supporting Automatic Model Inconsistency Fixing. In *ESEC/FSE '09*, pages 315–324, New York, NY, USA, 2009. ACM.

BIBLIOGRAPHY

- [500] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, ASE '07*, pages 164–173, Atlanta, Georgia, USA, Nov. 2007. Association for Computing Machinery.
- [501] Y. Xiong, H. Song, Z. Hu, and M. Takeichi. Synchronizing concurrent model updates based on bidirectional transformation. *Software & Systems Modeling*, 12(1):89–104, Feb. 2013.
- [502] W. Yeung. Checking Consistency between UML Class and State Models Based on CSP and B. *Journal of Universal Computer Science*, 10(11):1540–1559, Nov. 2004.

LITERATURE STUDY REFINEMENTS

Chap. 4 contains a literature study investigating approaches related to multi-model consistency management. It is based on the content of [451]. On the occasion of writing this thesis, that study had been extended and as a result the presentation of the feature model had been modified. Concerning the extension, Chap. 4 additionally covers the survey papers [461], [190], [350], and [416], adding an increased number of background material. Concerning the feature model, the following sections will explain the detailed changes of the feature model in this thesis, compared to the one in [451]:

Model

The changes to the Models feature dimension in [451] are visualised in Fig. A.1 and highlighted by red colour. I performed some renaming of features. For example, Trees have been renamed to Terms since the latter is the more general term. The sub-features of the abstract XML feature (now named XML-based) have been reorganised: EMF is now subsumed by XMI, Free Form and XML are the new XML feature since XSD is rightly an aspect of the Conformance dimension. Moreover, RDF is introduced as a new concrete XML-based technology for model representation. Finally, I introduced DSLs as a new concrete feature for representing models.

Change

The changes to the Changes dimension in [451] are depicted in Fig. A.2. The major change is a reorganisation on the top layer. Definition and Types are now grouped under the newly introduced abstract feature Allowed Updates, which makes the presentation more clear. Making Definition an optional sub feature makes the Undefined feature obsolete. Moreover, Recording is moved under Representation since it is recording is related to representation. Moreover, I introduced a more refined distinction beneath the Change Identification Procedure feature (now named Model Differencing) and under Definition::Customizable.

Conformance

The changes to the Formats dimension in [451] (now named Conformance to align it with the terminology of Chap. 3) are shown in Fig. A.3. The major change is that

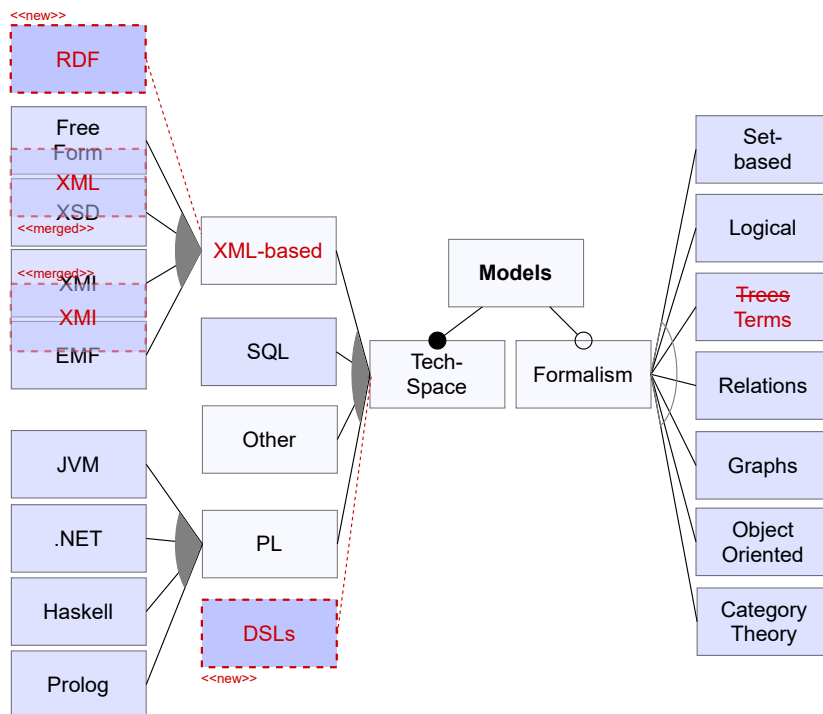


Fig. A.1: Differences w.r.t. the Models feature dimension

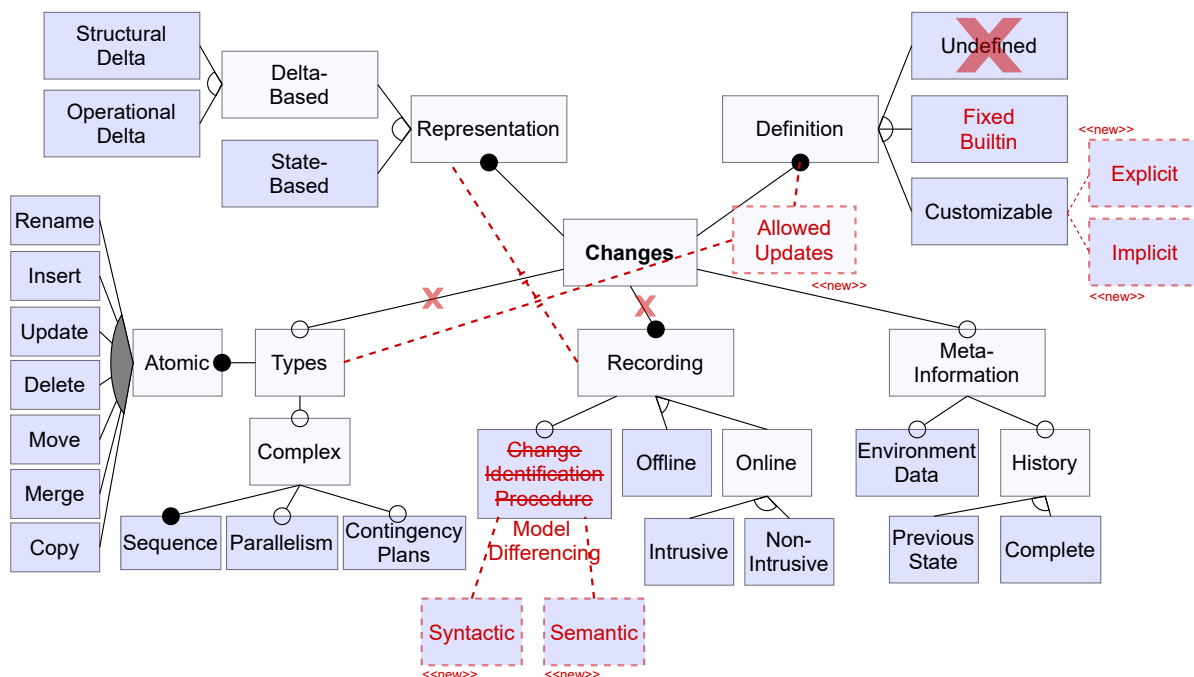


Fig. A.2: Differences w.r.t. the Change feature dimension

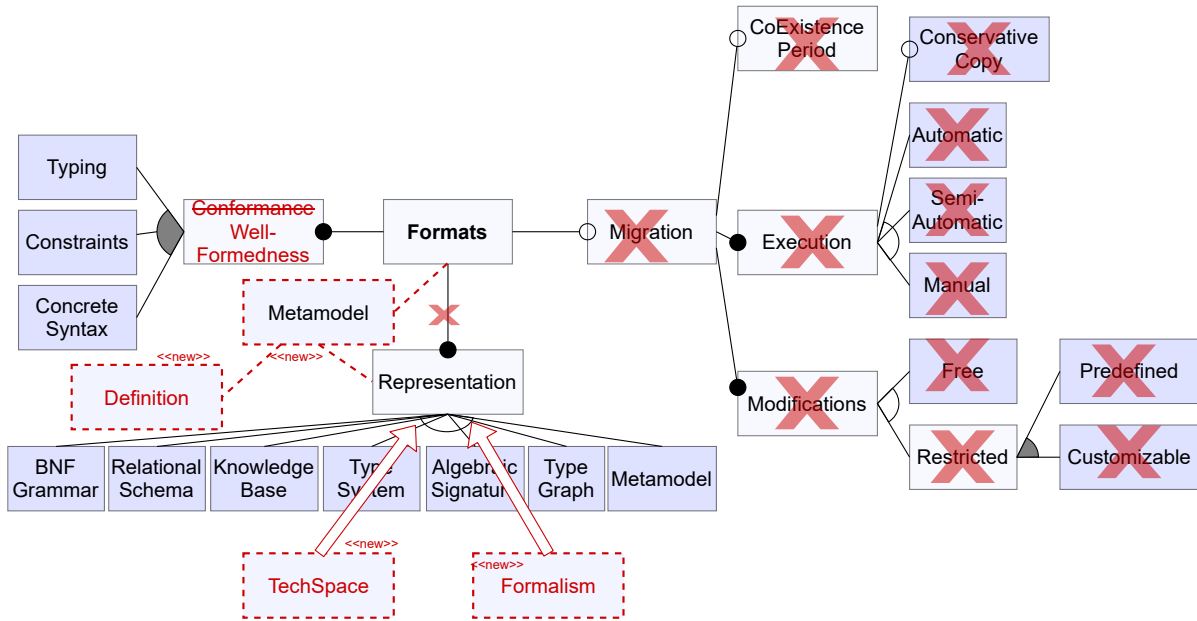


Fig. A.3: Differences w.r.t. the Conformance feature dimension

Migration and all its children are now removed since metamodel-model co-evolution is not the topic of this thesis. A minor change is that Conformance has had to be renamed to Well-Formedness to avoid name collision. The Representation feature is moved beneath the new abstract feature Metamodel, which additionally investigates ways of defining metamodels. Under the Representation aspect, the distinction is now split into TechSpace and Formalism, similar to the Models feature. In this context, also new technologies and formalisms had been added, e.g. OWL is introduced as a way for representing metamodels.

Correspondence

The changes to the Multi-Models dimension (now named Correspondence to align it with the terminology in Chap. 3) is shown in Fig. A.4. There are some changes in terminology, which are highlighted in the figure. The features Privacy, Authority, Arity are now all grouped beneath Properties. The Communication aspect had been deleted because it has little importance to my work. The two major changes are the introduction of the abstract feature Commonalities and General Architecture. The latter subsumes the old Synthesis feature and additionally considers the Maintainer Network approach, see Sec. 3.3.5. The Commonalities feature now groups the aspects Definition and Representation. The latter has been extended by Complement-based approaches, which are common in programming-based BX. Finally, Matching Procedure is now its own top-level feature (beneath Operations).

Consistency

The changes to the Consistency dimension in [451] are shown in Fig. A.5. Here, I mainly added new features, e.g. PL-based means for Rule::Definition, Other Formalisms, and the

