

Reuse of Model Transformations for Propagating Variability Annotations in Annotative Software Product Lines

Von der Universität Bayreuth
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von
Sandra Greiner

aus Landsberg am Lech

1. Gutachter: Prof. Dr. Bernhard Westfechtel
2. Gutachter: Univ.-Prof. Mag. Dr. Rick Rabiser

Tag der Einreichung: 29.07.2022
Tag des Kolloquiums: 31.10.2022

To my grandparents.

~ Preface ~

Software development is a young profession, and we are still learning the techniques and building the tools to do it effectively. (Martin Fowler, [Fow22])

Decades have passed since in 1968 leading computer scientist have come together at the NATO *software engineering* conference to solve the challenges of the *software crisis* explained by Edsger Dijkstra in the following way: “when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem” [Dij72]. Accordingly, they faced the situation of exponentially growing computer technology which made it hardly possible to write useful and powerful programs in an adequate amount of time. Solution ideas to overcome the software crisis emphasized the adoption of (mechanical) engineering ideas and transfer them to the level of programming software. Thus, key principles, such as approaching the problem in an *iterative* way and *reusing* existing concepts or already programmed source code, were proposed. However, still today, in 2022, the software architect Martin Fowler states the words quoted in the beginning that software development can still be considered a “young profession” where we keep and need to keep learning new methods, for instance, in order to address the rapid technological changes.

In contrast to the 20th century, today one kind of program to solve a problem or to offer some automated functionality does not suffice anymore. A large diversity of computer hardware, involving capabilities which the members of 1968’s Nato conference might not have imagined, such as multi-threading, embedded, cloud, and mobile computing, exists while the customer requirements vary among private persons, companies, and even countries, too. To efficiently serve specific customer needs, *software product line engineering* [CN01; PBL05] has become prominent as one technique adopted from mechanical engineering. By relying on the principle of *organized reuse*, product line engineering aims at supporting the mass customization of a product family. Configuring a software platform allows for deriving customized programs automatically and, thus, increases productivity. Another software engineering direction stresses *abstraction* and *modeling* techniques to automatically generated the source code realizing the modeled system. Using *models* is native to engineers: Herbert Stachowiak [Sta73] postulated the usage of a *model* by employing the principles of *mapping*, *reduction*, and *pragmatism* to reflect the original system. Since then, the discipline *model-driven software engineering* [Sta+06; BTG12] has installed standardized *modeling languages* with formal semantics which, on the one hand, abstract from the source code. On the other hand, *model transformation* languages ease the automated generation of another representation for a given model by focusing on the *declaration* of correspondences and relations between the two representations instead of the concrete execution steps necessary to create the new representation. As a consequence, the level of productivity is raised by abstracting from realization details and automating the source code generation.

The combination of model-driven software engineering and software product line engineering, *model-driven software product line engineering* [Cza+05], promises to take advantage of the upsides of both disciplines to systematically advance the levels of automation and customization. It seeks to build large software systems holding inherent variability efficiently. Models are employed to describe the multi-variant system and model transformations should be used to generate customized source code by the end of the production phase. However, model-driven product line engineering raises several challenges and requires optimized techniques to increase automation and to decrease manual engineering efforts. The goal of this thesis is to solve one of these challenges in a reuse-based, generic and automated way.

Abstract

Model-Driven Software Product Line Engineering (MDPLE) is a discipline that bespeaks increased productivity when developing variability-intense software by combining the benefits of both disciplines, *model-driven software engineering* (MDSE) and *software product line engineering* (SPLE): SPLE grounds on the principles of *organized reuse* and explicit *variability* to build a (*multi-variant*) *platform* from which customized software products can be derived automatically. In contrast, MDSE raises the level of abstraction by employing models of different kinds throughout the development process and eventually represents the software system with executable models. *Model transformations* serve to create not only different model representations from a given model but also to derive executable source code automatically.

While MDPLE should take advantage of the positive effects of each discipline when combining them, several problems, such as the following, threaten these effects: Annotative product lines build the multi-variant platform by superimposing single variants of the software. For deriving a customized product, developers map *annotations* onto elements of the platform and provide a *configuration* of the distinguishing features of the product line. These annotations are Boolean expressions over the common and distinguishing features of the product line. A *filter* mechanism will remove elements from the platform if their annotations are not satisfied by the configuration. Since multiple artifacts, such as models, build scripts, and source code, form the platform, still annotating each of these artifacts is a *manual* process which is tedious and prone to errors. Few approaches automate this task by manipulating the execution semantics of model transformations. This requires exclusive access and, in the worst case, to consider variability explicitly in the transformation which increases the level of cognitive complexity and may require to learn new syntax.

Therefore, in this thesis we develop a solution which acknowledges the existing and mature technology of (single-variant) model transformations which automate the creation of new models or source code but are unaware of the variability present in form of annotations in *multi-variant* product line artifacts. In contrast to existing approaches, our contribution does not modify the execution semantics of existing model transformation languages but reuses them and their generated artifacts to propagate annotations *generically* (i.e., independent of the transformation language and of the transformed instances of metamodels) and *automatically*.

Specifically, the thesis contributes the informal and formal concepts and implementation of 1) an a posteriori bytecode model analysis, 2) an a posteriori propagation of annotations based on transformation trace, as well as 3) a propagation during the execution based on a generic *aspect*. Furthermore, the thesis examines the strengths and weaknesses of employing a propagation DSL and model matching of instances of different metamodels to reconstruct trace information.

The evaluation confirms a significant reduction in manual efforts to annotate the product line: We examine the trace-based propagation of annotations based on traces of different completeness-levels and the generic aspect in a controlled setting with small to medium academic product lines. The aspect-oriented approach assigns 95 – 100% of the annotations correctly. Furthermore, even if the trace completeness is insufficient and does not record all target elements, our approaches compute more than 90% of the annotations correctly whereas the propagation based on generation-complete traces determines *all* target annotations correctly.

In summary, the thesis solves the problem of diminished productivity in MDPLE by providing the concepts, realization, and evaluation of automated techniques to propagate annotations in annotative product lines without the need to change established technologies.

Zusammenfassung

Die Disziplin *modell-getriebene Software Produktlinienentwicklung* (MDPLE) verspricht, die Produktivität beim Entwickeln hochvariabler Software zu steigern, indem sie von den positiven Effekten der beiden Disziplinen *modell-getriebene Softwareentwicklung* und *Software Produktlinienentwicklung* profitiert. *Organisierte Wiederverwendung* und *Variabilität* bilden die Kernprinzipien der Software Produktlinienentwicklung. Beide Konzepte werden benutzt, um kundenspezifische Produkte anhand einer konfigurierten Plattform abzuleiten. Die modell-getriebene Softwareentwicklung hingegen verwendet während des Entwicklungsprozesses unterschiedliche Arten von *Modellen*. Am Ende dieses Prozesses stehen ausführbare Modelle, die das Softwaresystem repräsentieren. *Modelltransformationen* stellen eine Automatisierungstechnik dar, die es ermöglicht, anhand eines gegebenen Modells weitere Repräsentationen und schließlich den ausführbaren Quellcode des Programms automatisch zu generieren.

Obwohl MDPLE von den positiven Eigenschaften beider Disziplinen bei deren Kombination profitieren sollte, ergeben sich verschiedene Probleme, die die Vorteile konterkarieren, unter anderem das Folgende: In annotativen Produktlinien entsteht die Plattform durch das Überlagern aller erlaubten Softwarevarianten. Entwickler weisen den Elementen der Plattform *Annotationen* zu und geben eine Konfiguration der auswählbaren Eigenschaften an, um ein kundenspezifisches Produkt abzuleiten. Annotationen sind Boolesche Ausdrücke über Literale, die die gemeinsamen und unterschiedlichen Eigenschaften der Produktlinie repräsentieren. Sobald eine Annotation nicht durch die gegebene Konfiguration erfüllt ist, entfernt ein *Filter* die entsprechenden Modellelemente von der Plattform, um so das Produkt abzuleiten. Bei der Plattform handelt es sich jedoch nicht um ein einzelnes Artefakt: verschiedene Modelle, Buildskripte, Quellcode, etc., sind darin enthalten, deren variablen Elemente alle meist händisch annotiert werden müssen. Wenige Ansätze automatisieren das Annotieren bislang und manipulieren dabei die Ausführungssemantik bestehender Einzelvariantentransformationen. Dies verlangt jedoch den Zugriff und die Erlaubnis, die Ausführungsumgebung zu ändern. Mitunter muss Variabilität explizit in der Modelltransformation berücksichtigt werden, was wiederum die kognitive Komplexität beim Erstellen und Warten einer solchen Transformationsdefinition erhöht.

Aus diesem Grund entwickelt die vorliegende Arbeit Lösungsansätze, die auf dem Wissen aufbauen, dass Modelltransformationen eine bereits ausgereifte Technik darstellen, um neue Modelle oder Quellcode zu generieren, der jedoch der Umgang mit Variabilität in Form von Annotationen nicht bekannt ist. Im Gegensatz zu den bisher publizierten Ansätzen, ändern die beigetragenen Lösungen nicht die Ausführungssemantik. Stattdessen benutzen sie die Transformationswerkzeuge und deren generierte Artefakte, um Annotationen *generisch* (d.h., unabhängig von einer Transformationsprache oder Transformationsdefinition) und *automatisch* in das Zielmodell zu übertragen.

Insbesondere trägt die vorliegende Arbeit die informellen und formellen Konzepte sowie die Implementierung 1) einer a posteriori Analyse von Bytecodeinstruktionen, 2) einer a posteriori Übertragung von Annotationen anhand von Traceinformation sowie 3) einen aspekt-orientierten Ansatz, der die Annotationen während der Ausführung in generierten Quellcode einbettet, bei. Des Weiteren untersucht die Arbeit die Vor- und Nachteile einer Propagations-DSL und des Vergleichs von Instanzen unterschiedlicher Metamodelle bezüglich der Möglichkeit Traceinformation zu rekonstruieren.

Die Evaluierung der beigetragenen Ansätze bestätigt eine signifikante Reduktion des händischen Annotationsaufwands. Dazu wird die Annotationsübertragung anhand von unterschiedlich vollständigen Traces und bei der Verwendung eines generischen Aspekts in kontrollierten Experimenten mit kleinen und mittelgroßen akademischen Produktlinien untersucht. Es zeigt sich, dass der aspektorientierte Ansatz 95 – 100% der Annotationen richtig ermittelt. Außerdem berechnet die trace-basierte Übertragung über 90% der Annotationen korrekt, selbst wenn die Vollständigkeit

des Traces nicht hinreichend genau ist. Bei der Berechnung von Annotationen anhand von erzeugungsvollständigen Traces werden sogar *alle* Annotationen korrekt bestimmt.