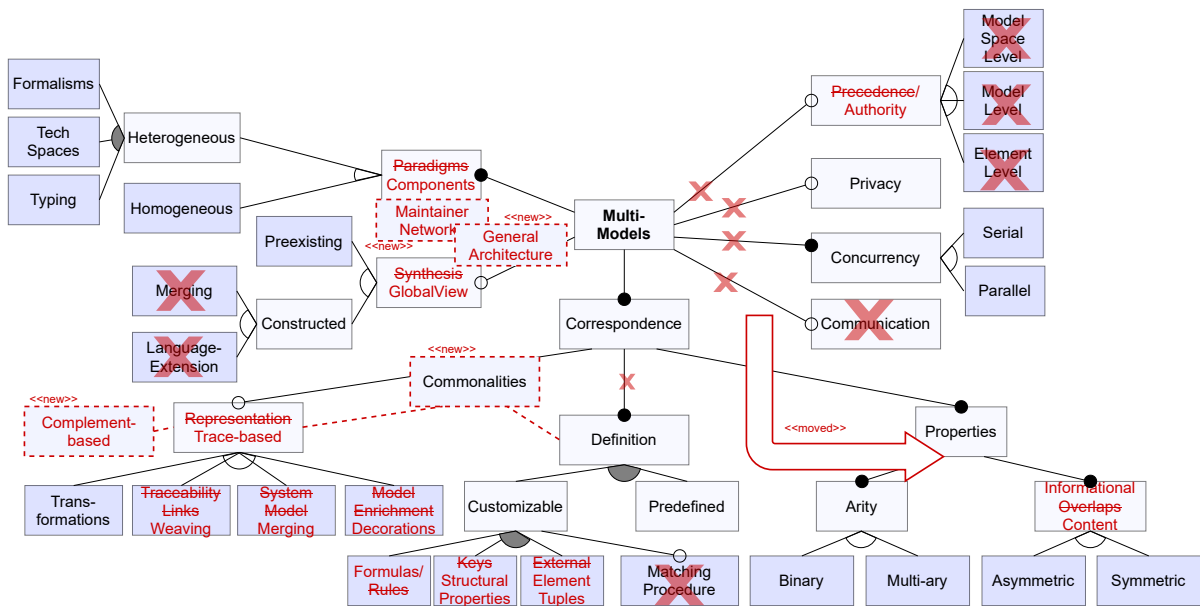


Fig. A.4: Differences w.r.t. the Correspondence feature dimension

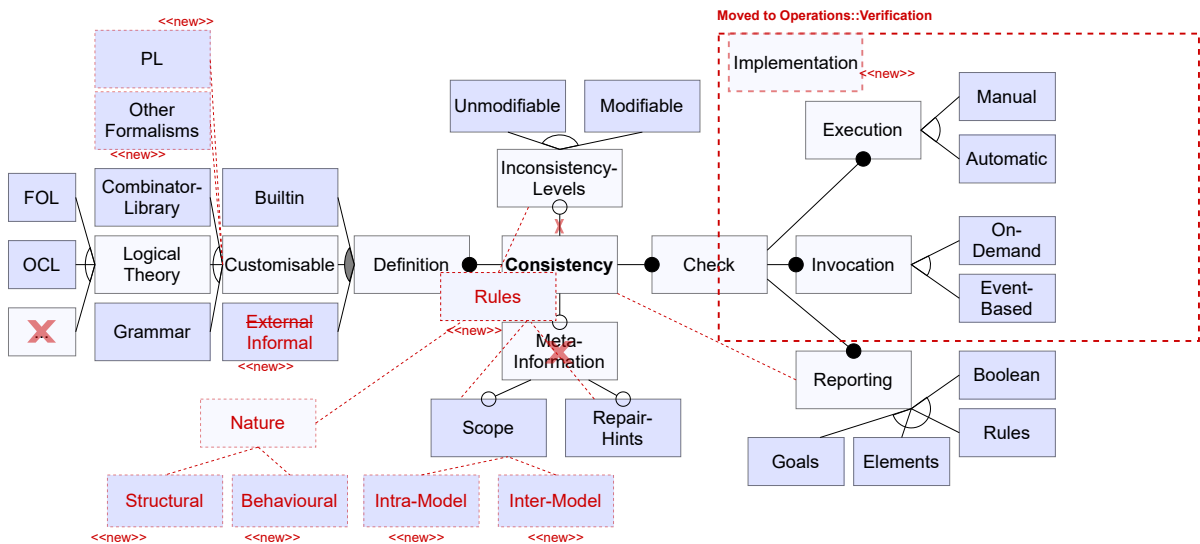


Fig. A.5: Differences w.r.t. the Consistency feature dimension

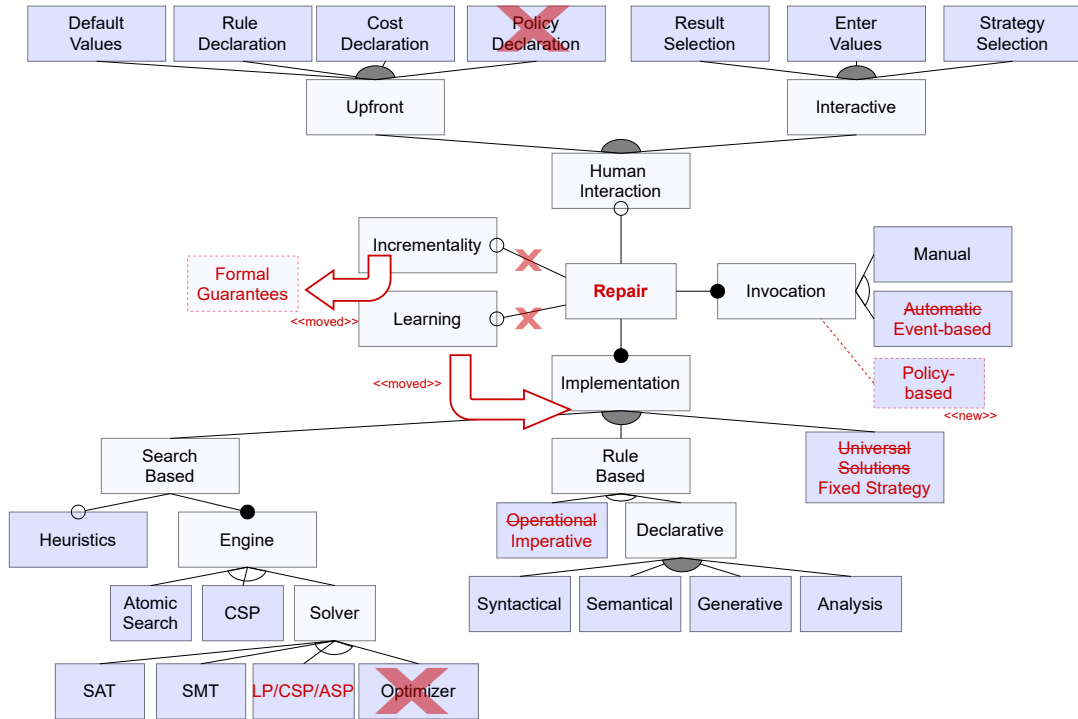


Fig. A.6: Differences w.r.t. the Repair feature dimension (1 / 2)

distinction between Behavioural and Structural rules. The abstract feature Check (now called Verification) has become its own top-level feature (beneath Operations), which also investigates implementation approaches now.

Operations

In [451], I considered Repair as the only real operation. In Chap. 4, I added Verification and Matching to the list of operations. Beneath the Repair feature, I did some changes, which are highlighted in Fig. A.6 and Fig. A.7, respectively. The general distinction into Implementation, Human Interaction, Properties (now named Formal Guarantees), Invocation, and Result has stayed the same. Some features have moved down in the tree and some features have been renamed, which is visible in the figures. The feature

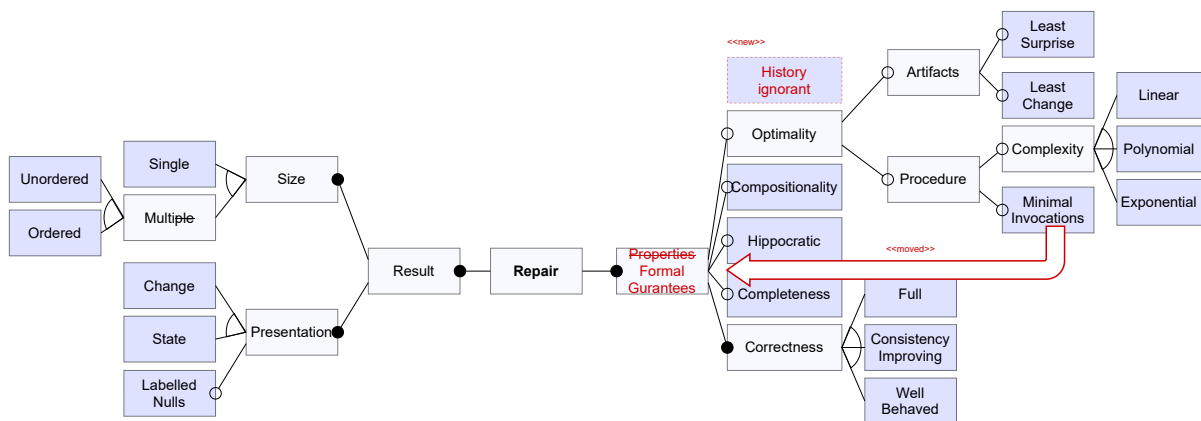


Fig. A.7: Differences w.r.t. the Repair feature dimension (2 / 2)

Optimizer has been removed, because it turned out that the only related study [312] used optimisation for “model matching” and not for “model repair”.

APPENDIX B

PROOFS

B.1 Proof of Theorem 8

The schema for this proof is sketched in (B.1). The proof of Prop 4 showed that \mathbb{M} is a full subcategory of the diagram category $\mathbb{G}^{\mathbb{I}} = (\mathbf{Set}^{\mathbb{B}})^{\mathbb{I}}$. Using *cartesian closedness* of the category of small categories (Fact 41), I get that $\mathbf{Set}^{\mathbb{B} \times \mathbb{I}} \cong (\mathbf{Set}^{\mathbb{B}})^{\mathbb{I}}$ (Fact 40). Intuitively, this means that a functor with two arguments (of type \mathbb{B} and \mathbb{I} , resp.) can be *curried*, i.e. it can be interpreted as a family of functors, each of which has one argument of type \mathbb{B} , and the family varying over a parameter of type \mathbb{I} . Finally, we define an auxiliary category \mathbb{N} (B.1) as a subcategory of $\mathbf{Set}^{\mathbb{B} \times \mathbb{I}}$ of those functors $N : \mathbb{B} \times \mathbb{I} \rightarrow \mathbf{Set}$ that map “the same” morphism (modulo the construction in [3]) to monomorphisms as in \mathbb{M} (B.1). Finally, we show that \mathbb{N} is isomorphic to \mathbb{CS} .

$$\begin{array}{ccccc}
 \mathbf{Set}^{\mathbb{B} \times \mathbb{I}} & \xleftarrow{\cong [3, 27.3 (e)]} & & \xrightarrow{\cong} & (\mathbf{Set}^{\mathbb{B}})^{\mathbb{I}} & \quad (B.1) \\
 \uparrow \text{(Sect. B.1)} & & & & \uparrow \text{(Prop. 4)} & \\
 \mathbb{CS} & \xleftarrow{\cong \text{(Sect. B.1)}} & \mathbb{N} & \xleftarrow{\cong \text{(Sect. B.1)}} & \mathbb{M} &
 \end{array}$$

Definition of \mathbb{N}

Because \mathbb{I} contains $2n + 1$ objects, the cartesian product $\mathbb{B} \times \mathbb{I}$ of \mathbb{B} and \mathbb{I} in the category of small categories has $2n + 1$ copies $\mathbb{B}_{-n}, \dots, \mathbb{B}_0, \dots, \mathbb{B}_n$ of \mathbb{B} together with arrow spans

$$(s, 0) \xleftarrow{(\text{id}_s, j0)} (s, -j) \xrightarrow{(\text{id}_s, jj)} (s, j)$$

for each $j \in \{1, \dots, n\}$ and for each $s \in |\mathbb{B}|$.

An example of a product category for $n = 2$ is shown in (5.19) in Example 5.2. Hence, one obtains $2n + 1$ columns in (5.19).

Hence, objects in $\mathbf{Set}^{\mathbb{B} \times \mathbb{I}}$ are functors $N : \mathbb{B} \times \mathbb{I} \rightarrow \mathbf{Set}$, which simultaneously act as $2n + 1$ functors from \mathbb{B} to \mathbf{Set} , augmented with spans

$$N((s, 0)) \xleftarrow{N((\text{id}_s, j0))} N((s, -j)) \xrightarrow{N((\text{id}_s, jj))} N((s, j))$$

of total functions for each $s \in |\mathbb{B}|$ and all $j \in \{1, \dots, n\}$. In (5.19) these are the two spans

in the top row, if $s = V$. For $s = E$ these spans occur in the bottom row in (5.19).

I define \mathbb{N} to be the subcategory of $\mathbf{Set}^{\mathbb{B} \times \mathbb{I}}$ of functors N , which maps all (id_s, j_0) to monomorphisms (injective functions) in \mathbf{Set} for all $s \in |\mathbb{B}|$.

Equivalence of \mathbb{N} and \mathbb{M}

The equivalence between $\mathbf{Set}^{\mathbb{B} \times \mathbb{I}}$ and $(\mathbf{Set}^{\mathbb{B}})^{\mathbb{I}}$ is based on currying and un-currying the respective functor definitions. The category \mathbb{M} has imposed the restriction, that all images $M(j_0) : \mathcal{M}(-j) \rightarrow \mathcal{M}(0)$ of j_0 under an \mathbb{M} -object \mathcal{M} are monomorphisms in \mathbb{G} . The latter is represented as a family $(M(j_0)(s) : \mathcal{M}(-j)(s) \rightarrow \mathcal{M}(-j)(s))_{s \in |\mathbb{B}|}$, which has a one-to-one correspondence with the family $N((s, j_0))$ ($s \in |\mathbb{B}|, j \in \{1, \dots, n\}$).

Equivalence of \mathbb{CS} and \mathbb{N}

Let $N \in |\mathbb{N}|$ and $C \in |\mathbb{CS}|$. We will show that every comprehensive system C has an equivalent representation as an \mathbb{N} -object. First, we define a one-to-one correspondence within C 's components. We identify

- $N((s, i))$ and $C_i(s)$ for all $s \in |\mathbb{B}|, 0 \leq i \leq n$ (1. in Definition 5.19).
- and $N((\text{op}, \text{id}_i)) : N((s, i)) \rightarrow N((s', i))$ and $C_i(\text{op}) : C_i(s) \rightarrow C_i(s')$ for all $\text{op} : s \rightarrow s' \in \mathbb{B}^{\rightarrow}, 0 \leq i \leq n$ (2. in Def. 5.19).

Furthermore, in Sec. 5.1.3 it was demonstrated, how a partial morphism in some category can be expressed by an equivalence class of spans with the inner leg being a monomorphism. Therefore,

- $N((s, k))$ for all $s \in |\mathbb{B}|, -n \leq k < 0$ (the apex of the span),
- $N((\text{id}_s, j_0)) : N(s, -j) \rightarrow N(s, 0)$ for all $s \in |\mathbb{B}|, 0 < j \leq n$ (the domain embedding),
- and $N((\text{id}_s, jj)) : N(s, -j) \rightarrow N(s, j)$ for all $s \in |\mathbb{B}|, 0 < j \leq n$ (the concrete assignment)

form a concrete representative of the projection $p_{j,s}^C : C_0(s) \rightarrow p_{j,s}^C$ in C .

The remaining constituents of N :

- $N((\text{op}, j_0))$ for all $0 < j \leq n$ and non-identity morphisms $\text{op} : s \rightarrow s' \in \mathbb{B}^{\rightarrow}$,
- $N((\text{op}, jj))$ for all $0 < j \leq n$ and non-identity morphisms $\text{op} : s \rightarrow s' \in \mathbb{B}^{\rightarrow}$

are subject to the following equations, which are a consequence of the definition of composition in the product category $\mathbb{B} \times \mathbb{I}$ (compare with the diagonals in (5.19)) and of N being a functor (which must preserve these compositions):

$$\begin{aligned} N((\text{id}_{s'}, jj)) \circ N((\text{op}, \text{id}_{-j})) &= N((\text{id}_{s'}, jj) \circ (\text{op}, \text{id}_{-j})) \\ &= N((\text{op}, jj)) \\ &= N((\text{op}, \text{id}_j) \circ (\text{id}_s, jj)) \\ &= N((\text{op}, \text{id}_j)) \circ N((\text{id}_s, jj)) \end{aligned}$$

$$\begin{aligned}
N((\text{id}_{s'}, j_0)) \circ N((\text{op}, \text{id}_{-j})) &= N((\text{id}_{s'}, j_0) \circ (\text{op}, \text{id}_{-j})) \\
&= N((\text{op}, j_0)) \\
&= N((\text{op}, \text{id}_j) \circ (\text{id}_s, j_0)) \\
&= N((\text{op}, \text{id}_j)) \circ N((\text{id}_s, j_0))
\end{aligned}$$

These conditions correspond to the generalised edge-node incidence (5.13)+(5.14).

Hence $\mathbb{CS} \cong \mathbb{M}$, as claimed. \square

B.2 Proof of Theorem 9

Let $(m : B \rightarrow D, f : C \rightarrow D)$ be a co-span in \mathbb{CS} . Utilizing Theorem 8, I choose to perform the proof in \mathbb{M} , i.e. showing the existence of $(g : A \rightarrow B, n : A \rightarrow C)$ as a pullback for (m, f) in \mathbb{M} .

Recall that $\mathbb{G} = \mathbf{Set}^{\mathbb{B}}$ has all pullbacks, which are constructed component-wise for each $s \in |\mathbb{B}|$ (Lemma 32). The component-wise construction lifts to $\mathbb{G}^{\mathbb{I}}$ (for each $i \in \mathbb{I}$) resulting in the j -indexed family of cubes as shown in (B.2). Note that for any object $\mathcal{M} \in |\mathbb{M}|$, the span $(\mathcal{M}(0) \leftarrow \mathcal{M}(-j) \rightarrow \mathcal{M}(j))$ in (5.8) can be seen as a partial \mathbb{G} -morphism $p_j^{\mathcal{M}} : \mathcal{M}^0 \rightarrow \mathcal{M}^j$.

$$\begin{array}{c}
\begin{array}{ccccc}
& & A^0 & \xrightarrow{n_0} & C^0 \\
& & \uparrow \subseteq_j^A & & \uparrow \subseteq_j^C \\
& B^0 & \xleftarrow{g_0} & D^0 & \xleftarrow{f_0} & C^0 \\
& \uparrow \subseteq_j^B & & \uparrow \subseteq_j^D & & \uparrow \subseteq_j^C \\
& \text{dom}(p_j^A) & \xrightarrow{n_{-j}} & \text{dom}(p_j^C) & & \\
& \uparrow \subseteq_j^B & & \uparrow \subseteq_j^D & & \uparrow \subseteq_j^C \\
& \text{dom}(p_j^B) & \xrightarrow{m_{-j}} & \text{dom}(p_j^D) & & \\
& \uparrow p_j^B & & \uparrow p_j^D & & \uparrow p_j^C \\
& A^j & \xrightarrow{n_j} & C^j & & \\
& \uparrow g_j & & \uparrow f_j & & \\
& B^j & \xrightarrow{m_j} & D^j & &
\end{array} \\
\begin{array}{c}
0 \\
\uparrow j_0 \\
-j \\
\downarrow j_j \\
j
\end{array}
\end{array} \tag{B.2}$$

The spans (g_0, n_0) , (g_{-j}, n_{-j}) , and (g_j, n_j) are constructed component-wise as pullbacks in \mathbb{M} . The universal pullback property of the top face w.r.t to the horizontal inner face in the middle provides the morphism \subseteq_j^A that makes the upper rear faces commute. And the universal pullback property of the bottom face w.r.t to the horizontal inner face in the middle provides the morphism p_j^A that makes the lower rear faces commute.

It remains to show that $A \in |\mathbb{M}|$. For this I have to show that \subseteq_j^A is a monomorphism

Assume two morphisms $x : X \rightarrow \text{dom}(p_j^A)$ and $y : X \rightarrow \text{dom}(p_j^A)$ such that $\subseteq_j^A \circ x = \subseteq_j^A \circ y$. Postcomposing this arrow simultaneously with n_0 and g_0 yields

$$\begin{aligned}
n_0 \circ \subseteq_j^A \circ x &= n_0 \circ \subseteq_j^A \circ y \\
g_0 \circ \subseteq_j^A \circ x &= g_0 \circ \subseteq_j^A \circ y
\end{aligned}$$

using the commutativity of the left and back face, followed by monomorphism property (C.13) of \subseteq_j^B and \subseteq_j^C , I get

$$\begin{aligned} n_{-j} \circ x &= n_{-j} \circ y \\ g_{-j} \circ x &= g_{-j} \circ y \end{aligned}$$

Recall that the horizontal inner face is a pullback, i.e. n_{-j} and g_{-j} are jointly monic (Fact 35) and therefore $x = y$ as required. \square

B.3 Proof of Theorem 5

Let $f : G \rightarrow H \in \mathbb{C}^{\rightarrow}$ be a (signature) morphism, $\mathfrak{C} \in \text{Sen}^{\mathcal{S}(\mathbb{C}^{\Pi})}(\mathbb{C})$ be a sketch (sentence), and $i : I \rightarrow H \in |\mathbb{C} \downarrow H|$ a slice-category object (model).

The following equivalence has to be proven

$$(\mathbb{C} \downarrow _)(f)(i) \models \mathfrak{C} \Leftrightarrow i \models \text{Sen}^{\mathcal{S}(\mathbb{C}^{\Pi})}(f)(\mathfrak{C}) \quad (\text{B.3})$$

Let in the following $i^* : O \rightarrow G$ be the image of $\mathbb{C} \downarrow _)(f)(i)$.

“ \Rightarrow ”:

Assume that $i^* \models (b, p)$ for all diagrammatic constraints $(b, p) \in \text{Constr}(\mathfrak{C})$, i.e. there exists a span $(b' : X \rightarrow O, i' : X \rightarrow \text{ar}(p))$ that is a pullback of $(i^* : O \rightarrow G, b : \text{ar}(p) \rightarrow G)$, see (B.4). Now both squares in (B.4) compose due to the pullback composition lemma (Fact 30). As a consequence, I have that $i \models (f \circ b, p)$. Hence, $i \models \text{Sen}^{\mathcal{S}(\mathbb{C}^{\Pi})}(f)(\mathfrak{C})$ as required.

$$\begin{array}{ccc} O & \xrightarrow{f^*} & I \\ \uparrow b' & \searrow i^* & \downarrow i \\ X & & G \xrightarrow{f} H \\ & \searrow i' & \uparrow b \\ & & \text{ar}(p) \end{array} \quad (\text{B.4})$$

“ \Leftarrow ”:

Assume that $i \models \text{Sen}^{\mathcal{S}(\mathbb{C}^{\Pi})}(f)(\mathfrak{C})$. Hence for all $(f \circ b, p) \in \text{Constr}(\text{Sen}^{\mathcal{S}(\mathbb{C}^{\Pi})}(f)(\mathfrak{C}))$ there is a $((b, f)' : X \rightarrow I, i' : X \rightarrow \text{ar}(p))$ that is the pullback of $(f \circ b : \text{ar}(p) \rightarrow H, i : I \rightarrow H)$. Now, due to the pullback decomposition lemma (Fact 30) there is a unique morphism $b' : X \rightarrow O$ such that (b', i^*, i', b) is a pullback square, see (B.5). Hence, $i^* \models (b, p)$, which yields $(\mathbb{C} \downarrow _)(f)(i) \models \mathfrak{C}$ as required.

$$\begin{array}{ccccc}
X & & & & \\
\downarrow & \searrow^{(b,f)'} & & & \\
O & \xrightarrow{f^*} & I & & \\
\downarrow \lrcorner & \searrow^{i^*} & \downarrow i & & \\
G & \xrightarrow{f} & H & & \\
\uparrow b & & & & \\
\text{ar}(p) & & & &
\end{array}
\tag{B.5}$$

□

B.4 Proof of Proposition 13

Isomorphisms trivially yield naturality squares that are pullbacks. The composition of two reflective monomorphisms is a reflective monomorphism as well because pullbacks compose. To see that reflective monomorphisms are stable under pullback, consider again our pullback construction in \mathbb{M} from the proof of Theorem 9, depicted in (B.2). This time, the upper front face is a pullback and all components of m are monic, i.e. m is a reflective mono. We have to show that all components of n are monic and that the upper back face is a pullback, i.e. n is a reflective mono. The former, however, is easy, since pullbacks in \mathbb{M} preserve monomorphisms (Fact 34): Recall that A was constructed via taking pullbacks component-wise in Theorem 9. For the pullback property consider the diagram in (B.6)

$$\begin{array}{ccccc}
& & \subseteq_j^B \circ g_{-j} & & \\
\text{dom}(p_j^A) & \xrightarrow{\subseteq_j^A} & A^0 & \xrightarrow{g_0} & B^0 \\
\downarrow n_{-j} & & \downarrow n_0 & & \downarrow m_0 \\
\text{dom}(p_j^C) & \xrightarrow{\subseteq_j^C} & C^0 & \xrightarrow{f_0} & D^0 \\
& & \subseteq_j^D \circ f_{-j} & &
\end{array}
\tag{B.6}$$

I use the fact that the upper front-face in (B.2) is a pullback because m is reflective monomorphism by assumption. I compose it with the horizontal inner face in (B.2), which is a pullback by construction resulting in a pullback that forms the outer rectangle in (B.6). The right square (b) in (B.6) is the top face in (B.2) and therefore also a pullback by construction. Now we know that the upper and lower triangles in (B.6) commute because they represent the upper left and upper right faces in (B.2). Therefore, I can use the pullback-decomposition lemma to conclude that (a) is a pullback, as desired. □

B.5 Proof of Theorem 14

Let $m : A \rightarrow B$ and $f : A \rightarrow C$ be a span of comprehensive systems with $m \in \mathcal{M}$ a reflective monomorphism. Again, I construct the pushout $(g : B \rightarrow D, n : C \rightarrow D)$

component-wise in \mathbb{M} like I did in the proof of Theorem 9. The construction is dual to the one in the proof of Theorem 9 and again I have to pay special attention to the upper cubes (images of j_0)

$$\begin{array}{ccccc}
 & & A^0 & \xrightarrow{m_0} & B^0 \\
 & & \uparrow \subseteq_j^A & & \uparrow \subseteq_j^B \\
 C^0 & \xleftarrow{f_0} & & \xrightarrow{g_0} & D^0 \\
 & \xrightarrow{n_0} & & & \\
 \subseteq_j^C \uparrow & & \text{dom}(p_j^A) & \xrightarrow{m_{-j}} & \text{dom}(p_j^B) \\
 & \xleftarrow{f_{-j}} & & \xrightarrow{g_{-j}} & \\
 \text{dom}(p_j^C) & \xrightarrow{n_{-j}} & & & \text{dom}(p_j^D)
 \end{array}
 \tag{B.7}$$

Consider the commutative cube in (B.7). The rear faces are given via the span (m, f) where the back face is a pullback because $m \in \mathcal{M}$. The top and bottom faces are constructed as pushouts and by the universal property of the bottom face pushout, we get the morphism $D(j_0)$ that makes the front and right face commute. Since \mathbb{G} is adhesive [304], pushouts preserve monomorphisms such that n_0 and n_{-j} are monomorphisms.

Next, we show that the front face is a pullback, for this consider the diagram in (B.8).

$$\begin{array}{ccccc}
 & & f_0 \circ \subseteq_j^A & & \\
 & & \curvearrowright & & \\
 \text{dom}(p_j^A) & \xrightarrow{f_{-j}} & \text{dom}(p_j^C) & \xrightarrow{\subseteq_j^C} & C^0 \\
 & \downarrow m_{-j} & \downarrow n_{-j} & \downarrow n_0 & \\
 & (a) & & (b) & \\
 \text{dom}(p_j^B) & \xrightarrow{g_{-j}} & \text{dom}(p_j^D) & \xrightarrow{\subseteq_j^D} & D^0 \\
 & & \curvearrowleft & & \\
 & & g_0 \circ \subseteq_j^B & &
 \end{array}
 \tag{B.8}$$

The square (a) is a pushout by construction (bottom face in B.7) and the outer square is a pullback composed out of the back (pullback due the reflective property of m) and top face (pushouts along monomorphisms in adhesive categories are pullbacks as well) in (B.7). Using the special pullback-pushout property in \mathbb{G} [305, Lemma 6] the square (b) becomes a pullback.

It remains to show that \subseteq_j^D is a monomorphism. For this consider the following **Set**-theoretic argument, which is stable under sort-wise construction (lifts to $\mathbf{Set}^{\mathbb{B}}$): Assume \subseteq_j^D is not monic, then there are two elements $x, y \in \text{dom}(p_j^D)$ that \subseteq_j^D maps to the same element $z \in D^0$. Now, I know that $\text{dom}(p_j^D)$ is the apex of a pushout, therefore n_{-j} and g_{-j} are jointly surjective (Fact 38) and thus x, y must have pre-images x', y' under (n_{-j}, g_{-j}) in $\text{dom}(p_j^C)$ or $\text{dom}(p_j^B)$.

Note that the cases $x' \notin \text{dom}(p_j^C) \wedge y' \notin \text{dom}(p_j^B)$ or $x' \in \text{dom}(p_j^C) \wedge y' \notin \text{dom}(p_j^B)$ disqualify immediately since (a) is a pullback. Also $x', y' \in \text{dom}(p_j^C)$ is not possible since (b) is a pullback and \subseteq_j^C is monic. Therefore consider the case $x', y' \in \text{dom}(p_j^B)$ further. Then x', y' must have distinct images under inclusion \subseteq_j^B in B^0 that must

be mapped to z via g_0 . Now D^0 is also constructed as the apex of a pushout and for $x', y' \in B^0$ being mapped to the same element in D^0 , there must be pre-images of x', y' in A^0 that are mapped to the same element in $z \in C^0$. But $\text{dom}(p_j^C)$ is the pullback object of n_0 and \subseteq_j^D and therefore $z \in C^0$ must have two pre-images in $\text{dom}(p_j^C)$ which violates the monomorphism property of \subseteq_j^C .