Insgesamt löst die vorliegende Arbeit damit ein Problem der minimierten Produktivität in der modell-getriebenen Software Produktlinienentwicklung, indem sie die theoretische Konzepte, deren Realisierung und Evaluierung beiträgt, um Annotationen in annotativen Produktlinien automatisch und ohne die Veränderung bewährter Technologien auf weitere Artefakte zu übertragen.

Contents

Preface	i
Abstract	iii
Zusammenfassung	v
I Introduction	1
1 Problem Statement	3
1.1 Background	4
1.2 Research Objective	5
1.3 Scope of Contribution	7
1.4 Overview	8
1.4.1 Contribution	8
1.4.2 Structure	10
II Relevant Software Engineering Disciplines	11
2 MDSE	13
2.1 Modeling Concepts	14
2.1.1 Preliminaries – Associated Engineering Disciplines	14
2.1.2 Background	14
2.1.3 Metamodels	15
2.1.4 Classifying Properties	19
2.1.5 Eclipse Modeling Framework	21
2.2 Model Transformations	21
2.2.1 Classification	22
2.2.2 Transformation Languages and Tools	25
3 SPLE	30
3.1 Terminology	32
3.1.1 Product Line Analysis	32
3.1.2 Product Generation	33
3.1.3 Engineering Strategies	33
3.2 Development Processes	34
3.2.1 Three Simultaneous Activities	34
3.2.2 Two-Layered Process	34
3.2.3 Double Spiral Model	35
3.2.4 Four-Clustered Process	35
3.3 Variability Modeling	36
3.3.1 Feature Models	36
3.3.2 Further Types of Variability Models	37
3.3.3 Endnotes	38
3.4 Variability Implementation Techniques	38

3.4.1	Basics	38
3.4.2	Compositional Variability	40
3.4.3	Transformational (Delta-Oriented) Variability	41
3.4.4	Annotative Variability	42
3.4.5	Multi-Paradigmatic Approaches	44
3.5	Product Well-Formedness	44
3.6	Summary	45
4	MDPLE	46
4.1	Multi-Variant Models	47
4.1.1	Preliminaries	47
4.1.2	Variability Mechanisms	49
4.2	Annotation Maintenance in Existing MDPLE Solutions	57
4.2.1	Classifying Properties of Annotation Maintenance	57
4.2.2	Annotation Maintenance in MDPLE Solutions	59
4.2.3	Results	61
4.3	Multi-Variant Model Transformations	62
4.3.1	Model Transformation Reuse	62
4.3.2	Classification	63
4.3.3	Variation in Transformation	65
4.3.4	Annotation Transformation	66
4.3.5	Results	68
4.4	Bottom Line	69
III	Trace-Based Propagation of Variability Annotations	71
5	Informal Properties of Trace-Based Propagation	73
5.1	Example of Trace-based Transformation	74
5.1.1	Single-Variant Model	74
5.1.2	Example Transformation	75
5.1.3	Multi-Variant Model	76
5.2	Properties of Transformation Traces	77
5.2.1	Traces in Existing Model Transformation Solutions	77
5.2.2	Feature-Based Trace Classification	79
5.2.3	Common Trace Metamodel for Annotation Propagation	81
5.3	Trace-Based Annotation Propagation	81
5.3.1	Schematic Overview	82
5.3.2	Annotation Propagation Procedure	82
5.3.3	Computational Model	83
5.4	Summary	85
6	Formal Foundations	86
6.1	Models as Graphs	87
6.1.1	Single-Variant Models	87
6.1.2	Graph Morphisms	88
6.2	Variability in Graphs	90
6.3	Graph Transformations	93
6.3.1	(In-Place) Rules and Derivations	93
6.3.2	Properties of Derivations	95
6.3.3	Out-Place Rules and Derivations	96
6.4	Trace-Based Annotation Propagation	101
6.4.1	Propagation Algorithm	101
6.4.2	Commutativity of Derivations	103
6.5	Summary	106

IV	Extensions to Trace-Based Annotation Propagation	107
7	Missing Trace Information	109
7.1	<i>Generation-Complete</i> Traces	111
7.1.1	Problem Description	111
7.1.2	Completely Annotated Target Model	111
7.1.3	Correctness of Propagated Annotations	111
7.1.4	Consequences	113
7.2	<i>Coarse-Grained</i> Traces	113
7.2.1	Problem Statement	114
7.2.2	Bytecode Instruction Analysis	116
7.2.3	Classification of Patterns in Model Transformation Languages	119
7.2.4	Propagation Process	125
7.2.5	Foundations	130
7.2.6	Discussion	132
7.3	<i>Incomplete</i> Trace Information	135
7.3.1	Problem Statement	135
7.3.2	Foundations	137
7.3.3	Computation of Missing Annotations	140
7.3.4	Discussion	149
7.4	<i>No Persistent Trace</i> Information	155
7.4.1	Problem Statement	156
7.4.2	Propagation DSL	156
7.4.3	Trace Generation by Model Matching	162
7.5	Incremental Annotation Propagation	169
7.5.1	Problem Statement	169
7.5.2	Background	171
7.5.3	Incremental Annotation Maintenance	173
7.5.4	Discussion	177
7.6	Summary	179
8	Model-To-Text Annotation Propagation	181
8.1	Problem Statement	182
8.1.1	Motivation	182
8.1.2	Consequences	184
8.2	Aspect-Oriented Multi-Variant Source Code Generation	184
8.2.1	Template-Based Model-To-Text Transformation	185
8.2.2	Aspect-Oriented Programming	190
8.3	Foundations	191
8.3.1	Descriptive Overview	191
8.3.2	Computational Model	197
8.3.3	Formal Foundations	201
8.4	Discussion	209
8.4.1	Computational Model	209
8.4.2	Related Work	211
8.5	Summary	212
V	Validation	213
9	Implementation	215
9.1	Preliminaries	216
9.1.1	Eclipse Modeling Framework	216
9.1.2	Famile	217
9.1.3	Delimitation	220
9.2	Multi-Variant Model Transformation Framework	220

9.2.1	Overview	220
9.2.2	MuVaTra Core	222
9.2.3	Mapping Maintenance With Famile	223
9.3	Realization Specifics	225
9.3.1	Trace-Based Annotation Propagation	225
9.3.2	ATL/EMFTVM Rule Analysis-Based Propagation	228
9.3.3	Model-To-Text Aspect-Oriented Propagation	233
9.4	Summary	239
10	Evaluation	240
10.1	Evaluation Goal	241
10.1.1	Genericity of Propagation	241
10.1.2	Correctness of Propagation	242
10.1.3	Propagation Benefit	244
10.2	Evaluation Setup	244
10.2.1	Commutativity Evaluation Framework	245
10.2.2	Product Lines	250
10.2.3	Model-To-Model Transformations	254
10.2.4	Model-To-Text Transformations	270
10.3	Results	272
10.3.1	Ecore2UML	273
10.3.2	Base UML2Java	275
10.3.3	Advanced UML2Java	277
10.3.4	Ecore2SQL	283
10.3.5	Model-To-Text	286
10.4	Discussion	292
10.4.1	Trace-Based Propagation	293
10.4.2	Aspect-Oriented Propagation	300
10.4.3	Threats to Validity	302
10.5	Summary	307
VI	Conclusion	309
11	Resume	311
11.1	Summary of Contribution	312
11.1.1	Overview of Content	312
11.1.2	Consequences	313
11.1.3	Design Decision	313
11.2	Benefits and Take-Away	314
11.2.1	For Research	314
11.2.2	For Practitioners and Industry	315
11.2.3	Take Away	316
11.3	Future Research	317
A	Appendix	319
A.1	Classification of Annotation Maintenance in MDPLE Approaches	319
A.2	ATL/EMFTVM Bytecode Instruction Opcodes	320
A.3	Flexible Preprocessor	320
	List of Figures	323
	List of Tables	325
	List of Listings	326
	Abbreviations	327
B	Bibliography	328
B.1	Third-Party Publications	328

B.2 (Co-)Authored Publications Related with Thesis	349
B.3 Further (Co-)Authored Publications	351
Acknowledgments	352

Part I

Introduction

Chapter 1 Problem Statement

*Take the best that exists and make it better
When it does not exist, design it.*

Sir Frederick Henry Royce

~

This thesis offers a solution for a problem situated in *annotative* model-driven software product line engineering. To state the targeted problem, this introduction to the thesis offers an overview of the research disciplines and the current shortcoming in model-driven software product line engineering. Based on that overview, the chapter states the problem as well as the research objectives and summarizes the research outcome which the following chapters present in detail.

After stating background information and introducing the problem in Sec. 1.1, Sec. 1.2, Sec. 1.3 and Sec. 1.4 present the research objectives, delimit the scope by design decisions and enumerate the concrete contributions of the thesis, respectively. The end of this chapter outlines the structure of the contents of the thesis.

1.1 Background

Overview This thesis solves a problem which is situated in the discipline of *model-driven product line engineering*, which combines the methods used in the disciplines *model-driven software engineering* and *software product line engineering*. The goal of the thesis is to reuse *model transformations*, a state-of-the-art technique from the discipline of model-driven software engineering, to propagate variability annotations in model-driven product lines. The following paragraphs introduce each discipline in short to explain the problem being solved thereafter.

Disciplines *Software product line engineering* [PBL05; Ape+13], on the one hand, builds on the principles of *organized reuse* and *variability*. The level of *domain engineering* builds a superimposed platform of all variants whereas the level of *application engineering* maintains single variants which are delivered as final products in the end [PBL05]. A key artifact of systematic product line engineering are *variability models* [Cza+12] which capture differences (and commonalities) among the software variants of a product line. *Variability mechanisms* [Ape+13] serve to *derive* and, in the end, build customized variants. The most common ones are the *compositional*, *transformational* (also delta-oriented), and *annotative* variability mechanisms.