Hence, one must conclude that \subseteq_j^D is a monomorphism. \square

B.6 Proof of Theorem 15

Let B be a comprehensive system. For the existence of \mathcal{M} -partial arrow classifiers, I have to show the existence of an \mathcal{M} -morphism $\eta_B : B \rightarrow \mathcal{L}B$ such that for every span $(f : X \rightarrow B, m : X \rightarrow A)$ there exists a unique morphism $\overline{[m, f]} : A \rightarrow \mathcal{L}B$ such that the resulting square is a pullback. Again I perform the construction in \mathbb{M} and focus on the image of j_0 's. For the j_j 's some aspects need not be considered (e.g. pullbacks and monos), hence from the proof for the j_0 's the proof for the j_j 's follows.

$$\begin{array}{ccccc}
 & & X^0 & \xrightarrow{m_0} & A^0 \\
 & & \uparrow \subseteq_j^X & & \uparrow \subseteq_j^A \\
 B^0 & \xleftarrow{f_0} & X^0 & \xrightarrow{m_0} & A^0 \\
 & & \downarrow \eta_{B^0} & & \downarrow \subseteq_j^A \\
 & & \mathcal{L}B^0 & \xleftarrow{\overline{[m_0, f_0]}} & A^0 \\
 & & \downarrow \eta_{\text{dom}(p_j^X)} & & \downarrow \subseteq_j^A \\
 \text{dom}(p_j^X) & \xrightarrow{m_{-j}} & \mathcal{L}\text{dom}(p_j^X) & \xrightarrow{\overline{[\eta_{\text{dom}(p_j^X)}, \subseteq_j^B]}} & \text{dom}(p_j^A) \\
 & & \downarrow \eta_{\text{dom}(p_j^B)} & & \downarrow \subseteq_j^A \\
 \text{dom}(p_j^B) & \xrightarrow{f_{-j}} & \mathcal{L}\text{dom}(p_j^B) & \xrightarrow{\overline{[m_{-j}, f_{-j}]}} & \text{dom}(p_j^A) \\
 & & \downarrow \eta_{\text{dom}(p_j^B)} & & \downarrow \subseteq_j^A \\
 & & \mathcal{L}\text{dom}(p_j^B) & \xrightarrow{\overline{[m_{-j}, f_{-j}]}} & \text{dom}(p_j^A)
 \end{array}
 \tag{B.9}$$

Consider the cube in (B.9). I use the fact that \mathbb{G} has mono-partial arrow classifiers and construct the partial arrow classifier of B in \mathbb{M} component-wise. This gives the existence of unique (dashed arrows) $\overline{[m_0, f_0]}$ and $\overline{[m_{-j}, f_{-j}]}$ such that the top and bottom face become pullbacks. Using the partial arrow classifier property of $\eta_{B^0} : B^0 \rightarrow \mathcal{L}B^0$ w.r.t. $[\eta_{\text{dom}(p_j^B)}, \subseteq_j^B]$ yields $\overline{[\eta_{\text{dom}(p_j^B)}, \subseteq_j^B]}$ such that the front face becomes a pullback, i.e. η_B is a reflective monomorphism if it is a monomorphism. The former is due to the fact that if in a partial arrow span $[m, f]$, f is a monomorphism, then also $\overline{[m, f]}$ is a monomorphism, see Lemma 45. Finally, I have to show that the right face commutes, i.e. $\overline{[m_0, f_0]} \circ \subseteq_j^A = \overline{[\eta_{\text{dom}(p_j^B)}, \subseteq_j^B]} \circ \overline{[m_{-j}, f_{-j}]}$ consider the diagram (B.10).

$$\begin{array}{ccc}
 \text{dom}(p_j^X) & \xrightarrow{m_{-j}} & \text{dom}(p_j^A) \\
 \downarrow f_0 \circ \subseteq_j^X & & \downarrow \subseteq_j^A \\
 B^0 & \xrightarrow{\eta_{B^0}} & \mathcal{L}B^0
 \end{array}
 \tag{B.10}$$

$\overline{[\eta_{\text{dom}(p_j^B)}, \subseteq_j^B]} \circ \overline{[m_{-j}, f_{-j}]}$ (left arrow from $\text{dom}(p_j^A)$ to $\mathcal{L}B^0$)
 $\overline{[m_{-j}, f_0 \circ \subseteq_j^X]}$ (right arrow from $\text{dom}(p_j^A)$ to $\mathcal{L}B^0$)
 $\overline{[m_0, f_0]} \circ \subseteq_j^A$ (right arrow from $\mathcal{L}B^0$ to $\text{dom}(p_j^A)$)

Using the partial map $[m_{-j}, f_0 \circ \subseteq_j^X]$ w.r.t. η_{B^0} , there is a unique (dashed) arrow making the square a pullback. Thus

$$\overline{[m_0, f_0]} \circ \subseteq_j^A = \overline{[m_{-j}, f_0 \circ \subseteq_j^X]} = \overline{[\eta_{\text{dom}(p_j^B)}, \subseteq_j^B]} \circ \overline{[m_{-j}, f_{-j}]}$$

□

B.7 Proof of Proposition 17

(The proof is performed in \mathbb{M}) For the first part of the proof let $\mathbf{0}$

$$\mathbf{0} := (\emptyset : \emptyset \rightarrow \emptyset)_{1 \leq j \leq n}$$

be the totally undefined multi-model span. It is easy to see that this object must be an initial object in \mathbb{M} , which immediately follows from the fact that the empty graph \emptyset is the initial object in \mathbb{G} .

For the second part of the proof let \mathcal{M} and \mathcal{N} be two multi-model spans. Consider (B.11) and recall that \mathbb{G} has coproducts.

$$\begin{array}{ccccc}
 & & M^0 + N^0 & & \\
 & \nearrow \lambda_0 & & \nwarrow \kappa_0 & \\
 M^0 & & \text{dom}(p_j^M) + \text{dom}(p_j^N) & & N^0 \\
 \uparrow \subseteq_j^M & \nearrow \lambda_{-j} & & \nwarrow \kappa_{-j} & \uparrow \subseteq_j^N \\
 \text{dom}(p_j^M) & & M^j + N^j & & \text{dom}(p_j^N) \\
 \downarrow p_j^M & \nearrow \lambda_j & & \nwarrow \kappa_j & \downarrow p_j^N \\
 M^j & & & & N^j
 \end{array} \tag{B.11}$$

Hence, one can construct coproducts for each component yielding the respective family of coproduct injections λ and κ . Due to the universal coproduct product of $\text{dom}(p_j^M) + \text{dom}(p_j^N)$ there are morphisms $u_j : \text{dom}(p_j^M) + \text{dom}(p_j^N) \rightarrow M^0 + N^0$ and $v_i : \text{dom}(p_j^M) + \text{dom}(p_j^N) \rightarrow M^j + N^j$ for all $1 \leq j \leq n$ making the respective diagrams commute. To show that the family of spans $M^0 + N^0 \xleftarrow{u_j} \text{dom}(p_j^M) + \text{dom}(p_j^N) \xrightarrow{\kappa_j} M^j + N^j$ forms a multi-model span one has to verify that the u 's are monomorphisms.

For this, note that coproducts extend to a functor $+$: $\mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}$ and recall that \mathbb{G} is *extensive* [81], i.e.

$$\mathbb{G} \downarrow A \times \mathbb{G} \downarrow B \cong \mathbb{G} \downarrow A + B \tag{B.12}$$

is an equivalence between slice categories. Now, note that that the pair of monomorphisms $\subseteq_j^M : \text{dom}(p_j^M) \hookrightarrow M^0$ and $\subseteq_j^N : \text{dom}(p_j^N) \hookrightarrow N^0$, which can be considered as objects of the comma categories $\mathbb{G} \downarrow M^0$ and $\mathbb{G} \downarrow N^0$ respectively, are a monomorphism in $\mathbb{G} \downarrow M^0 \times \mathbb{G} \downarrow N^0$ (products preserve monos). Hence, due to the equivalence in (B.12), u_i , which is the image of $(\subseteq_j^M, \subseteq_j^N)$ under $+$, must be a monomorphism as well. □

B.8 Proof of Theorem 18

An immediate consequence of the definitions of \mathbb{M} in terms of coproducts is that T is injective on objects and on morphism sets, hence an embedding, such that it remains to show preservation of pushouts.

As seen in Theorem 14, pointwise pushout construction of a span in \mathbb{M} may fail to belong to \mathbb{M} . This obstacle can be overcome because I use coproducts in the construction of T . Let $\nu : \mathcal{M} \rightarrow \mathcal{N} := T(\mathfrak{n} : D \rightarrow H)$, then the naturality squares $\nu_0 \circ \subseteq_j^M = \subseteq_j^N \circ \nu_{-j}$ (images of $j0$'s) are pullbacks due to the following:

The definition of $\mathcal{M}(j0)$ can also be written

$$\mathcal{M}(j0) : \coprod_{r \in R} D_r^j \hookrightarrow \coprod_{r \in R} D(r)$$

with $D_r^j = D(r)$, if there is $f \in \mathbb{I}^{\rightarrow}(_, j)$ and $r = \text{dom}(f)$, and $D_r^j = 0$ (the initial object, see Appendix C.3, i.e. the empty graph) otherwise, because $X + 0 \cong X$ in \mathbb{G} . Similarly, this inclusion can be extended for M^1 . In both cases the summand-wise squares

$$\begin{array}{ccc} D(r) & \xrightarrow{n_r} & H(r) \\ \text{id or } 0_{D^0(r)} \uparrow & & \uparrow \text{id or } 0_{D^1(r)} \\ D_r^j & \xrightarrow{\text{id}_r \text{ or } \text{id}_0} & H_r^j \end{array} \quad (\text{B.13})$$

are pullbacks, such that it suffices to show that two pullback squares in \mathbb{G} always add up to a pullback square of their coproducts, see (B.14).

$$\begin{array}{ccc} \begin{array}{ccc} A_1 & \xrightarrow{h_1} & B_1 \\ k_1' \uparrow & \text{(p.b.)} & \uparrow k_1 \\ C_1 & \xrightarrow{h_1'} & D_1 \end{array} & \begin{array}{ccc} A_2 & \xrightarrow{h_2} & B_2 \\ k_2' \uparrow & \text{(p.b.)} & \uparrow k_2 \\ C_2 & \xrightarrow{h_2'} & D_2 \end{array} & \Rightarrow & \begin{array}{ccc} A_1 + A_2 & \xrightarrow{h_1+h_2} & B_1 + B_2 \\ k_1'+k_2' \uparrow & \text{(p.b.)} & \uparrow k_1+k_2 \\ C_1 + C_2 & \xrightarrow{h_1'+h_2'} & D_1 + D_2 \end{array} \end{array} \quad (\text{B.14})$$

This can be demonstrated as follows: \mathbb{G} is known to be *extensive*, see Sec. B.7, i.e. the functor $+: \mathbb{G} \downarrow B_1 \times \mathbb{G} \downarrow B_2 \rightarrow \mathbb{G} \downarrow (B_1 + B_2)$ between comma categories is an equivalence of categories, its inverse is taking pullbacks along coproduct injections [81]. This adds pullbacks adjacent on the right of the two left pullbacks in (B.14) and, by pullback composition [34], we obtain two pullbacks with the arrow $k_1 + k_2$ as right vertical arrow. Since \mathbb{G} is a topos [207], it can be shown that these two then add to the right pullback in (B.14), see §5.3. in [207].

Now, consider the cube from (B.7) in the proof of theorem 14. This time left and back faces are pullbacks. Using the fact that pushouts in \mathbb{G} are mono-hereditary [227], I conclude that front and right faces are pullbacks and that \subseteq_j^D is a monomorphism, i.e. the result is actually a comprehensive system. Hence, I have to show that all components are pushouts, i.e. the right squares in (B.15) are pushouts in \mathbb{G} for all

$i \in \mathbb{I}^\rightarrow$.

$$\begin{array}{ccccc}
 D & \xrightarrow{m} & G^I & & M & \xrightarrow{\mu} & N & & \mathcal{M}(i) & \xrightarrow{\mu_i} & \mathcal{N}(i) & & \\
 \Downarrow f & & \Downarrow f' & & \Downarrow \phi & & \Downarrow \phi' & & \phi_i \downarrow & & \downarrow \phi'_i & & \\
 H & \xrightarrow{m'} & J & & K & \xrightarrow{\mu'} & L & & \mathcal{K}(i) & \xrightarrow{\mu'_i} & \mathcal{L}(i) & &
 \end{array} \tag{B.15}$$

This is, however, clear from the definition of T for $i > 0$ (because models are untouched and the left square is a pushout by assumption). For $i \leq 0$, all four objects in the right square are coproducts over a certain indexing set I ($I = \mathbb{I}^\rightarrow$ for $i = 0$ and $I = \mathbb{I}^\rightarrow(_, j)$ for $i = -j < 0$), where the coproduct amalgamates relation graphs of the graph diagrams (index $r \in R$).

Finally, since $\llbracket _ \rrbracket$ is a functor from \mathbb{G}^I to \mathbb{G} , which is left-adjoint to the diagonal functor Δ_I (cf. [34, Ex.13.2.4]), it preserves colimits, hence all squares are pushouts, because in the left square there are pointwise pushouts separately for each relation index $r \in R$. \square

CATEGORY THEORY ESSENTIALS

C.1 History and Background

Category theory was originally invented by Eilenberg and MacLane [152] to bridge two seemingly disparate mathematical domains: *topology* (i.e. the study of abstract shapes) and *algebra* (i.e. the study of abstract equations). It allowed to “translate” results from one field to the other and vice versa. Shortly after its inception, Grothendieck used category theory to advance the theory about algebraic geometry [212]. Lawvere recognised that category theory could serve as an alternative foundation to mathematics and, in his frequently cited PhD thesis [309], showed how algebraic theories are interpreted in the categorical framework. His work paved the way for the categorical study of logic and algebra. Today, teaching universal logic, model theory and universal algebra generally encompasses teaching category theory as well. Lambek [306] made the discovery that the theoretical foundation of functional programming (i.e. the typed λ -calculus) can be described by a special type of categories (i.e. cartesian closed categories) and Moggi [345] proposed a categorical concept (i.e. monads) to model effects in functional programming. Since then category theory has become a popular tool for formal investigations about functional programming [344]. Also other areas of (theoretical) computer science have been affected by category theory [203]. For instance, algebraic specification [415], concurrency theory [490], or formal verification [205]. In the context of software engineering, category theory has been used as a means of data specification [117, 252] and as the foundation for algebraic graph transformation [149], a popular formalism for describing model transformations [452]. Recently, quantum physics has been described by means of category theory [93] based on its visual and simultaneously formal (string) diagram language [425]. This is a rather compact summary of the historical development of category theory and its applications, for a more detailed account see [298].

The central authoritative textbook about category theory is MacLane’s *Categories for the Working Mathematician* [326], which, however, is rather advanced and targeted primarily at mathematicians. A more gentle introduction for a broader audience is found in Lawvere’s *Conceptual Mathematics* [310]. There are also textbooks targetted specifically at computer scientists and software engineers [34, 171, 380, 478]. Spivak and Fong’s *Seven Sketches* [177] is the most recent introductory textbooks, which emphasizes the practical applicability of category theory in Science and Engineering. In the following, I will give a short presentation of the concept required for the

understanding of Chap. 5 in order to make this thesis self-contained. For a more background information, I refer to the textbooks mentioned above.

C.2 Introduction

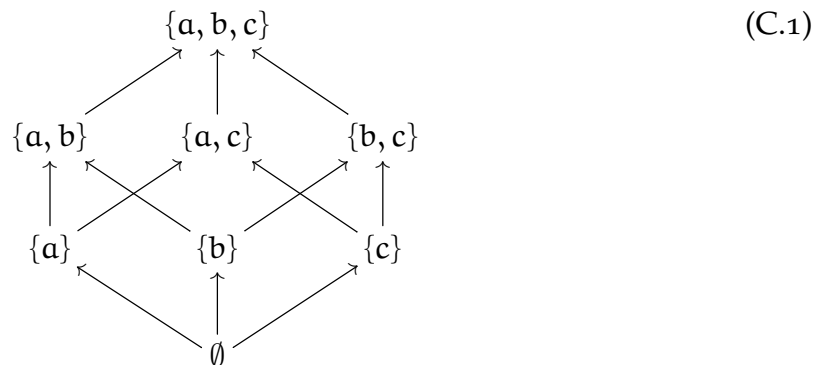
Intuitively, a *category* is a universe of homogenous mathematical structures (the *objects*) together with abstract means to “compare” them (the *morphisms*). One may think of categories as a generalisation of (pre-)orders, monoids and graphs together.

First Intuition: Preorders

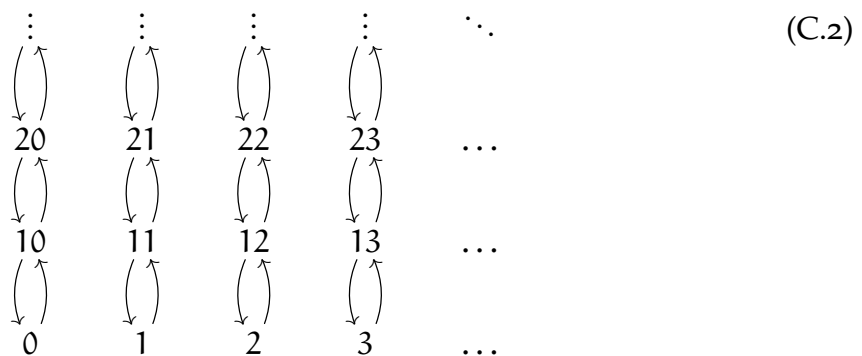
The mathematical concept behind “comparison” is given by orders. The most generic form of an order is a *preorder*, which is a relation $\preceq \subseteq X \times X$ on some set X satisfying the following conditions:

$$\begin{aligned} \forall x \in X : x \preceq x & \qquad \qquad \qquad \text{(Reflexivity)} \\ \forall x, y, z \in X : x \preceq y \wedge y \preceq z \implies x \preceq z & \qquad \text{(Transitivity)} \end{aligned}$$

The tuple (X, \preceq) is called a *preordered set*.



(C.1) contains a visualisation of a preordered set, the subset lattice $(\wp X, \subseteq)$ of the three-element set $X := \{a, b, c\}$. The visualisation is given in the form of a directed graph, i.e. every element of $\wp X$ becomes a vertex and an edge is drawn if there is a subset relation $X' \subseteq X''$ for two sets $X', X'' \in \wp X$. Note, however, that not all edges have to be drawn, e.g. there is no direct edge between \emptyset and $\{a, b\}$ and no loop on $\{a\}$. This is because, we know that (C.1) depicts a preorder and therefore (Reflexivity) and (Transitivity) hold.



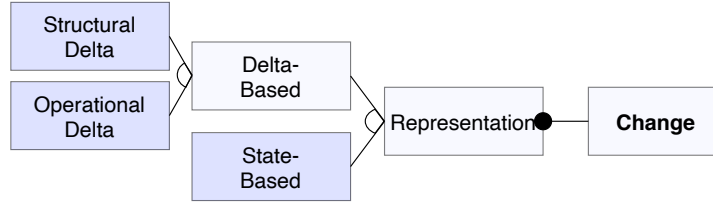


Fig. C.1: Feature model excerpt (of Fig. 4.3)

(C.2) illustrates a different (and this time infinite) preorder $(\mathbb{N}at, \equiv_{\text{mod } 10})$, i.e. the set of all natural numbers together with the modulus relation. Two natural numbers n and m are congruent modulo 10, written $n \equiv m(\text{mod } 10)$, if there is an integer k such that $n = a * 10 + m$. Comparing (C.2) to (C.1), there is a new phenomenon: There are cycles. This is due to the fact that $(\mathbb{N}at, \equiv_{\text{mod } 10})$ is an *equivalence relation*. An equivalence relation is a preorder that additionally is *symmetric*:

$$\forall x, y \in X : x \preceq y \implies y \preceq x \quad (\text{Symmetry})$$

A programming related example of an equivalence relation is given by the `equals()` method in `Object`, the abstract superclass of classes in Java. By providing a custom implementation for this method in a concrete subclass, the programmer implicitly defines an equivalence relation and hence should ensure that his implementations conforms to the properties of ([Reflexivity](#)), ([Transitivity](#)), and ([Symmetry](#)).

What is commonly understood under the term order is called *partial* or *total* order in mathematics. A partial order is an *anti-symmetric* preorder:

$$\forall x, y \in X : x \preceq y \wedge y \preceq x \implies x = y \quad (\text{Anti-Symmetry})$$

Graphically speaking, a partial-order only allow those cycles that arise from ([Reflexivity](#)). There are many examples of partial orders in software engineering, e.g. the inheritance-relationship among classes in UML class digrams, the commit-history of a git-repository, or the package-structure in a Java-program. Also this thesis comprises examples of partial orders. For instance, the feature model in Chap. 4 represents a partial order.

Consider Fig. C.1, which represent an excerpt of Fig. 4.3 and, read left-to-right, can formally be interpreted by a preordered set (ignoring the different types of feature relationships such as *mandatory*, *optional*, *or-group*, *xor-group*).

A total order is partial order that additionally is *strongly-connected*, i.e. there are no “incomparable” pairs of elements:

$$\forall x, y \in X : x \preceq y \vee y \preceq x \quad (\text{Strongly connected})$$

A well known total order is given by the *natural numbers* $\mathbf{Nat} = (\mathbb{N}at, \leq)$:

$$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow \dots$$

or *truth values* denoted by $\mathbf{2} := (\{\perp, \top\}, \{(\perp, \perp), (\top, \top), (\perp, \top)\})$, which is visualised as

follows:

$$\perp \longrightarrow \top \tag{C.3}$$

In programming languages, the latter is usually called Boolean and its two elements are called true and false.

Finally, instead of expressing \preceq as a relation, I could instead have defined as a function¹ $\text{Hom}_{\preceq} : X \times X \rightarrow \mathbf{2}$. The relationship between Hom_{\preceq} and \preceq is captured by the following equivalence:

$$\forall x, y \in X : x \preceq y \Leftrightarrow \text{Hom}_{\preceq}(x, y) = \top \tag{C.4}$$

This alternative approach is important to keep in mind for later when I will provide the final definition of categories, which will also explain the choice of the name Hom.

Second Intuition: Monoids

A *monoid* is a triple (M, ϵ, \otimes) where M is the *carrier set*, $\epsilon \in M$ is called the *neutral element* and $\otimes : M \times M \rightarrow M$ is a function, the *monoid multiplication* satisfying the following conditions:

$$\forall m \in M : m \otimes \epsilon = m = \epsilon \otimes m \tag{Identity Element}$$

$$\forall m, n, k \in M : (m \otimes n) \otimes k = m \otimes (n \otimes k) \tag{Associativity}$$

There are many examples of monoids in Mathematics, e.g. natural numbers with additions $(\text{Nat}, 0, +)$ or natural numbers with multiplication $(\text{Nat}, *, 1)$. The most graphic example of monoids for Software Engineers are *lists* over some data type T , e.g. `List<T>` in Java: The instances of T provide the material for the carrier set, the empty list represents the neutral element, and list concatenation is the monoid multiplication. It is easy to verify that this construct satisfies the monoid axioms ([Identity Element](#)) and ([Associativity](#)),

Third Intuition: Graphs

Graphs are utilised to visualise orders in Sec. C.2 but they have many more application areas and there are different “types” of graphs, see Sec. 5.1.1. For this thesis, the most important concept are *directed multigraphs* $G := (V, E, o, t)$ where V is a set of *vertices*, E is a set of *edges*, $\text{owner} : E \rightarrow V$ is a function that assigns the *owner* vertex to an edge (origin), and $\text{target} : E \rightarrow V$ is another function that assigns the *target* vertex to an edge.

Graphs are encountered almost everywhere in Computer Science and software engineering. One popular example are class diagrams. For example consider Fig. C.2, which is actually an excerpt of Fig. 6.3. This “graph” contains 6 vertices and 7 edges: Every class (4 elements) and data type (2 elements) is interpreted as a vertex and every reference (3 elements) and attribute (4 elements) is interpreted as an edge.

¹In typical abuse of notation, I am using $\mathbf{2}$ to refer to both the respective preorder as well as the underlying set.

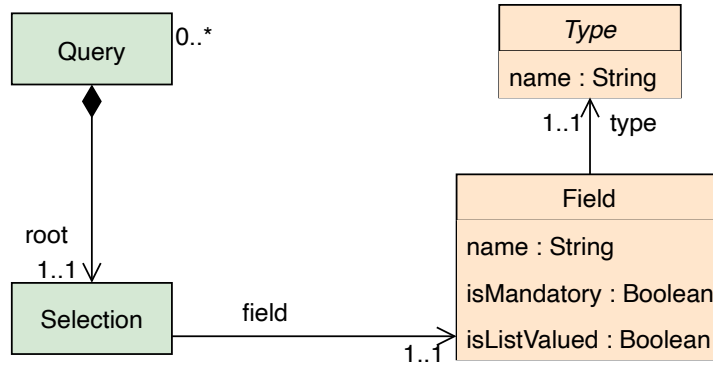


Fig. C.2: Class diagram excerpt (of Fig. 6.3)

A common phenomenon of object-oriented programming is navigation on such class diagram graphs. For instance, a programmer may write down the following path

`root.field.type.name`

inside a method of the class `Query`. Formally, this expression can be interpreted as a list of edge names, i.e. there is an underlying “path monoid” where monoid multiplication is denoted by “.” and the neutral element is given by “this”. There is an important intricacy to this monoid, its multiplication operator is only *partial*. This means that one can only concatenate $e . e'$ two edge names $e, e' \in E$ if they are incident, i.e. $t(e) = o(e')$. Thus, writing “this.root.type” starting in `Query` would not be allowed.

Another example of this idea is the network of updates in a model space, see Sec. 3.3.1: Models represent vertices, updates represent edges, idle updates are the empty paths, and composed updates are general paths.

A category can actually be described via paths on directed multigraphs by introducing some special terminology: Vertices are renamed to *objects*, paths over edges are renamed to *morphisms*, the function $o : E \rightarrow V$ is renamed to *domain*, and the function $t : E \rightarrow V$ is renamed to *codomain*.

Definition and Examples

Definition C.1 Category

A category \mathbb{C} consists of the following:

- A class of *objects* $|\mathbb{C}|$.
- For every pair of objects $A, B \in |\mathbb{C}|$, there is a set of *morphisms* called a *hom-set* and denoted $\mathbb{C}(A, B)$. For every member $f \in \mathbb{C}(A, B)$, $A = \text{dom}(f)$ is called the *domain* of f and $B = \text{cod}(f)$ is called the *codomain* of f . The class of *all* morphisms (i.e. union of all hom-sets) is denoted by \mathbb{C}^\rightarrow .
- For every object $A \in |\mathbb{C}|$ there is a unique *identity* morphism $\text{id}_A \in \mathbb{C}(A, A)$.
- For every triple of objects $A, B, C \in |\mathbb{C}|$ and morphisms $f \in \mathbb{C}(A, B)$ and $g \in \mathbb{C}(B, C)$, there is a *composite* $g \circ f \in \mathbb{C}(A, C)$. Thus, *composition* can be interpreted as binary partial operation

$$_ \circ _ : \mathbb{C}^\rightarrow \times \mathbb{C}^\rightarrow \rightharpoonup \mathbb{C}^\rightarrow$$

that is defined iff its arguments are incident ($\text{cod}(f) = \text{dom}(g)$).

Furthermore, \mathbb{C} has to satisfy the following laws:

- Composition \circ is *neutral* w.r.t. identities, i.e. for all $f \in \mathbb{C}(A, B)$:

$$\text{id}_B \circ f = f = f \circ \text{id}_A \quad (\text{Identity Laws})$$

- Composition \circ is *associative*, i.e. for all $f \in \mathbb{C}(A, B)$, $g \in \mathbb{C}(B, C)$, and $h \in \mathbb{C}(C, D)$:

$$(h \circ g) \circ f = h \circ (g \circ f) \quad (\text{Associativity Laws})$$

A category where the class of objects is a *set* is called *small*.

Coming back to the introductory examples in Sec. C.2, every preorder is a category: The carrier set provides the objects and the relation provides the morphisms, (**Reflexivity**) guarantees that the (**Identity Laws**) hold, and (**Transitivity**) guarantees that the (**Associativity Laws**) hold. Intuitively speaking, a preorder is category where the hom-set for each pair of object at most contains a single element, e.g. named \top (see the concluding remark of Sec. C.2). Thus, there was no need to give a name to the edges in (C.1) and (C.2).

Likewise, every monoid (M, ϵ, \otimes) is a category \mathbb{C} . This time the class of objects is a singleton set. The name of the element in this singleton does not matter, one may call it $*$. The carrier set M of the monoid is interpreted as the homset $\mathbb{C}(*, *)$. The neutral element is the identity of $*$ and composition \circ is defined through \otimes . The resemblance between (**Identity Element**) and (**Identity Laws**) as well as (**Associativity**) and (**Associativity Laws**) is obvious.