This thesis focuses on the annotative variability mechanism (c.f., Sec. 3.4.4), which constructs a superimposed *multi-variant platform* to incorporate all program variants available to a customer. Without loss of generality, we assume that a *feature model* [Kan+90] captures the variability of the system. *Annotations* are Boolean expressions over the features of the feature model and are mapped onto the elements of the platform. For deriving a variant, the product developer configures the variability model by providing selection states for each optional feature. Given such *feature configuration*, a *filter* mechanism will remove elements if their annotations are not satisfied by this feature configuration.

Model-driven software engineering [BCW12; Sta+06; Sch06], on the other hand, relies on the principles of abstraction and automated generation. *Models* which are instances of *metamodels* [Küh06] represent the software system which should be created. To generate source code and thereby realizing the executable program, *model transformations* [SK03; CH06] serve as key technology for automating the creation of another representation, such as refined models or source code.

Furthermore, in *model-driven software product line engineering* [Gom05], models form the core elements of the platform. As such, not only one *domain model* but multiple types of models may compose the platform. Similar as classical software product line engineering, model-driven product line engineering can employ the same variability mechanisms which instead of code manage *multi-variant models* [SBW16]. In annotative model-driven product lines, the multi-variant model, also denoted as *150 %-model*, composes the variants and maps annotations onto model elements to allow for filtering customized models from that platform model. We assume classical *forward engineering* development which builds coarse-grained models from requirements engineering in early development stages and subsequently refines them into design and implementation models. These refinements may occur over several iterations. To support the *automated* propagation of information of a coarse-grained level to the next, more fine-grained level, model-driven software engineering employs (incremental) *unidirectional model transformations*. As an example, the product line developer can refine an Ecore model into a UML class model or create a relational database schema as additional information. The developer can further refine the UML class model into a Java model which can be used to generate (multi-variant) source code. A model-driven product line developer typically employs transformations to automate the creation of new types of models.

Challenge Even though mature tool support for multiple kinds of model transformation languages exists including their proper integration into the Eclipse IDE, the (single-variant) transformation engines and languages are unaware of the variability in form of annotations mapped onto the multi-variant models of a model-driven product line. For this reason, it is *possible* to execute a *single-variant* transformation to generate another model representation for an input *multi-variant* model (e.g., create a UML class model from an Ecore model) but it is *impossible* to propagate the

annotations mapped onto source model elements therewith, in general. Therefore, product line developers need to map annotations onto target elements *manually*. This process is laborious, time-consuming, and prone to errors: Either the developers assign annotations completely anew or they look up the annotations of (what they assume to be) corresponding source elements. Alternatively, extending existing transformation definitions – if possible – is a similarly error-prone effort which has to be repeated for each kind of relevant transformation definition.

Existing Solutions Before working on the thesis, only few techniques existed that regard variability in model transformations: *Variability-based rules* [Str+18b] annotate existing rules and allow their variation to serve the purpose of generating different, customized artifacts by configuring the transformation. The *lifting* of transformation rules [Sal+14] changes the semantics of the model-to-model graph transformations and, thus, requires to modify the execution engine. A similar approach incorporates variability directives by extending the syntax of the transformation language **ATL** (Atlas Transformation Language) [Sij10] and transforms the resulting execution model with an ATL transformation into the default ATL execution model in a higher-order transformation. In summary, these three and further closely related approaches [Tae+17; Fam+15] either manipulate the execution semantics or require to consider variability in the transformation definition explicitly. On the one hand, accessing the execution engine implies the right and possibility to access and manipulate the existing semantics of a language. On the other hand, exposing the transformation developer to the variability increases the amount of cognitive complexity to define the transformation definition and may require to learn new transformation syntax and semantics.

Consequence Driven by the observations of the possibility to reuse model transformations and the limitations of existing techniques, the goal of this thesis is to answer and provide an adequate solution to the following problem:

So far, the product line developer needs to annotate multi-variant (source) models manually. As one product line consists of multiple modeled artifacts, the developer has to redundantly map annotations onto each newly introduced artifact in laborious and error-prone handwork. Since model transformations automate the creation of target representations, they should be equally reusable to propagate the annotations of source elements to corresponding target elements in an automated way.

1.2 Research Objective

Based on the goal to propagate annotations automatically by reusing transformation technology, we derive the following research objectives targeted in this thesis.

RO1 Reuse of existing transformation languages and engines

RO2 Automation of propagating annotations

RO3 Genericity of propagating annotations (i.e., independent of a specific transformation language or the transformation definition)

RO4 Correctness of propagated annotations

RO1 Reuse Firstly, the objective to **reuse** existing technology aims to preserve the already invested time, work, and efforts. Transformation languages with formal semantics have already been designed and optimized for efficiency. For instance, active research continuously optimizes the unidirectional model-to-model transformation language ATL [Jou+08] to perform faster and more accurately by improving the front-end (e.g., validation and repair operations) [Var+21] and the back-end (e.g., faster compile and execution time) [Cal+19; CGL18; Cua+22]. At the same time, the ATL zoo [All22] collects several transformation definitions specified in this language. When building a model-driven product line, the developer may employ existing transformation

definitions if instances of corresponding metamodels form part of the product line. For example, transformation definitions which turn a class model into a relational database schema to realize object-relational mappings serve as a standard example to demonstrate the syntax and semantics of the language. Therefore, this transformation is available in several languages, such as QVT-R [Wes15] and QVT Operational Mappings [Obj16], ATL [All22], and Henshin [Kra].

Overall, reusing the existing know-how offers the benefits that there is *no* necessity for the product line developer to

- learn new language constructs
- think about the transformation of annotations explicitly, so that the level of cognitive complexity of the transformation definition does not increase additionally
- extend existing transformation engines
- create new transformation definitions from scratch (which might require an expert in model transformations)

RO2 Automation Secondly, the product line developer should have to invest no or (only if indispensable) minimal manual annotation efforts. Thus, the propagation mechanism needs to map the annotations onto elements of the target model **automatically**. Accordingly, the propagation mechanism should ensure that an annotation is mapped onto all (relevant) elements which are created by the reused transformation after having performed the propagation. If the product line developer needs to modify an automatically assigned annotation manually after the automatic propagation, executing the transformation another time (e.g., because the source model changed) should *preserve* the manually modified annotations.

RO3 Genericity Thirdly, it should be possible to employ the same propagation mechanism to propagate annotations **generically**. Neither the metamodel to which the source and target model conform nor the kind of transformation engine which creates the target model should provoke a different propagation mechanism. In other words, the propagation should be processed independently of the transformation definition and of the transformation engine.

Building a generic solution offers the benefit of (mostly) technology-independent applicability. By not assuming a certain metamodel or transformation language, a respective solution can be employed in almost any transformation scenario. Thus, the product line developers are not forced to employ a specific tool environment, transformation language, or transformation definition but can reuse the existing tools and languages which are available and adequate.

RO4 Correctness Finally, the propagation of annotations should be **correct**. Correctness is not defined by prescribing the concrete annotations assigned to target elements but by examining their effect on the derived variants. In concrete, transformation and filter operations need to *commute* as sketched in Fig. 1.2.1:

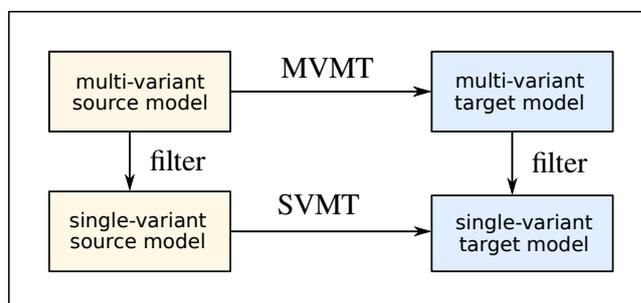


Figure 1.2.1: *Commutativity* of transformations noted informally.

A *multi-variant model transformation* (MVMT) creates the multi-variant (annotated) target model. Then, the same *single-variant target model* should be created by filtering the *multi-variant source model* and transforming the resulting *single-variant source model* with the reused *single-variant*

model transformation (SVMT) as results from filtering the *multi-variant target model* by the same feature configuration. If this property holds for each valid feature configuration, we consider the propagation result correct. To the best of our knowledge, the Lifting [Sal+14] approach postulated *commutativity* of transformations as correctness criterion for the first time.

1.3 Scope of Contribution

According to stating the background information and the problem addressed in this thesis, we take several design decisions which delimit the scope of the approach. Although the detailed description of the three software engineering disciplines introduces and explains some of these decisions subsequently in Part II, this section collects the important ones and thereby delineates the scope of the contribution.

D1 Annotative Model-Driven Product Line

The contributed annotation propagation mechanisms serve to diminish the problem of manually annotating a model-driven product line model and are essential when employing the annotative variability mechanism. We do not examine the effect or necessity of a similarly automated annotation technique when utilizing the compositional or transformational variability mechanism or solely implementation techniques to build varying products.

D2 Element-Wise Mappings

Multiple ways to persist the mapping of an annotation onto a model element exist which Sec. 4.2.2 illuminates. Our solution targets the maintenance of annotations recorded by the means of element-wise mappings. Those may be either stored externally, which means in a physically file separated from the domain model, or internally, which means stored as elements of the target representation (e.g., as preprocessor directives). Thus, we do not consider *annotation-wise* mappings which record one kind of annotation once and map several model elements, particularly of physically different models, onto the corresponding single annotation.

D3 Constrained Variability

As Sec. 4.1.1 will explain, we consider the transformation of domain models which realize constrained variability. Since (single-variant) state-of-the-art modeling languages, such as Ecore or UML, do not offer dedicated constructs to represent and respect the variability inherent in a multi-variant domain model, the latter type of model is subject to the constraints defined in the single-variant modeling language's syntax. Accordingly, for instance, it is not possible to define names of one UML class which vary with respect to the given feature configuration. This would require to persist different values of the single-valued attribute `name` and map different annotations onto them. Only few approaches support such mechanism, mostly in a user-hidden way [Reu+20; SW16; BS12]. Therefore, we restrict our solution to constrained variability.

D4 Proactive Development

This thesis focuses on the proactive (i.e., from scratch) development of a product line. Even though we assume that at least one multi-variant source model is annotated completely, we do not regard combining multiple products into a product line as in *extractive* approaches or to integrate new products into an existing platform as in *reactive* approaches.

D5 Non-Intrusive Reuse

Following from the research objective to *reuse* existing model transformation technology, we restrict the solution concepts to modify neither the syntax of existing transformation languages nor the semantics of their execution engines. On the one hand, in this way the product line developer may reuse existing transformation definitions to transform multi-variant models. This is beneficial, particularly, whenever the developer is unable to access the transformation engine. In addition, the developer is not required to know profoundly how model transformations behave. On the other hand, in the transformation definition, the developers do not have to regard the additional complexity introduced by annotations.

D6 Commutativity as Correctness Criterion

Finally, as stated as research objective (c.f., Sec. 1.2), we foster commuting multi- and single-variant model transformations as correctness criterion. According to this criterion, an annotation mapped onto a target element is not determined uniquely. Thus, we do not evaluate the correctness of an annotation by prescribing a specific expression but by regarding the presence of the element onto which it is mapped in derived products.

In summary, these design decisions determine and delimit in which areas of software product line engineering the present work is applicable.

1.4 Overview

In the course of this thesis, we contribute diverse mechanisms which all intend to propagate annotations to the target model by adhering to the four research objectives. For that reason, Sec. 1.4.1 illustrates the novelties contributed in this thesis, dedicating a particular focus on their relationships and dependencies. While Sec. 1.4.1 announces the chapters in which the contributions appear, Sec. 1.4.2, provides an extended outline of the entire thesis.

1.4.1 Contribution

To meet the research objective, this thesis examines different solution strategies. This section provides an overview of the main contributions and foreshadows how they appear subsequently in the following chapters. Fig. 1.4.1 illustrates the dependencies between the contributions of this thesis and states the corresponding chapters. B.2 collects the peer-reviewed publications which lay the base for each chapter and the following descriptions cite the respective ones. Furthermore, the introduction to each chapter mentions the publications which lay its foundation. The following paragraphs subsequently summarize the contents of each contribution which answer four overarching questions:

How do existing MDPLE tools maintain annotations and keep them consistent? The first part of the contribution considers the related work by examining annotation maintenance methods in existing MDPLE tools [GW21] and classifies the properties of multi-variant model transformations [GW21; WG20a].

Although several solutions to maintain variability in model transformations exist, Sec. 4.3 concludes that none of the examined MDPLE *tools* considers or employs an automated mechanism that explicitly keeps corresponding annotations of diverse artifacts consistent.

How and to what extent can we propagate annotations automatically? The following chapters contribute conceptual solutions to propagate annotations automatically, generically and correctly by reusing already existing technology.

On the one hand, Chp. 5 and Chp. 6 describe the concept of propagating annotations based on a transformation trace in *model-to-model* (*M2M*) transformations, at an informal level [GSW17], and the conditions which have to be satisfied, at a formal level [WG18; WG20a], respectively.

On the other hand, Chp. 8 presents an approach to employ aspect-oriented programming techniques to map annotations onto text fragments in *model-to-text* (*M2T*) transformations [GW18b]. As the computational model, which formally proves the correctness of trace-based propagation, is quite restrictive and fosters the usage of *complete* traces, Chp. 7 offers practical solution strategies to maintain situations which violate the computational model. Therefore, it discusses the effect of a *generation-complete* trace and describes propagation strategies for when traces are *incomplete* [GW19c; GW18c] or not existing at all [BG18]. This involves the necessity to preserve annotations which the developer has to modify manually because of inaccurate annotations provoked by missing information [GW20].

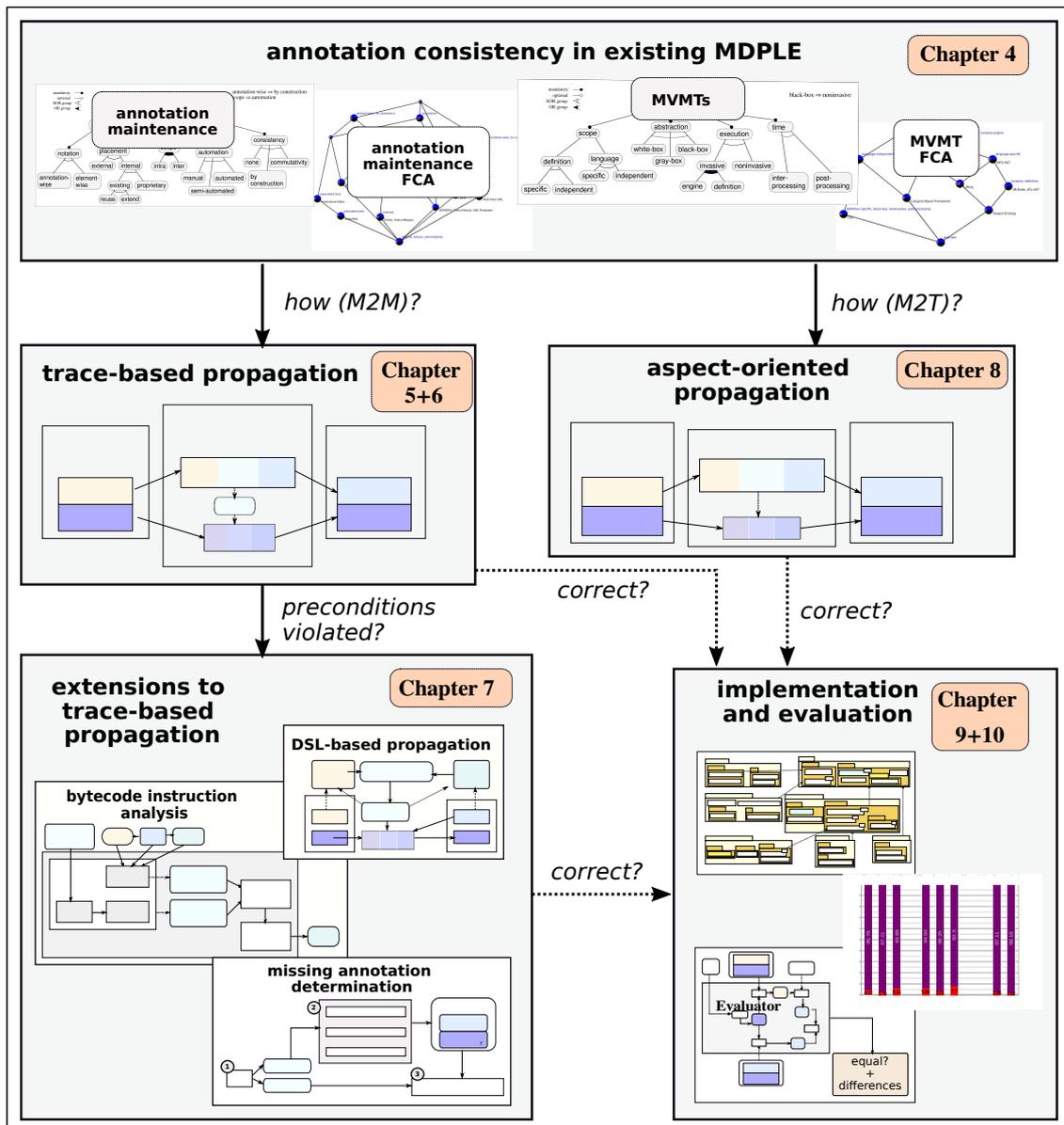


Figure 1.4.1: Overview of contributions and their appearance in the thesis. The second part of the bibliography list (B.2) compiles the publications that lay the grounds for these chapters.

To what extent do the conceptual approaches satisfy the objectives in practice? The final part presents an implementation of the approaches as proof of concept (c.f., Chp. 9) and evaluates the benefits and limitations gained by the propagation strategies (c.f., Chp. 10). The respective chapters illustrate the functionality of the realizations to propagate annotations as well as how commutativity can be examined practically [GW19b]. The evaluation of the propagation approaches complements already published work [GW19a; GW18a] and is extended to mitigate some threats of validity.

While – by design – each of the solutions is automated, generic and reuses existing technology, the results demonstrate that in general correctness cannot be guaranteed in practical situations. However, the evaluation shows that the propagation based on generation-complete traces results in commuting transformations in each of the examined situations. If traces are incomplete, the proposed alternative propagation mechanisms (completion strategies) compute more than 90% of the annotations correctly which means that less than *ten* percent of the target annotations have to be repaired manually (in the worst case scenarios). In addition, the evaluation of the aspect-

oriented propagation of annotations demonstrates that violations to its computational model may provoke erroneous annotations. However, in the respective multi-variant source code we have to repair only *four* annotations which means that in total more than 98% of preprocessor directives are computed and inserted correctly.

1.4.2 Structure

Before closing the first part of this thesis, this section explains the organization of the remainder. Part I gave an overview of the contribution in terms of stating the problem, the research objectives, and the design decisions. The second part, Part II, scrutinizes the research disciplines. In this way, Part II narrows down the problem and offers detailed background information to readers not entirely familiar with either model-driven software engineering (Chp. 2), software product line engineering (Chp. 3), or their combination (Chp. 4). Furthermore, while the first two sections of Chp. 4, Sec. 4.1 and Sec. 4.2, present background information how to combine both disciplines, Sec. 4.3 illuminates how to convey variability information in model transformations. The closing section motivates the necessity for propagating annotations based on the preceding examination of existing solutions.

The following parts present the contributed propagation approaches: Part III describes the computational model for trace-based propagation: Chp. 5 illustrates the concept of trace-based propagation and its properties informally whereas Chp. 6 offers the foundations formally noted. As the computational model may be violated in practice, Part IV explains extensions to the trace-based propagation approach. On the one hand, Chp. 7 offers several strategies to solve situations in which trace information is only insufficiently available. On the other hand, Chp. 8 describes how to transfer variability information in model-to-text transformations by employing an aspect-oriented approach.

The thesis continues to examine to what extent the contributed concepts are applicable in practice in Part V: Chp. 9 presents the implementation of the contributed propagation mechanism whereas Chp. 10 evaluates inasmuch the propagation mechanisms satisfy the research objectives, particularly commutativity, in varying transformation scenarios.

Part VI finalizes the thesis: Chp. 11 summarizes and reflects on the contribution, discusses design decisions, and collects important lessons learned which may be relevant not only for research but also for practitioners and the industry. An outlook on potential future research directions concludes the thesis.

Part II

Relevant Software Engineering Disciplines

Chapter 2 Model-Driven Software Engineering

*In nova fert animus mutatas dicere formas / corpora*¹

Ovid, Met., Book I, Lines 1–2

~

As first part of introducing background information on the thesis' topic, this chapter illuminates the discipline *model-driven software engineering*. While models, which conform to metamodels, abstract and reduce the original in a pragmatic way, their sole use will not produce executable source code. Thus, model transformations represent the key technology to maintain the models in this discipline.