Arguably, the most important example of a category is the category **Set** (the assembly language of mathematics).

Fact 20 Category Set

The category of *sets and functions* **Set** comprises:

- The class^a of *all* sets as objects,
- For a pair of sets $A, B \in |\mathbf{Set}|$, their hom-set $\mathbf{Set}(A, B)$ is the set of all functions $f : A \rightarrow B$.
- The identity for a set A is given by the *identical* mapping on this set:

$$\text{id}_A := \begin{cases} A \rightarrow A \\ x \mapsto x \end{cases}$$

- The composition of two functions $f : A \rightarrow B$ and $g : B \rightarrow C$ with incident domain/codomain is defined by *function composition*:

$$g \circ f := \begin{cases} A \rightarrow C \\ a \mapsto g(f(a)) \end{cases}$$

^aSee Russel's paradox [485]

The notion of composition as function composition is the reason for category theory sometimes being described as the “algebra of functions” [478]. One may also consider other categories built over the collection of all sets. They differ in the definition of their morphisms and consequentially in the definition of identities and composition.

Example C.1 Category of Sets and Relations **Rel**

The category of *sets and relations* **Rel** comprises:

- The class of *all* sets as objects,
- For a pair of sets $A, B \in |\mathbf{Rel}|$, their hom-set $\mathbf{Rel}(A, B)$ is the set of all relations $R \subseteq A \times B$.
- The identity for a set A is given by the *diagonal relation* on this set:

$$\Delta_A := \{(a, a) \mid a \in A\}$$

- The composition of two relations $R \in \mathbf{Rel}(A, B)$ and $Q \in \mathbf{Rel}(B, C)$ is defined by *relation composition*:

$$Q \circ R := \{(a, c) \mid \exists b \in B : (a, b) \in R \wedge (b, c) \in Q\}$$

Example C.2 Category of Sets and Inclusions **Incl**

The category of *sets and inclusions* **Incl** is the partial order given by the *subset* relation

$$A \subseteq B \Leftrightarrow \forall a \in A \implies a \in B$$

over the class of all sets.

Example C.3 **Nat**

Nat denotes a category given by the total order on the set of natural numbers.

Categories that are based on the collection of all sets are very big. Hence, I want give two examples of very small categories.

Example C.4 The empty category **0**

The *empty category* **0** has an empty set of objects and hence an empty set as morphisms. Identities and composition are totally undefined.

Example C.5 The terminal category **1**

The *terminal category* **1** has exactly one object \bullet and exactly one morphism, the identity on \bullet .

Finally, one can construct categories from known ones.

Definition C.2 Opposite Category \mathbb{C}^{op}

Let \mathbb{C} be a category. There is an opposite category \mathbb{C}^{op} , where

- the class of objects is the same, i.e. $|\mathbb{C}| = |\mathbb{C}^{\text{op}}|$,
- the morphisms are “inverted”, i.e. $f \in \mathbb{C}(A, B) \Leftrightarrow f \in \mathbb{C}^{\text{op}}(B, A)$.
- identities are the same, and
- composition is inherited from \mathbb{C} in the opposite way:

$$g \circ f \in \mathbb{C}^{\rightarrow} \Leftrightarrow f \circ g \in \mathbb{C}^{\text{op}\rightarrow}$$

Definition C.3 Product Category $\mathbb{C} \times \mathbb{D}$

Let \mathbb{C} and \mathbb{D} be categories. Their product category $\mathbb{C} \times \mathbb{D}$ comprises

- The cartesian product $|\mathbb{C}| \times |\mathbb{D}|$ as objects,
- For $(C, D), (C', D') \in |\mathbb{C} \times \mathbb{D}|$, the hom-set $\mathbb{C} \times \mathbb{D}((C, D), (C', D')) = \mathbb{C}(C, C') \times \mathbb{D}(D, D')$ is given by the cartesian product of hom-sets from \mathbb{C} and \mathbb{D} ,
- identities are pairs of identities from \mathbb{C} and \mathbb{D} , i.e. $\text{id}_{(C, D)} = (\text{id}_C, \text{id}_D)$ for $C \in |\mathbb{C}|$ and $D \in |\mathbb{D}|$, and
- composition is defined category-wise, i.e. $(f, g) \in \mathbb{C} \times \mathbb{D}((C, D), (C', D'))$ and $(f', g') \in \mathbb{C} \times \mathbb{D}((C', D'), (C'', D''))$ their composite is defined as follows:

$$(f', g') \circ (f, g) = (f' \circ f, g' \circ g)$$

Intuitively, the “constructions” in Def. C.2 and Def. C.3 are described as “inverting the arrows” and “composing two categories in parallel”, respectively. The former ascribes to a categorical phenomenon called *Duality*: One may say that a construction or a proof can be carried out “dually”. This means that it happens in the same way but the arrows point in the other direction.

Isomorphism

Due to the abstract nature of categories, it is generally not possible to check if two objects represent the same thing because one can generally not have a look *inside* the objects. However, one can compare them via the morphisms that go in and out. If two objects are comparable by invertible morphisms, they are called *isomorphic*, i.e. “identical modulo some internal restructuring”. In **Set**, this translates to “identical modulo *renaming*”.

Isomorphisms are closely related to equivalence relations, see the cycles in (C.2).

Definition C.4 Isomorphism

Let \mathbb{C} be a category and $A, B \in |\mathbb{C}|$ two objects in this category. A and B are called *isomorphic*, written $A \approx B$, if and only if there exist two morphisms $i : A \rightarrow B \in \mathbb{C}^{\rightarrow}$ and $i^{-1} : B \rightarrow A \in \mathbb{C}^{\rightarrow}$ such that $\text{id}_A = i^{-1} \circ i$ and $\text{id}_B = i \circ i^{-1}$. The morphisms, i and i^{-1} are called *isomorphisms*.

The following Fact follows immediately from Def. C.1 and Def. C.4.

Fact 21 Equivalence Relation \approx

Let \mathbb{C} be a category. The relation $\approx \subseteq |\mathbb{C}| \times |\mathbb{C}|$ induced by the concept of isomorphisms is an equivalence relation. An equivalence class $[A]_{\approx} := \{A' \mid A \approx A'\}$ is called an *abstract object*.

C.3 Important Concepts

In the remainder of this appendix chapter, I will enumerate several categorical concepts that are used in this thesis. I will provide a definition and intuitive motivation for each of them. Yet, these presentations are rather brief. For detailed treatment, I refer to the textbooks mentioned in the introduction of this appendix section.

Functors and Adjunctions

In the previous section, I introduced several examples of categories. Some of them smaller, some of them bigger. Moreover, some of them are apparently contained in each other. For instance, the category **Set** is apparently contained in **Rel** since every function can be seen as a special relation and composition of functions in **Rel** is function composition. Hence, a concept for “comparing” categories is needed, which is provided by the notion of *functors*.

Definition C.5 **Functor**

Let \mathbb{C} and \mathbb{D} be two categories. A functor $F : \mathbb{C} \rightarrow \mathbb{D}$ comprises,

- an object mapping, i.e. for every object $A \in |\mathbb{C}|$ in the source category, F assigns an object $F(A) \in |\mathbb{D}|$ in the target category,
- and a morphism mapping, i.e. for every morphism $f : A \rightarrow B$, F assigns a morphism $F(f) : F(A) \rightarrow F(B)$ in the target category,

such that

- identities are mapped to identities, i.e. for all $A \in |\mathbb{C}|$: $F(\text{id}_A) = \text{id}_{F(A)}$.
- and composition is preserved, i.e. for all $f \in \mathbb{C}(A, B)$ and $g \in \mathbb{C}(B, C)$: $F(g \circ f) = F(g) \circ F(f)$ ^a.

F is called an *embedding*, written $\mathbb{C} \sqsubseteq \mathbb{D}$, if and only if it is injective on objects of \mathbb{C} and injective on $\mathbb{C}(A, B)$ for all $A, B \in |\mathbb{C}|$.

Furthermore, a functor F is called *contravariant* if either the domain or codomain category is an opposite category, e.g. $F : \mathbb{C} \rightarrow \mathbb{D}^{\text{op}}$.

^aNote, that the composition \circ in the left hand side of the identity is the composition of \mathbb{C} , while the composition in the right hand side is the composition of \mathbb{D} .

In this thesis, I will often write the definition of a functor in a more compact way:

$$F := \begin{cases} \mathbb{C} \rightarrow \mathbb{D} \\ A \in |\mathbb{C}| \mapsto F(A) \in |\mathbb{D}| \\ f : A \rightarrow B \in \mathbb{C} \mapsto F(f) : F(A) \rightarrow F(B) \in \mathbb{D} \end{cases}$$

If the object mapping is trivial, the middle line is omitted. Let me now give an example that let us compare **Incl** with **Set**, which turns out to be an embedding.

Example C.6 Embedding Incl into Set

There is an embedding functor \square :

$$\square := \begin{cases} \mathbf{Incl} \rightarrow \mathbf{Set} \\ A \subseteq B \in \mathbf{Incl}^{\rightarrow} \mapsto \text{incl} : A \hookrightarrow B \in \mathbf{Set}^{\rightarrow} \end{cases}$$

where $\text{incl} : A \hookrightarrow B$ is a special injective functions (monomorphism):

$$\text{incl} : \begin{cases} A \hookrightarrow B \\ a \in A \mapsto a \in B \end{cases}$$

It is also possible to compare **Incl** with **Nat** when restricting **Incl** to finite sets,

Example C.7 Cardinality as a functor

There is a cardinality functor $|_|$:

$$|_| := \begin{cases} \mathbf{Incl} \rightarrow \mathbf{Nat} \\ A \mapsto |A| \\ A \subseteq B \mapsto |A| \leq |B| \end{cases}$$

that assigns the *cardinality* (i.e. number of elements) to each set.

The functor concept allows to compare categories at a large scale.

Fact 22 Category of Categories: \mathbf{Cat} and \mathbf{CAT}

The category of *small categories and functors* \mathbf{Cat} comprises

- The class of *all* small categories as objects,
- For a pair of small categories $\mathbb{C}, \mathbb{D} \in |\mathbf{Cat}|$, their hom-set $\mathbf{Cat}(\mathbb{C}, \mathbb{D})$ is the set of all functors $F : \mathbb{C} \rightarrow \mathbb{D}$.
- The identity for a category \mathbb{C} is given by the *identity* Functor:

$$\text{id}_{\mathbb{C}} := \begin{cases} \mathbb{C} \rightarrow \mathbb{C} \\ A \mapsto A \\ f : A \rightarrow B \mapsto f : A \rightarrow B \end{cases}$$

- The composition of two functors $F : \mathbb{C} \rightarrow \mathbb{D}$ and $G : \mathbb{D} \rightarrow \mathbb{E}$ is defined by component-wise composition:

$$g \circ f := \begin{cases} \mathbb{C} \rightarrow \mathbb{E} \\ A \mapsto G(F(A)) \\ f : A \rightarrow B \mapsto G(F(f)) : G(F(A)) \rightarrow G(F(B)) \end{cases}$$

One may also consider to replace the class of all small categories with the class of *all* categories. However, the resulting construct (quasi-category) \mathbf{CAT} cannot be called category (“Russell’s paradox”).

Furthermore, one may now be interested to “compare” functors. The respective concept for this are *natural transformations*:

Definition C.6 Natural Transformation

Let $F : \mathbb{C} \rightarrow \mathbb{D}$ and $G : \mathbb{C} \rightarrow \mathbb{D}$ be two functors between the same categories. A natural transformation $\alpha : F \Rightarrow G$ is given by a $|\mathbb{C}|$ -indexed family of \mathbb{D} -morphisms $(\alpha_A : F(A) \rightarrow G(A))_{A \in |\mathbb{C}|}$, such that for every $f \in \mathbb{C}(A, B)$ the following diagram commutes:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \alpha_A \downarrow & & \downarrow \alpha_B \\ G(A) & \xrightarrow{G(f)} & G(B) \end{array} \quad (\text{C.5})$$

The diagram in (C.5) is called a *naturality square*. When every \mathbb{D} -morphisms in a natural transformation α is an isomorphism, α is called a *natural isomorphism*.

Functors and natural transformations can be organised as category as well:

Fact 23 Functor Category

Let \mathbb{C} and \mathbb{D} be small categories. The *functor category* $\mathbb{D}^{\mathbb{C}}$ comprises

- The class of all functors from \mathbb{C} to \mathbb{D} as objects,
- For a pair of functors $F, G : \mathbb{C} \rightarrow \mathbb{D}$, their hom-set $\mathbb{D}^{\mathbb{C}}(F, G)$ is the class of all natural transformations $\alpha : F \Rightarrow G$.
- The identity for a functor F is given by the *natural identities*:

$$\text{id}_F := (\text{id}_{F(A)})_{A \in |\mathbb{C}|}$$

- The composition of two natural transformations $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$ is defined object-wise composition of morphisms in \mathbb{D} :

$$\beta \circ \alpha := (\beta_A \circ \alpha_A)_{A \in |\mathbb{C}|}$$

The concept of functor categories (Fact 23) is used to construct new categories from known ones. Often, these constructions are based on **Set**. Hence, functors $G : \mathbb{B} \rightarrow \mathbf{Set}$ going into set will play an important role in this thesis, see Def. 5.1. For example, directed multigraphs can be described via such a construction.

With functors and natural transformation, we have all tools at hand to check whether two categories (i.e. classes of mathematical structures) are essentially “the same” modulo isomorphism. The latter is called an *equivalence*.

Definition C.7 Equivalence of Categories

Let \mathbb{C} and \mathbb{D} be two categories. They are said to be *equivalent*, written $\mathbb{C} \cong \mathbb{D}$, if there exists a pair of functors $R : \mathbb{C} \rightarrow \mathbb{D}$ and $L : \mathbb{D} \rightarrow \mathbb{C}$ together with two natural isomorphisms $\approx_{\mathbb{C}} : L \circ R \Rightarrow \text{id}_{\mathbb{C}}$ and $\approx_{\mathbb{D}} : \text{id}_{\mathbb{D}} \Rightarrow R \circ L$. If these families of isomorphisms are actually identities, then \mathbb{C} and \mathbb{D} are said to be *isomorphic*.