Therefore, this chapter introduces general *modeling* concepts in its first part, Sec. 2.1 while the second part of this chapter, Sec. 2.2, scrutinizes the key technology in model-driven software engineering, the concept of *model transformations*.

¹ “I intend to speak of forms changed into new entities”

2.1 Modeling Concepts

Road Map Before diving into the characteristics of model-driven software engineering, this section discusses terminology associated with the discipline in order to pigeonhole our approach correctly. In Sec. 2.1.1. Thereafter, Sec. 2.1.1 delimits different terminology to build a common understanding of the area in which this thesis is situated whereas Sec. 2.1.2 illuminates the motivation why software engineers may employ models as form of abstraction to build software systems. Finally, the remaining sections introduce basic modeling concepts and their realization mechanisms to talk about models throughout the remainder in an informed way.

2.1.1 Preliminaries – Associated Engineering Disciplines

Besides *model-driven software engineering* (**MDSE**), the terms *model-based engineering* (**MBE**) and *model-driven (software) development* (**MDD**) coexist as names of this research discipline. All of these directions are centered around the usage of models in order to raise the level of abstraction when creating a system.

While MDSE and MDD can be regarded synonyms, MBE focuses on models differently. In MBE, models are associated artifacts which, together with source code, make up the system. They are not intended to replace the necessity for manually written source code. Conversely, in MDSE – also denoted as *model-driven engineering* [Sch06]– or MDD[Sta+06; BCW12] *models* are the primary artifacts to prescribe the system, thereby ensuring that executable source code can be generated automatically. This thesis contributes a technology to enrich MDSE.

2.1.2 Background

In all engineering disciplines models have always predominated the development of a system:

Model Definition in (General) Engineering According to Stachowiak, a *model* respects the following three essential features: *mapping*, *reduction* and *pragmatism*. Firstly, a model always maps the elements of a natural or an artificial original to some representation, the elements of model. Secondly, the model reflects relevant parts of the original only, thereby reducing it. Finally, the pragmatic feature involves three instances, the *user* of the model, the point in *time* when it is used and the *purpose* to which the model is used. By considering these three factors, the model is trimmed to cover the system best from a pragmatic point of view. [Sta73]

Model Definition in Software Engineering The following proposition by Kühne transfers Stachowiak’s general definition of a model in engineering sciences into a definition of a model in software engineering.

A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made. [Küh06]

In his work, Kühne presents key concepts on how to recognize models in MDSE by defining the relationship of a *model* to the *system* it describes. In this context, he further elaborates on metamodels and model transformations, to which we refer in subsequent sections.

First of all, as stated before, a model abstracts from a real or a language-based system with a specific purpose in mind [Küh06]. While the abstraction of the real-world (system) in form of a model is in line with Stachowiak, the abstraction of a language(-based system) may explicitly target the software engineering community.

Abstracting Software Languages In a similar direction, the overall aim of *software language engineering* has been – and is still today – to raise the level of abstraction and automation. By now, only few programmers write assembler code or even machine code. Even when hardware-specific optimizations are required, developers prefer *domain-specific languages* [Fow10], from which GPL code is generated which in turn is transformed into assembler or machine code. A prominent example of abstracting from assembler languages is **mbeddr** which provides a language workbench

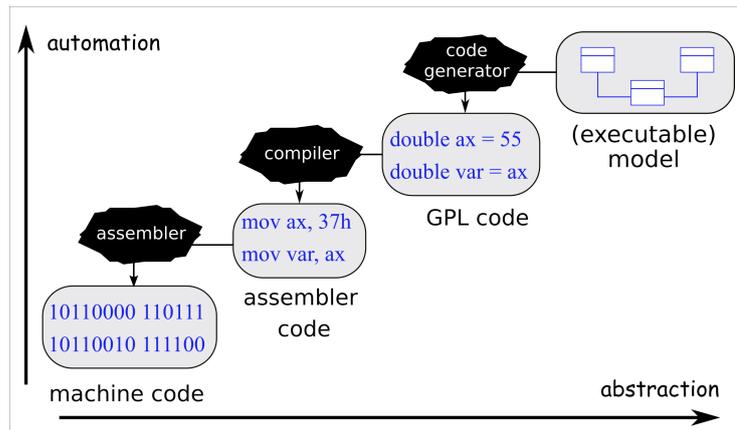


Figure 2.1.1: Automation and abstraction climax.

for specifying source code for embedded systems [Voe+13]. Today, the majority of programmers (still) employs general purpose languages, such as Java or C/C++ [Str13], when developing software. Compilers [Aho+13], in turn, generate a lower-level representation, typically machine or assembly code, from the GPLs automatically. As shown in Fig. 2.1.1, executable models continue this climax of abstracting the system and generating the lower-level representation automatically. To this end, models adhering to formal syntax allow for deriving the GPL source code from the model. As a consequence, the question arises how to note a model formally in order to generate executable source code from it?

Object Management Group In 1989, the *object management group*² (OMG) has been founded as international non-profit consortium for setting modeling standards and, thus, answering these questions. By now, many standards have been established by this facility and some will be introduced in the next paragraphs as well as the answer to the question how to express models.

2.1.3 Metamodels

Metamodels are the prominent answer to the question of how to (de-)note a model. The prefix “meta” originates from the Greek language where “μετα” means “after”. *After* writing his work “on nature” (*φυσικη*), Aristotle (4th century BC) published notes reflecting on this first work. These second notes are nowadays commonly called “Metaphysics” (*μεταφυσικη*) since they use methods defined in the initial work to reflect on physics (though originally there was no official title). The idea of using a science to define this science itself has been put up firstly in modern times by mathematicians, such as David Hilbert (1862-1943) and Kurt Gödel (1906-1978). In 1920 Hilbert proposed a research program called “metamathematics” using mathematical methods, for example a universal axiom system, to define mathematics. Similarly, the idea of defining physics based on physics was caught up by Gödel. This historic excursion indicates that the introduction of metamodels by the OMG, which is described in the following paragraphs, is no revolutionary new idea but continues a long historical tradition.

Overview The grounds for metamodeling, i.e., defining models based on models, lie in the specifications of the *Model-Driven Architecture* and the *Meta Object Facility*, both explained in the following paragraphs. Furthermore, as a metamodel defines the *abstract syntax* and *static semantics* of a model, this section delimits metamodels from *modeling languages*. Additionally, the sequel introduces the key concepts of the UML metamodel, which is one of the most widespread metamodels and serves as example in the subsequent explanations, and the OCL language, which allows for defining the static semantics of a metamodel.

² www.omg.org

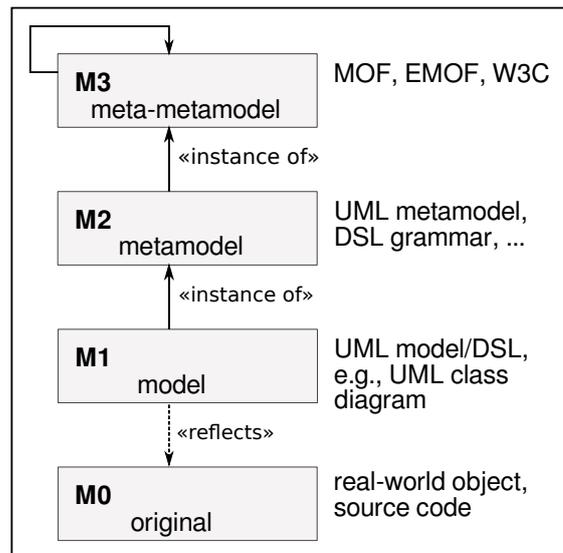


Figure 2.1.2: Classical MOF modeling hierarchy.

Model-Driven Architecture The foundations for modeling lay in the *model-driven architecture* (MDA) [Obj00] proposed by the OMG. The MDA specifies the design of *platform independent models* (PIM) which according to Richard Solely, main author of the specification draft, allow “to derive code from a stable model” ([Obj00], p.3) while the underlying technology changes over the course of time. The term *platform* refers to any technical or business details specific to a system. When deriving source code for a specific platform, firstly the PIM needs to be turned into a *platform specific model* (PSM), an intermediate state which additionally includes the technical or business semantics of the target platform.

Meta Object Facility The *Meta Object Facility* (MOF) [Obj19b] standard, proposed by the OMG, declares the standards for metamodeling in the context of the MDA. Classically, the MOF declares a four-layered hierarchy of modeling which is sketched in Figure 2.1.2. At the bottom, M0, resides the original. The original may be a real object, which is reflected by some element in the model, or, for instance, source code, which the model describes. Level M1 contains the actual model which reflects the original. In turn, the metamodel at level M2 defines the syntax for models residing at level M1 which are instances of the metamodel. Finally, metamodels are instances of a meta-metamodel, located at level M3. Since this hierarchy could be extended to infinity, the standard involves a bootstrapping mechanism. The topmost layer defines its language itself, thus, being a *reflexive* metamodel. Ex. 2.1.1 provides a concrete example of models residing at the four levels and is explained after introducing the specializations of MOF, EMOF and CMOF.

The MOF standard further distinguishes between essential and complete MOF: *Essential MOF* (EMOF), as part of the MOF standard, offers the minimal framework for building metamodels at level M2. Thereby, the EMOF model merges elements of the UML2 standard, which is introduced in the sequel, with basic MOF concepts and constraints. In this way, EMOF suffices to bootstrap metamodels conforming to EMOF. Finally, *Complete MOF* (CMOF) merges several further packages, for instance the EMOF and additional parts of the UML2 metamodel, in order to provide for advanced metamodeling capabilities. [Obj19b]

Example 2.1.1: Modeling Hierarchies

Fig. 2.1.3 exemplifies the different hierarchy levels by using concrete model excerpts: The model at level M1 reflects families consisting of members. The family Obama with a parent called “Michelle” and a child called “Natasha” serves one possible (partial) instance, representing the original. The model at level M1 is expressed in UML which resides as metamodel at level M2. At the topmost level, the meta-metamodel defines the syntax to

which the metamodels conform. In this example, we assume the UML model is expressed as an Ecore model, as such Ecore servers as meta-metamodel.

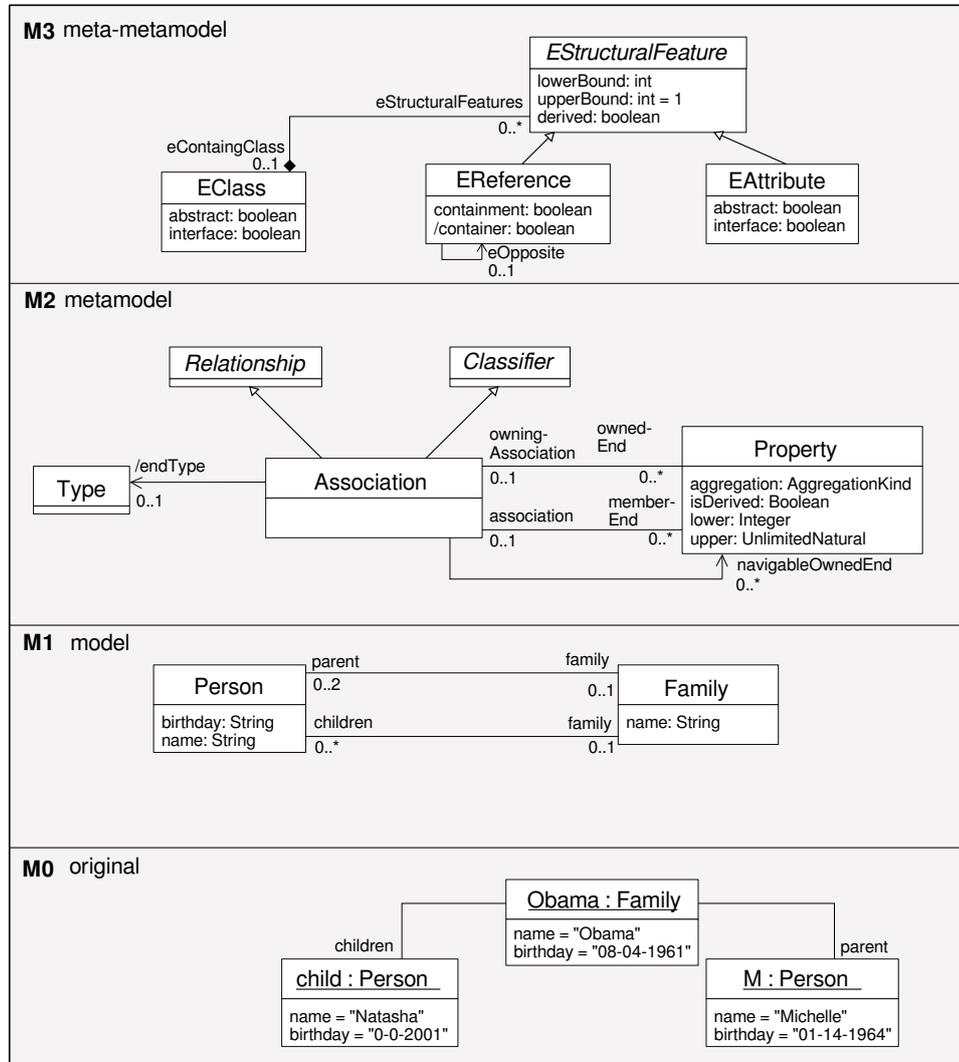


Figure 2.1.3: Modeling hierarchy for UML class models.

Despite the precise structure of this four-level metamodeling hierarchy to which further state-of-the-art OMG standards, such as the UML standard, refer, research has shown that multi-levels are closer to reality and ease modeling [AK08]. Multi-level programming distinguishes the linguistic from the ontological level of modeling which is beyond the scope of this thesis. In its recent update of the MOF standard, the OMG refrains from the strict four-level concept. The OMG denotes the instance-of relationship as the most relevant aspect of the four-level model. Specifically, the possibility to navigate from objects to their defining classes or, in different terminology, from instances to their classifiers represents its core contribution [Obj19b].

Concrete vs. Abstract Syntax To explain the relationship between metamodels and models more precisely, this paragraph distinguishes between the meanings of concrete and abstract syntax first.

The *model* reflects the original. If it is noted in *the concrete syntax*, it will instantiate the language constructs as they are defined in the metamodel. The *metamodel* defines the syntactic constructs,

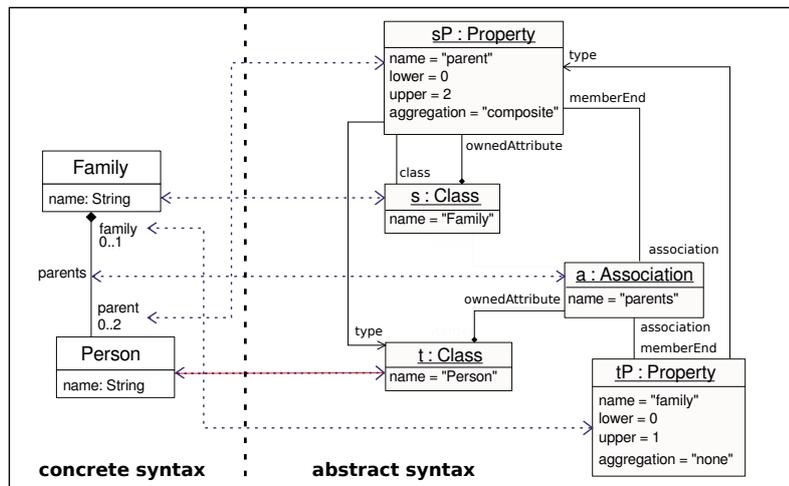


Figure 2.1.4: UML association in *concrete syntax* and in a simplified form in its abstract syntax. Fig. 2.1.3 depicts the corresponding metamodel elements at M2.

the relationships among and the static semantics, which allow to note the model. Noting the model in *abstract syntax* instead, shows which of the language constructs defined in the metamodel are instantiated in the model. Ex. 2.1.2 provides an example which demonstrates the relationship between concrete and abstract syntax. In the course of this thesis, several figures will note models in abstract syntax to demonstrate the syntactic constructs of the metamodels declared in the transformation specification.

Example 2.1.2: Concrete vs. Abstract Syntax

Fig. 2.1.4 exemplifies the differences between concrete and abstract syntax by taking up the example of the modeling layers. The left hand-side demonstrates a class **Family** consist of up to two **Person**. This model noted in concrete syntax conforms to the UML2 metamodel. It consists of two classes combined with a bidirectional association.

The right hand side shows relevant excerpts of the model noted in abstract syntax. Key element is the instance of the UML **Association** which is called **parents** and comprises the two member ends, **family** and **parent** which each are an instance of the UML **Property**. Furthermore, the class of the respective opposite end contains these properties, for example the class **Family** encompasses the property **parent**.

In this way, the right side contains the same information as the model on the left side but expresses it by instantiating the corresponding metamodel elements. Thus, it employs the abstract syntax.

Metamodel – a Modeling Language? A metamodel defines the *abstract syntax* and *static semantics* of a model thereby encompassing the syntactic constructs, their relationships as well as further constraints on them. Conversely, a *modeling language* does not only define the metamodel but all further levels necessary to formalize a programming language. Thus, the modeling language also defines the execution semantics of the metamodel and potentially further analysis methods, proofs, etc., too.

Unified Modeling Language The *Unified Modeling Language* (UML) and its extensions (e.g., *foundational UML* (fUML) and the *Action Language for Foundational UML* (ALF)), are some of the most frequently used standardized languages for building models of originals. The language originates from the Rational Software Corporation where it was designed by Grady Booch, Ivar Jacobson and James Rumbaugh [BJR96] to quieten the call for a unified notation to describe a system. The by-now-official UML standard, firstly proposed by the OMG in 1997 and latest

revised in 2017 [Obj17b], defines 14 kinds of models to express a system's structure and behavior. For designing the structure or the static parts of a system, the standard defines, for instance, *package models*, *class models* or *component models* whereas, for instance, *activity*, *sequence* or *communication models* serve to express the behavior of the system or some of its specific instances. Sec. 2.1.3, II illuminates the UML standard with respect to build class models in more detail.

Object Constraint Language The *Object Constraint Language* (OCL) [Obj14] issued by the OMG complements the MOF, by a subset, and the UML, by the full set of language constructs, respectively. The language allows to declare further constraints about their instances which cannot be expressed by the modeling language itself: For instance, UML or Ecore classes residing in one and the same package should not be given the same name represents a semantic constraint which is not expressible by the metamodels alone. As a formal language, OCL avoids the introduction of ambiguities due to constraints specified in natural language. OCL expressions are guaranteed to be without side effects and can be utilized to either query model elements or to declare constraints, such as pre-/postconditions on operations, invariants or the derivation of attributes.

2.1.4 Classifying Properties

Following above descriptions, this section outlines categories which allow to distinguish models and the design purposes.

Syntax Levels As it was explained in the previous section, *concrete syntax* is differentiated from the *abstract syntax*. While a model is typically noted in concrete syntax, a model noted in abstract syntax demonstrates which language constructs of the metamodel form the model.

Syntax Notation Models are commonly assumed to be noted as *diagrams*, most likely stemming from the fact, that many models are expressed in *graphical* syntax. For instance, classically, the UML specification defined the 14 model types as diagrams. However, recently, several of the UML model types can also be noted in textual syntax.

Accordingly, the way of *noting a model* is a second distinguishing factor for the syntax. Besides the graphical syntax, models can be written, for example, in a *textual* or a *tabular* notation. While a broader group of people may understand the graphical notation, the textual notation strongly relates with programming languages which may make it easier to understand and utilize by programmers.

Syntax Definition The model may be defined formally, semi-formally or informally. A formal model is grounded in a *formal modeling language*. Thus, it does not only conform to the abstract syntax defined in some metamodel but also respects the semantics of the modeling language and possibly further formal descriptions, such as proofs or inferences, for its constructs. In contrast, a semi-formally defined model conforms to a modeling language which is defined based on formal constructs combined with informal ones, such as natural language. An informally defined model adheres to informal descriptions only and, thus, is not considered a *model* by the means of the MDA. This thesis considers only metamodels which define the abstract syntax and static semantics as defined in the MDA.

Execution Semantics Besides the static semantics, the execution semantics describe the behavior at runtime for the language constructs. Similarly to defining syntax at different formal levels, the execution semantics can be defined informally with natural language or with (semi-)formal methods. In the context of modeling languages, a unique transformation into executable source code may declare the execution semantics of a modeling construct.