In category theory there is also a weaker notion than equivalence, called *adjunction*². Intuitively speaking it means that two classes of structures are equivalent modulo some “free construction” that can be universally applied. An example for such a construction is the free monoid A^* (lists) over a set A .

²It is sometimes noted that this notion is the actual reason category theory was invented

Initial and Terminal Objects

Final and initial objects are defined over an empty diagram and therefore relate uniquely to *all* objects in a category.

Definition C.10 Initial and Terminal Objects

An object $1 \in \mathbb{C}$ is said to be *terminal* iff. for all $X \in |\mathbb{C}|$ there exists a unique morphism $1_X : X \rightarrow 1$. Dually, an object $0 \in \mathbb{C}$ is said to be *initial* iff for all $X \in |\mathbb{C}|$ there exists a unique morphism $0_X : 0 \rightarrow X$.

$$\begin{array}{ccc} X & & 0 \\ \vdots \downarrow 1_X & & \downarrow 0_X \\ 1 & & X \end{array}$$

Fact 24 Terminal object in Set

A one-element set, e.g. $1 = \{\bullet\}$ is a terminal object in **Set**.

Fact 25 Initial object in Set

The initial object in **Set** is the empty set $0 = \emptyset$.

Coproducts

Coproducts a.k.a. *sums* provide means to collect a set of objects and work with them uniformly, similar to type abstraction in programming.

Definition C.11 Binary Coproduct

Let \mathbb{C} be a category and $A, B \in \mathbb{C}$ be objects. A binary *coproduct* of A and B is given by an object $A + B$ and two coproduct *injection* morphisms $\iota_A : A \rightarrow A + B$ and $\iota_B : B \rightarrow A + B$ such that for all pairs of \mathbb{C} -morphisms $f : A \rightarrow C$ and $g : B \rightarrow C$ with $C \in \mathbb{C}$ there exists a unique morphism $[f; g] : A + B \rightarrow C$ such that $[f; g] \circ \iota_A = f$ and $[f; g] \circ \iota_B = g$, visualised in (C.7):

$$\begin{array}{ccc} & C & \\ & \uparrow [f; g] & \\ f & A + B & g \\ \downarrow \iota_A & & \downarrow \iota_B \\ A & & B \end{array} \quad (\text{C.7})$$

The mediating morphism $[f; g]$ acts like f and g via case distinction and allows to

extend Def. C.11 into a functor definition:

$$- + - := \begin{cases} \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C} \\ (A, A') \mapsto A + A' \\ (f : A \rightarrow B, g : A' \rightarrow B') \mapsto [\iota_B \circ f, \iota_{B'} \circ g] : A + A' \rightarrow B + B' \end{cases}$$

A multi-ary coproduct \coprod is given by multiple applications of the binary coproduct functor, because the latter are associative $((A_1 + A_2) + A_3 \cong A_1 + (A_2 + A_3))$ and commutative $(A_1 + A_2 \cong A_2 + A_1)$ up to isomorphism. The (multi-ary) coproduct over an I-indexed family of \mathbb{C} -objects $(A_i)_{i \in I}$ is denoted $(\coprod_{i \in I} A_i, (\iota_i : A_i \rightarrow \coprod_{i \in I} A_i)_{i \in I})$ and the mediating morphism for a family of morphisms $(f_i : A_i \rightarrow C)_{i \in I}$ by $\coprod f_i : \coprod_{i \in I} A_i \rightarrow C$.

Fact 26 Coproducts in Set

Set has all coproducts. A binary coproduct in **Set** is given by disjoint union $A \sqcup B := \{(i, x) \mid (x \in A \wedge i = 1) \vee (x \in B \wedge i = 2)\}$ for A and B being sets. The initial object 0 in **Set** is the empty set \emptyset .

Lemma 27 Coproducts in Set^B [207]

Every functor category **Set^B** has coproducts due to the fact that **Set** has all coproducts and we can construct them pointwise.

Proof. Let F and G be two objects in **Set^B** and consider the family of diagrams in (C.8), which is indexed by $f : A \rightarrow B \in \mathbb{B}^{\rightarrow}$

$$\begin{array}{ccccc}
 & F(A) & & G(A) & \\
 & \downarrow & \searrow^{\iota_{F(A)}} & \swarrow_{\iota_{G(A)}} & \downarrow \\
 & F(A) & \rightarrow & F(A) + G(A) & \rightarrow & G(A) \\
 & \downarrow^{F(f)} & & \downarrow & & \downarrow^{G(f)} \\
 & F(B) & & F(B) + G(B) & & G(B) \\
 & \downarrow & \searrow^{\iota_{F(B)}} & \swarrow_{\iota_{G(B)}} & \downarrow \\
 & F(B) & \rightarrow & F(B) + G(B) & \rightarrow & G(B) \\
 & & & \downarrow & & \\
 & & & F(f) + G(f)! & & \\
 & & & \downarrow & & \\
 & & & F(B) + G(B) & &
 \end{array} \tag{C.8}$$

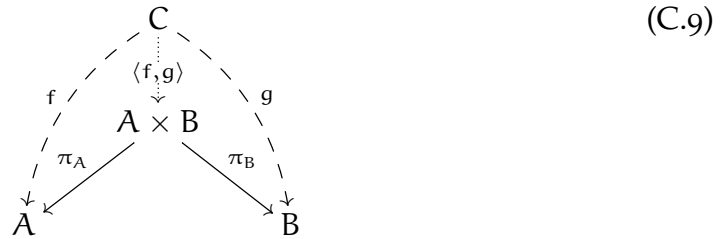
The coproduct of F and G for objects A, B is given by constructing the respective coproducts $F(A) + G(A)$ and $F(B) + G(B)$ in **Set**, the morphism mappings $(F + G)(f)$ (dotted line) arises uniquely from the universal property of coproducts. \square

Products

Products provide the means to express all possible ways of combining two objects and product types are part of every programming language where they are better known as “record types”.

Definition C.12 Binary Product

Let \mathbb{C} be a category and $A, B \in \mathbb{C}$ be objects. A binary *product* of A and B written $A \times B$ is given by a span $(A \times B, \pi_A : A \times B \rightarrow A, \pi_B : A \times B \rightarrow B)$ such that for all spans $(X, f : X \rightarrow A, g : X \rightarrow B)$ there is a unique morphism $(f, g) : X \rightarrow A \times B$ such that $\pi_A \circ (f, g) = f$ and $\pi_B \circ (f, g) = g$, visualised in (C.9)



The most well-known instance of products are cartesian products in set.

Fact 28 Products in Set

Set has all products. A binary product in **Set** is given by the cartesian product $A \times B := \{(a, b) \mid a \in A \wedge b \in B\}$ for A and B being sets.

Lemma 29 Products in Set^B

Every functor category **Set^B** has products due to the fact that **Set** has all products and one can construct them pointwise.

Proof. Dual to the proof in Lemma 29. □

Pullbacks

A pullback can be seen as the categorical version of an *inner join*: two structures A and B are combined where they coincide on a common structure C . Pullbacks are defined over a diagram called co-span, i.e. a pair of morphisms sharing the same codomain, and result in a span, i.e. a pair of morphisms sharing the same domain.

Definition C.13 Pullback

Let \mathbb{C} be category and a co-span of \mathbb{C} -morphisms $A \xrightarrow{a} C \xleftarrow{b} B$ be given. The pullback of a and b is given by the span $A \xleftarrow{\pi_A} A \times_{(a,b)} B \xrightarrow{\pi_B} B$ such that $a \circ \pi_A = b \circ \pi_B$ and for all pairs of \mathbb{C} -morphisms $f : D \rightarrow A$ and $g : D \rightarrow B$ such that $b \circ g = a \circ f$ and there exists a unique morphism $\langle f, g \rangle : D \rightarrow A \times_{(a,b)} B$ such that $\pi_A \circ \langle f, g \rangle = f$ and $\pi_B \circ \langle f, g \rangle = g$, visualized in (C.10):

$$\begin{array}{ccccc}
 D & & & & \\
 \swarrow \langle f, g \rangle & & & & \searrow g \\
 & A \times_{(a,b)} B & \xrightarrow{\pi_B} & B & \\
 \swarrow f & \downarrow \pi_A & \text{p. b.} & \downarrow \pi_B & \searrow b \\
 & A & \xrightarrow{a} & C &
 \end{array} \tag{C.10}$$

The morphisms π_A and π_B are sometimes called projections.

Throughout this thesis, I will regularly depict pullback diagrams. For a pullback diagram depicted below (which is highlighted by the small “ \lrcorner ” symbol in the upper left corner),

$$\begin{array}{ccc}
 D & \xrightarrow{g'} & A \\
 f' \downarrow & \lrcorner & \downarrow f \\
 B & \xrightarrow{g} & C
 \end{array}$$

the object D is called the “pullback object”. The morphisms (f, g, f', g') form a commutative square, written $g \circ f' = f \circ g'$. Moreover, f' is called “the pullback of f along g ” (analogously for g').

Fact 30 Pullback Composition and Decomposition

There is a well-known fact about composition and decomposition of pullbacks, which immediately follows from their universal property:

$$\begin{array}{ccccc}
 A & \xrightarrow{c} & B & \xrightarrow{a} & E \\
 d \downarrow & \text{(i)} & \downarrow b & \text{(ii)} & \downarrow f \\
 C & \xrightarrow{h} & D & \xrightarrow{g} & F
 \end{array} \tag{C.11}$$

Consider the diagram in (C.11), let (i) denote (b, h, d, c) , (ii) denote (f, g, b, a) , and (i) + (ii) denote $(f, g \circ h, d, b \circ c)$. Then, if (i) and (ii) are pullbacks, so is (i) + (ii). Likewise, if (i) + (ii) and (ii) are pullbacks, so is (i).

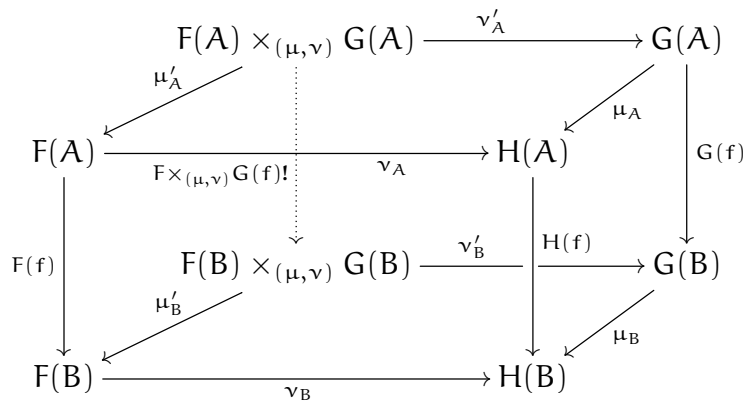
Fact 31 Pullbacks in Set

Set has pullbacks. Given two mappings $f : A \rightarrow C$ and $g : B \rightarrow C$ with same codomain the pullback $A \times_{(f,g)} B$ is given by the fibred product $A \times_{(f,g)} B := \{(a, b) \mid a \in A, b \in B, f(a) = g(b)\}$.

Lemma 32 Pullbacks in $\mathbf{Set}^{\mathbb{B}}$ [207]

Every functor category $\mathbf{Set}^{\mathbb{B}}$ has pullbacks due to the fact that **Set** has all pullbacks and we can construct them pointwise.

Proof. Let F, G and H be objects in $\mathbf{Set}^{\mathbb{B}}$ and $\nu : F \Rightarrow H$ and $\mu : G \Rightarrow H$ morphisms in $\mathbf{Set}^{\mathbb{B}}$. Consider the following cube for some $f : A \rightarrow B \in \mathbb{B}^{\rightarrow}$:



The pullback of μ and ν for objects A and B is given by constructing the respective pullbacks $F(A) \times_{(\mu, \nu)} G(A)$ and $F(B) \times_{(\mu, \nu)} G(B)$ in **Set** along (μ_A, ν_A) and (μ_B, ν_B) respectively, the morphism mapping $(F \times_{(\mu, \nu)} G)(f)$ (dotted line) arise uniquely from the universal property of the pullbacks in the bottom face of the cube. \square

Definition C.14 Monomorphism

A morphism $m : A \rightarrow B$ is called a *monomorphism* iff (id_A, id_A) is a pullback of the cospan (m, m) , see (C.12).

$$\begin{array}{ccc}
 A & \xrightarrow{id_A} & A \\
 id_A \downarrow & \text{p. b.} & \downarrow m \\
 A & \xrightarrow{m} & B
 \end{array} \tag{C.12}$$

In this case m has the left cancellation property, which is a consequence of the pullback property in (C.12):

$$\forall f, g : C \rightarrow A : \quad m \circ f = m \circ g \implies f = g \tag{C.13}$$

I sometimes highlight the special property of m by denoting it with a special arrow $m : A \rightrightarrows B$.

Fact 33 Monomorphism in Set

In **Set** the class of monomorphisms is exactly the class of *injective mappings*.

Fact 34 Pullbacks preserve Monos

If α is a monomorphism in the diagram of Def. C.13, then π_B is a monomorphism, as well.

Fact 35 Pullback projections are “jointly monic”

The morphisms π_A and π_B in (C.10) are jointly monic, i.e. for all $x : X \rightarrow A \times_{(\alpha, \beta)} B$ and $y : X \rightarrow A \times_{(\alpha, \beta)} B$:

$$\pi_A \circ x = \pi_A \circ y \wedge \pi_B \circ x = \pi_B \circ y \implies x = y$$

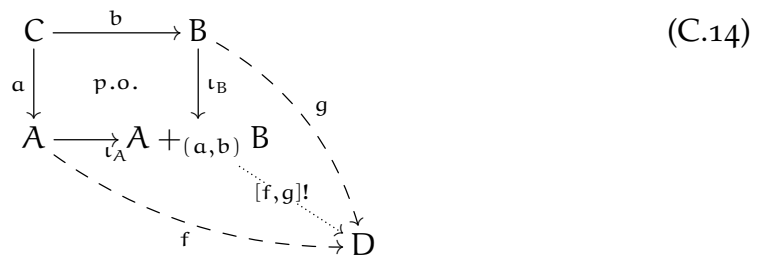
which immediately follows from the universal pullback property.