For instance, the UML standard does not prescribe the execution semantics for any of its model kinds. As a consequence, several ways exist, for example, to order multi-valued properties during integrating them into a class declarations. In contrast, for example, the OMG defines the execution semantics in natural language for BPMN Process Execution models [Obj19a].

Aspect: Structural vs. Behavioral Models In the context of MDSE we further distinguish the models by the aspect they cover. *Structural* models capture the relationships between entities of the system whereas *behavioral* models describe how an entity operates. Since the ultimate goal of MDSE is to turn the models into source code written in an GPL, a structural model upfront expresses how classifiers and packages are organized and allows to generate method stubs to establish relationships. In UML, for instance, class diagrams or package diagrams serve this purpose. In the Eclipse Modeling Framework the implementation of specific methods, preserving referential integrity, is generated completely. In contrast, *behavioral models* specify how the system *acts* in various situations. From behavioral models, e.g., the implementation of method bodies can be inferred. In UML, for instance, activity or sequence diagrams are categorized as behavioral models. More powerful are dedicated languages, like the ones specified in *foundational the subset for executable UML models (fUML)* [Obj18] and its instantiation in concrete syntax, the action language for foundational UML (**Alf**) [Obj17a]. fUML specifies precise semantics for a subset of UML which allows for generating executable source code. ALF is the standard providing a textual syntax on top of fUML, though ALF additionally supports structural modeling, its core is trimmed to express the behavior of method bodies.

Engineering Directions Finally, models reside on different levels of abstraction and the way and ordering in which they are built may depend on the system requiring a model. The three engineering directions of a system are sketched in Fig. 2.1.5. Since in Model-Driven Software Engineering (MDSE) each artifact in the development process should be modeled, we ideally refer to models at each development step, though it may be the case that, e.g., the implementation not only consists of an executable implementation model which is transformed into source code but of hand-written code as well. For that reason, the term “model” is put in brackets in the figure.

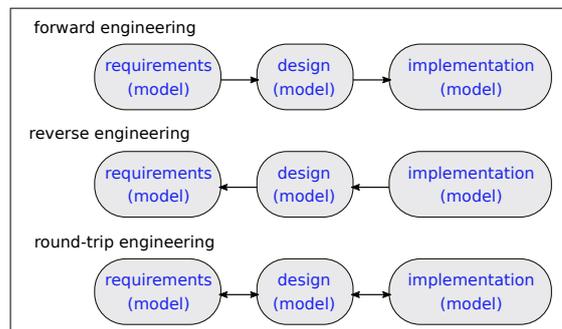


Figure 2.1.5: Classical software engineering directions.

Forward Engineering Classically, *forward engineering* adheres to the basic software development (*waterfall process*) [Roy87]: at first, an requirements model is created from analyzing the system requirements. This model is refined into a design model which, in turn, lays the foundations for the implementation. Since the models exist and describe the system before it is built, they are used in a prescriptive way.

Reverse Engineering On the contrary, in *reverse engineering* models are created only after the implementation already exists. As a consequence, based on the source code the implementation and design model are deduced. In the last step, the requirements of the system should be inferred. Since the implemented system exists before the models are derived, the models are used in a descriptive fashion.

Round-Trip Engineering Finally, in the real world, systems are usually subjects to steady evolution. In *round-trip engineering* the evolving factor is respected. As in forward engineering, the development process can be initiated in the forward direction by analyzing the requirements

and constructing the corresponding model, refining it into the design and thereafter, in the implementation. However, changes made to the source code or to one of the models in each stage can be played back in the opposite direction as well. Analogously, the process may as well start with the implementation as in the reverse engineering direction. The important aspect about the roundtrip-engineering is to orchestrate the synchronization of the models and the implementation as soon as any of the artifacts in the development process is modified. For automatically creating models based on existing models, the key technology are model transformations on which light is shed in Sec. 2.2.

2.1.5 Eclipse Modeling Framework

While above descriptions of the metamodeling foundations, in particular, of the MDA, MOF and UML, incorporate common concepts for metamodeling, the question arises how to transfer them into practice? As answer the *Eclipse Modeling Framework* (EMF) [Ste+09] has been built to realize these standards in the *integrated development environment* (IDE) Eclipse³.

At the core of EMF the *Ecore metamodel* complying to the Essential MOF (EMOF), i.e., a subset of the MOF, is located. As situated at level M3 of the MOF metamodeling hierarchy, it serves as meta-metamodel in EMF to build Ecore metamodels. Ecore metamodels are either represented as class diagrams or in a tree-structure from which applications can be built to express its models with a customizable tree-editor. EMF integrates a generator which transforms Ecore models, the instances of an Ecore metamodel, into Java source code. As the class diagrams only comprise structural elements, only methods stubs for modeled operations can be generated. Hence, the modeler needs to express the method bodies and can prohibit its deletion by either removing the annotation `generated` placed as comment before each source code fragment generated for a model element, or by renaming it into `generated not`.

2.2 Model Transformations

Several authors consider *model transformations* as “*the heart and soul of MDA*” [SK03] and “*the missing link in MDA*” [Ger+02] since almost 20 years. Informally, as depicted in Fig. 2.2.1, a function which receives (or *reads*) some source (model) input and transforms it into (*writes*) some target output can represent a model transformation. The source and target model are instances of a (not necessarily different) metamodel which may also be the language defining the source or target model. The *transformation specification* specifies the relationships of elements in the source metamodel and elements in the target metamodel and is executed by a *transformation engine*. [CH06]

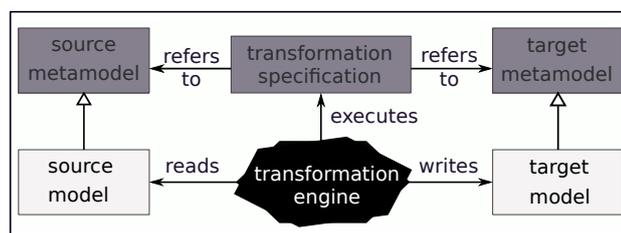


Figure 2.2.1: Schematic overview of a model transformation [CH06].

In MDSE, model transformations are the key technology for establishing the high degree of automation. In various situations model transformations may be exploited, upfront, when generating source code from a model by applying a *model-to-text* (M2T) transformation. In addition, creating a new model representation of an existing model (i.e., there is already (redundant) information encoded) manually, is an error-prone and laborious task contradicting the aim of raising automation in software engineering. In this case rather a *model-to-model* (M2M) transformation should

³ <http://www.eclipse.org/>

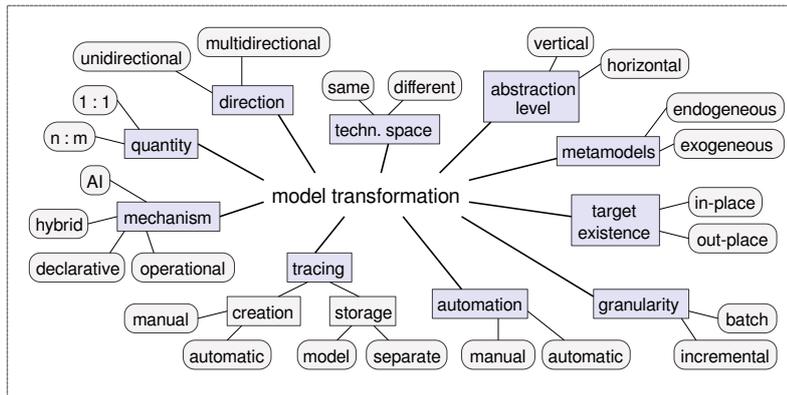


Figure 2.2.2: Classification of model transformations.

be applied. Likewise, a source in form of text can be turned into a model or into text by a *text-to-model* (T2M) or a *text-to-text* (T2T) transformation, respectively. On the whole, the input and output of the transformation may either be a source model or source text and a target model or target text, respectively. For that reason we refer to the input of the engine simply as *source* and to its output as *target* in all of the following explanations.

The first sections provide common features by which a model transformation (language) can be classified. Thereafter, concrete realizations are introduced.

2.2.1 Classification

By now, the research field of model transformations is largely populated and contains numerous different branches and supportive directions. Czarnecki and Helsen [CH03] presented a classification of model transformations in 2003 and extended the findings with a feature-based survey of model transformations in 2006 [CH06]. At the same time of the journal extension, Mens and Van Gorp published a “taxonomy of model transformations” [MG06]. While in Czarnecki et al.’s publications [CH03; CH06] the different features of transformation languages and tools are organized in a feature model, i.e., a tree structure with mandatory and optional features of a model transformation, Mens and Van Gorp [MG06] do not explicitly relate the terms of distinctive categories. In contrast, the authors focus on model transformations and their properties on a more superficial level whereas Czarnecki et al. detail additionally how model transformations can be noted and afterwards executed. In Fig. 2.2.2, we organize the most important and significant properties for this thesis using the terminology of Mens and Van Gorp at most. The categorization of transformation technologies, such as the facts how transformation rules may be specified or how they can be executed (e.g., scheduled), goes beyond the purpose of this chapter which aims to provide an overview of general transformation properties representing at least (still) vivid solutions.

I Mechanism

First of all, transformation languages may be differentiated by the kind of programming paradigm (mechanism), they apply. While Mens and Van Gorp distinguish *declarative* from *operational* languages, we have extended this classification to consider *hybrid* approaches as well as those based on *artificial intelligence* (AI), which have become investigated only recently.

Declarative Mechanism On their upside, *declarative* approaches allow to specify only relating elements of the source and target model. Such approaches abstract from explicitly navigating the source model, creating the target model and from specifying the order of executing the rules. Particularly, this category incorporates functional and logic-based transformation languages, which are explicitly distinguished in the survey of Czarnecki and Helsen [CH06].

Functional languages are natural to transformations since a transformation can be regarded as function receiving some input resulting in some output. As functions are first-class directives

in functional programming, a transformation can be treated as a model itself. However, it may become difficult to maintain the state of the transformation.