Pushouts

Pushouts are the “opposite” of pullbacks by providing a universal co-span completion. A pushout can intuitively be described as *gluing* of two structures at a defined interface.

Definition C.15 Pushout

Let \mathbb{C} be a category and a span of \mathbb{C} -morphisms $A \xleftarrow{\alpha} C \xrightarrow{\beta} B$ be given. The pushout of α and β is given by the co-span $A \xrightarrow{\iota_A} A +_{(\alpha, \beta)} B \xleftarrow{\iota_B} B$ such that $\iota_A \circ \alpha = \iota_B \circ \beta$ and for all pairs $f : A \rightarrow D$ and $g : B \rightarrow D$ there exists a unique morphism $[f; g] : A +_{(\alpha, \beta)} B \rightarrow D$ such that $[f; g] \circ \iota_A = f$ and $[f; g] \circ \iota_B = g$, visualized in (C.14):



The morphisms ι_A and ι_B are sometimes called *injections*.

Fact 36 Pushouts in Set

Set has all pushouts: Given two mappings $f : C \rightarrow A$ and $g : C \rightarrow B$ with same domain, consider a relation \sim on $A \sqcup B$, defined as follows (ι_A and ι_B are the embeddings into the disjoint union $A \sqcup B$)

$$a \sim b = \{f(c), g(c) \mid c \in C\}$$

and \equiv the least equivalence relation containing \sim , then the pushout of f and g is given by $A +_{(f,g)} B := (A \sqcup B) / \equiv$.

Lemma 37 Pushouts in $\mathbf{Set}^{\mathbb{B}}$

Every functor category $\mathbf{Set}^{\mathbb{B}}$ has pushouts due to the fact that **Set** has all pushouts and we can construct them pointwise.

Proof. Dual to the proof of Lemma 32. □

Fact 38 Pushout injections are “jointly surjective” in Set

Since, pushouts are dual to the notion of pullbacks, they show a property called *jointly epic*. An epimorphism is the dual notion of a monomorphism (Def. C.14) and in **Set** the epimorphisms correspond to the class of *surjective functions*. This means that in **Set**, pushout injections are jointly epic, which can formally be expressed as

$$\forall x \in A +_{(\alpha, \beta)} B : (\exists x' \in A : \iota_A(x') = x) \vee (\exists x' \in B : \iota_B(x') = x)$$

General Limits and Colimits

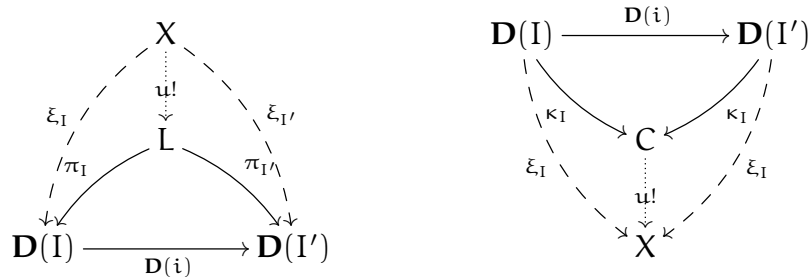
All of the concepts above can be generalised into the notion of *limits* and *colimits*, which are unique (co)cones. A cone is a distinguished object together with a family of *outgoing* morphisms that commute with all morphisms in the diagram (“sitting under”) and a cocone is a distinguished object together with a family of *incoming* morphisms commuting with all morphisms in the diagram (“sitting over”). Limits are generalized meets (“biggest cone below”), and colimits are generalized joins (“smallest cocone above”):

Definition C.16 (Co-)Cone

Let \mathbb{C} be a category and $\mathbf{D} : \mathbb{I} \rightarrow \mathbb{C}$ be a diagram. For an object $C \in |\mathbb{C}|$ we can define a functor $\Delta_C : \mathbb{I} \rightarrow \mathbb{C}$ that maps every object $I \in |\mathbb{I}|$ to C and every arrow $i : I \rightarrow I' \in \mathbb{I}(I, I')$ to id_C . A *cone* $(C, \pi : \Delta_C \Rightarrow \delta)$ is an object together with a natural transformation π , i.e. the selection induced by δ . A *cocone* $(C, \kappa : \delta \Rightarrow \Delta_C)$ is the dual notion.

Definition C.17 (Co-)Limit

Let \mathbb{C} be a category and $\mathbf{D} : \mathbb{I} \rightarrow \mathbb{C}$ be a diagram.
 A *limit* is a cone $(L, \pi : \Delta_L \Rightarrow \mathbf{D})$ such that for all cones $(X, \xi : \Delta_X \Rightarrow \mathbf{D})$ there exists a unique morphism $u! : X \rightarrow L$ such that $\pi_I \circ u! = \xi_I$ for all $I \in \mathbb{I}$. The components of the natural transformation π are called *projections*.
 A *colimit* is a cocone $(C, \kappa : \mathbf{D} \Rightarrow \Delta_C)$ such that for all cocones $(X, \xi : \mathbf{D} \Rightarrow \Delta_X)$ there exists a unique morphism $u! : C \rightarrow X$ such that $u! \circ \kappa_I = \xi_I$ for all $I \in \mathbb{I}$. The components of the natural transformation κ are called *injections*.
 Both notions are visualized in the diagrams below.



A functor $F : \mathbb{C} \rightarrow \mathbb{D}$ is said to preserve a limit $(L, \pi : \Delta_L \Rightarrow \mathbf{D})$ (respectively colimit) if $(F(L), F(\pi) : \Delta_L \Rightarrow F \circ \mathbf{D})$ is a limit in \mathbb{D} . The following important fact is found in every textbook about category theory (e.g. [326]) relates Def. C.17 and Def. C.8.

Fact 39

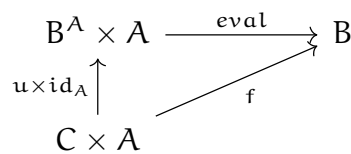
If $L \dashv R$ are two adjoint functors, then L preserves colimits and R preserves limits.

Exponentials and Cartesian Closedness

Exponentials can intuitively be described as a way to “objectify” morphisms in a category. The most well-known example are function types such as the set of functions between two domains.

Definition C.18 Exponential

Let \mathbb{C} be a category with binary products. An *exponential object* is given by an object $B^A \in |\mathbb{C}|$ equipped with a morphism $eval : B^A \times A \rightarrow B$ such that for every object $C \times A$ and morphism $f : C \times A \rightarrow B$ there exists a unique morphism $u : C \rightarrow B^A$ such that the following diagram commutes.



The object B^A is sometimes called an internal hom, i.e. the “objectification” of the hom-set $\mathbb{C}(A, B)$. The unique morphism u in Def. C.18 corresponds to the notion of “currying” in programming languages.

Definition C.19 Cartesian closed category

A category \mathbb{C} is called *cartesian closed* if and only if

- it has terminal objects,
- it has all binary products, and
- it has all exponentials.

Cartesian Closed Categories (CCCs) represent a model for the typed λ -calculus [306]. The following fact can easily be proven in every CCC and the two opposite isomorphisms in (C.15) correspond to “currying” and “uncurrying”.

Fact 40 “Currying” and “Uncurrying”

In every cartesian closed category \mathbb{C} there is an isomorphism:

$$A^{B \times C} \approx (A^B)^C \tag{C.15}$$

for $A, B, C \in |\mathbb{C}|$.

Fact 41 Cartesian Closed Categories

The categories **Set**, **Set^{bb}** (see [207]) and **Cat** (see [3, 27.3 (e)]) are cartesian closed.

Subobject and Partial Arrow Classifiers

Subobject classifiers are an essential ingredient for doing *logic* in a category. Predicates, which play a central role in logic, are closely related to subsets. Simultaneously, monomorphisms are representatives of subsets. Subobject classifiers are special morphisms that relate to monomorphism in a unique way.

Definition C.20 Subobject Classifier

Let \mathbb{C} be a category with terminal objects and pullbacks. A subobject classifier is given by a morphism $\top : 1 \rightarrow \Omega$ such that for every monomorphism $a : A \rightarrow X$ there is a unique morphism $\chi_A : X \rightarrow \Omega$ such that the following diagram is a pullback:

$$\begin{array}{ccc}
 A & \xrightarrow{!} & 1 \\
 a \downarrow & & \downarrow \top \\
 X & \xrightarrow{\chi_A} & \Omega
 \end{array}$$

Fact 42 **Subject Classifier in Set and $\mathbf{Set}^{\mathbb{B}}$**

Set has subobject classifiers: Every two-element set can serve as a subobject classifier. Also, $\mathbf{Set}^{\mathbb{B}}$ has subobject classifiers. However, their construction is a bit more involved, see [207].

Partial Arrow Classifiers are an implication of subobject classifiers, which play an important role in Chap. 5. This name comes from the fact that a span, which contains one monic morphism can be interpreted as a representative of a partial morphism, see Sec. 5.1.3.2.

Definition C.21 **Partial Arrow Classifier**

Let B be an object in \mathcal{C} . A partial arrow classifier for B is given by a monomorphism $\eta_B : B \rightarrow \mathcal{L}B$ such that for a partial morphism span $[m, f] : A \rightharpoonup B$ there exists a unique morphism $\overline{[m, f]} : A \rightarrow \mathcal{L}B$ (the *totalization*) such that the resulting square (C.16) is a pullback:

$$\begin{array}{ccc}
 X & \xrightarrow{m} & A & & (C.16) \\
 \downarrow f & & \downarrow \overline{[m, f]} & \text{p.b.} & \\
 B & \xrightarrow{\eta_B} & \mathcal{L}B & &
 \end{array}$$

Fact 43 **Partial Arrow Classifiers in Set**

Set and $\mathbf{Set}^{\mathbb{B}}$ for a small category \mathbb{B} have partial arrow classifiers. In **Set**, \mathcal{L} adds a new \perp -element to every set and turns a partial function into a total function by mapping all non-mapped elements to \perp . In $\mathbf{Set}^{\mathbb{B}}$, the construction becomes more involved, see [207, pp.202-210].

Fact 44 Partial Arrow Classifiers form Adjunction

In a category with partial arrow classifiers, \mathcal{L} extends to a functor, which follows directly from Def. C.21 and composition of partial morphisms via pullbacks (Def. 5.11). The resulting functor \mathcal{L} is right adjoint to Γ and defined as follows:

$$\mathcal{L} := \begin{cases} \text{Par}(\mathbb{C}) \rightarrow \mathbb{C} \\ A \mapsto \mathcal{L}A \\ \langle m, f \rangle : A \rightarrow B \mapsto \overline{\langle \eta_A \circ m, f \rangle} : \mathcal{L}A \rightarrow \mathcal{L}B \end{cases} \quad (\text{C.17})$$

where the morphism-mapping $\overline{\langle \eta_A \circ m, f \rangle}$ is explained in further detail by the diagram in (C.18):

$$\begin{array}{ccccc} X & \xrightarrow{m} & A & \xrightarrow{id_A} & A & & (\text{C.18}) \\ id_X \parallel & & id_A \parallel & & \downarrow \eta_A & & \\ X & \xrightarrow{m} & A & \xrightarrow{\eta_A} & \mathcal{L}A & & \\ f \downarrow & & \downarrow \overline{\langle m, f \rangle} & & \downarrow \overline{\langle \eta_A \circ m, f \rangle} & & \\ B & \xrightarrow{\eta_B} & \mathcal{L}B & \xrightarrow{id_{\mathcal{L}B}} & \mathcal{L}B & & \end{array}$$

The underlying co-free construction of the adjunction $\Gamma \dashv \mathcal{L}$ is shown in (C.19).

$$\begin{array}{ccc} \text{Par}_{\mathcal{M}}(\mathbb{C}) & & \mathbb{C} & (\text{C.19}) \\ & & \vdots & \\ \begin{array}{ccc} B & \xrightarrow{\forall \langle m, f \rangle} & \Gamma_{\mathcal{M}} A \\ \varepsilon_B \uparrow & \swarrow \Gamma_{\mathcal{M}}(\overline{\langle m, f \rangle}) & \\ \Gamma_{\mathcal{M}} \mathcal{L}B & & \end{array} & & \begin{array}{ccc} & & A \\ \exists! \overline{\langle m, f \rangle} & \swarrow & \\ \mathcal{L}B & & \end{array} \end{array}$$

with $\varepsilon_B := \langle \eta_B, id_B \rangle$ for all $B \in |\text{Par}_{\mathcal{M}}(\mathbb{C})|$ and $A \in |\mathbb{C}|$.

Def. C.21 and thus Fact 44 can be generalised by exchanging the class of monomorphisms by an admissible subclass \mathcal{M} , see Def. 5.21. The respective construction is then called a \mathcal{M} -partial arrow classifier and mentioned in Theorem 15.

Finally, there is an important lesser known fact about partial arrow classifiers:

Lemma 45 Partial Arrow Classifiers preserve monomorphisms

When the morphism f in (C.16) is a monomorphism, so is $\overline{\langle m, f \rangle}$.

Proof. Consider again the diagram in (C.18) and recall the adjunction $\Gamma \dashv \mathcal{L}$. Thus \mathcal{L} has left adjoint in Γ and therefore preserves limits, including monomorphisms (which are just special pullbacks). Hence, $\overline{\langle \eta_A \circ m, f \rangle}$ is a monomorphism since it is $\mathcal{L}f$. Monomorphisms compose and so is $\overline{\langle \eta_A \circ m, f \rangle} \circ \eta_A$ a monomorphism. The latter is equal to $\overline{\langle m, f \rangle}$ because of the universal property of the partial arrow classifiers: The upper squares are trivially pullbacks and the lower rectangle is a pullback by the universal property. The whole outline $[id_A \circ m, f \circ id_X]$ is a partial map and equal to

$[m, f]$. Since there is a unique morphism that make the outer square a pullback, we have

$$\overline{[m, f]} = \overline{[\eta_\lambda \circ m, f]} \circ \eta_\lambda$$

which is a monomorphism. □