Besides offering a query mechanism, *logic* programming provides the benefits, for example, of backtracking or propagating constraints. Since functional and logic programming abstract from specifying the execution behavior and focus on relations, they are included in the declarative category. [MG06]

Graph-based transformations are not explicitly mentioned in above cited classifications. A graph rather is considered to be the source and/or target format or in- and output pattern of the transformation than an explicit kind of transformation mechanism. Nonetheless, it seems to be justified to mention graph transformations separately, which originate from specifying pair grammars to transform text into a graph structure (a model) and vice versa [Pra71]. Such graph grammars allow to express the language of all graphs of one kind and released a large field of research. Nowadays, in many graph transformation approaches the rules define a *left-hand-side* (LHS), being matched in the source graph, and a *right-hand-side* which is created in the target graph. *Negative application conditions* (NAC) specify when the rule is not allowed to be executed. As graph transformations express relationships or patterns of the source and the target instead of providing operational procedures how to create target elements, we add them to the subcategory of the declarative mechanism.

Operational Mechanism On their downside, entirely declarative approaches miss the functionality, for example to specify the order of creating target elements or assigning properties explicitly, which may be beneficial in many real-world transformation scenarios. In this case, *operational* mechanisms, allowing to specify, for instance the point in time, when the engine should execute a rule, become worthwhile. Hence, an operational approach is beneficial, e.g., whenever either the order of executing rules is essential or it is hard to identify and specify relations in a declarative way. [MG06]

Hybrid Mechanism As it tends to be more comfortable to avoid the overhead of, e.g., explicitly defining execution orders of transformations, but guarding the order of executing transformation rules may be necessary in certain scenarios, many languages provide a *hybrid* approach. These languages typically support expressing basic relationships in a declarative way. In addition, they either provide integrated procedural elements to gain control over the execution or integrate GPLs with well-developed standard libraries, e.g., for composing strings or navigating directories. As an example, the ATL transformation language allows to declare rules which are automatically matched to elements of the input source model but at the same time rules can also be called explicitly by other rules without the implicit matching which the developer has no control of.

AI-Based Mechanism Finally, the current revival of artificial intelligence and the capabilities of modern hardware have encouraged researchers in the field of model transformations to examine established AI techniques for transforming models. Burgeño et al. [BCG19] have only recently exploited specific kinds of neural networks to learn transformation specifications. Consequently, the successful derivation of the transformation specification requires training sets of corresponding source **and** target models. As such, its acceptance in and benefit to the model transformation community is still questionable since it fosters the existence of target models, incorporating each source-target relationship, beforehand. Besides neural networks, using logical programming to deduce a general transformation specification based on specific input is an established technique in artificial intelligence. Consequently, logic-based transformation languages can be regarded to belong to this category, too.

II Quantity and Directionality

Quantity Besides the mechanism, the number of instances which participate in the transformation differentiate the capabilities of transformation languages. *1:1* transformations only transform one source into one target. In contrast, several languages (at least theoretically) allow for transforming multiples sources into multiple targets (*n:m*). This encompasses as corner cases, *1:n*

transformations, transforming one source into multiple targets, e.g., when transforming one PIM into many PSMs, or *n:1* transformations, which, for instance, summarize many models in one superimposed model as a refactoring activity. [MG06]

Directionality Related with the quantity, the *directionality* classifies transformation language and specifications. In their easiest form, transformation languages only support *unidirectional* transformations. In the *forward* direction the source is transformed into the target whereas a backward transformation specifies how to create the source from a given target. In the case of *n:m* transformations, the direction is not obvious per se. While straightforwardly the *n* sources could be transformed into the *m* targets in a unidirectional transformation, other combinations of source and targets are theoretically possible. Consequently, *n:m* transformations should be *multidirectional* allowing to create different kinds of targets based on different kinds of sources. A *bidirectional* transformation in which the forward as well as the backward direction is covered can be regarded as a easiest form of a multi-directional transformation.

Regarding the engineering directions explained in the previous section, the application of a unidirectional forward and backward transformation realizes forward and reverse engineering, respectively. In contrast, bidirectional transformations are ideal whenever round-trip engineering should be accomplished and relevant in many different engineering disciplines [Cza+09].

III Target Representation

Many properties are specific to the type of target representation, namely, the overall technical space and the fact whether a transformation is horizontal or vertical, the kind of target metamodel and the physical presence of the target.

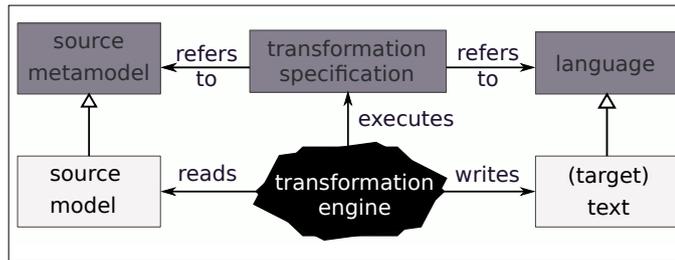
Technical Space First of all, most importantly the metamodels may reside in the *same* or *different technical space*. The technical space is defined based on the meta-metamodel (level M3), for instance, many model transformations reside in the standardized technical space of the OMG, the MOF, whereas XML is standardized by the *worldwide web consortium*⁴ (W3C). An actual transformation can take place in the *same* technical space. If it involves models of *different* technical spaces, one model must be imported for transforming it and the target exported afterwards. [MG06]

Abstraction Level Moreover, in line with the technical space we can distinguish a *horizontal* from a *vertical* transformation. In horizontal transformation source and target are situated at the same level of abstraction whereas a vertical transformation creates are target at a different level of abstraction. For instance, the transformation from an analysis model to a design model is categorized as vertical transformation as are refinements. Conversely, a refactoring exemplifies a horizontal transformation.

Metamodel As a further aspect, the target's metamodel is a distinguishing factor. In *endogenous* transformations the metamodel or grammar to which the source conforms is the same as the one of the target. For instance, specific refactoring rules can be applied to the source resulting in a new target which is still expressed in the same language. In contrast, an *exogenous* transformation generates a target representation conforming to a different metamodel than the source's one.

Physical Target Additionally, *in-place* transformations are differentiated from *out-place* transformations. An in-place transformation applies all changes to the source model so that it becomes the target model itself, as it may be the case, for example, in refactorings. In contrast, whenever the target resides in a different physical representation, we call it an out-place transformation. An exogenous transformation is always classified as out-place transformation because a physically different target is created. In endogenous this does not have to be the case necessarily: either the source model itself is updated to become the new representation, as it is the case of in-place

⁴ <https://www.w3.org/>

Figure 2.2.3: Schematic overview of $M2T$ transformations.

transformations, or a new physical target representation is created which, is expressed in the same language as the source, nonetheless.

IV Execution Modalities

Incrementality Moreover, we discern *batch* from *incremental* transformations. Batch transformations build the target representation in its entirety. In contrast, an incremental transformation requires that the target has already been created in a previous execution and applies only changes made to the source after the previous transformation to the target. Incremental transformations require a mechanism to detect the modifications applied to the source for propagating them to the target in the consecutive transformation execution. Such mechanism can be realized, for example, by recording changes or by comparing the previous state(s) with the current state. For detecting insertions, deletions, or modifications of source elements, many tools inspect trace information, which tracks corresponding source and target elements upon their creation.

Automation While the main goal of MDSE is to increase the level of automation, besides *automatic* transformations, there are transformations requiring *manual* interventions. In case the transformation language allows for ambiguities in its specifications which can not be resolved automatically interaction by the user may become necessary.

Tracing While Mens and Van Gorp do not go in further detail, Czarnecki and Helsens regard traces from the technical point of view. Firstly, traces are constructed either completely *automatically* or have to be *manually* maintained. Secondly, traces can be placed in different locations. On the one hand, a trace may be integrated in one of the participating models (source or target). This is beneficial since it does not require to specify and maintain any additional file. Alternatively, the trace could be stored separately, hence, realizing a clear separation of concerns between primary and secondary transformation artifacts. In this thesis traces play a key role and, thus, are discussed in greater detail in Sec. 5.2.1.

2.2.2 Transformation Languages and Tools

The classification properties allow to categorize transformation languages and their realizations as tools. We introduce an essential subset of transformation languages, which are relevant in this thesis. First of all, transformation languages are grouped into M2T, T2M and M2M transformations at a coarse-grained level. We do not present T2T languages because they are out of the scope of this thesis. T2M languages are shortly described because they are required in round-trip and reverse engineering scenarios, e.g., when creating an implementation model from the literal source code. Kahani et al. [Kah+19] provide an extensive survey and classification of 60 contemporary model transformation tools, which still not covers all available tools but, nevertheless, is taken into account for describing the languages and tools in this section.

I Model-To-Text Languages

In the first place, M2T transformations enable to perform the core automation step desired in MDSE, i.e., to create source code based on a model. In this case Fig. 2.2.1 describing a model

transformation, in general, can be refined by Fig. 2.2.3, describing the M2T transformation where the target is no model but text conforming to some language, in particular. Even though in MDSE M2T transformations serve to generate source code frequently, any other type of text can be generated as well, for example, configuration files, a format for serialization or a textual representation of the model (e.g., for textually comparing models).

Regarding the approaches to realize M2T transformations, Kahani et al. [Kah+19] distinguish *visitor-based* (also referred to as *procedural*) from *template-based* approaches. Visitor-based approaches write the text, e.g., by invoking methods for each model element while traversing the model. On the contrary, template-based approaches mix static text, which is written to the target in the same way as depicted in the template (*what you see is what you get* (WYSIWYG)), with dynamic text, inferred from the concrete source model. Due to the high similarity between the static elements in the template, and the outcome, template-based approaches tend to be easier to understand and, thus, are more popular [Kah+19]. As representatives of template-based approaches we introduce the standard proposed by the OMG, MOFM2T, and its implementation Acceleo as well as the template-based code generation language Xpand. Since some scenarios require more complex instructions, also *hybrid* solutions exist – as it is the case with model transformation mechanisms in general – e.g., including text generating templates in GPL methods or vice versa. As representative of a hybrid solution we introduce Xtend, which integrates Xpand.

MOF Model To Text/Acceleo The OMG standard for M2T languages is called *MOF Model to Text* (MOFM2T) [Obj08]. The specification suggests a template-based approach and resides in the MOF technical space, i.e., expecting Ecore models as input. Placeholders in the *templates* allow to query source model values which are turned into text by an expression language. In order to support large transformations, the templates are organized by *modules* where a *visibility* may restrict the access on the template. Moreover, a query can employ OCL expressions, e.g., to build complex Strings. Control structures may iterate over collections and branching is supported explicitly. The standard further defines an incremental mode by using *protected* regions, which are comparable to Ecore’s **generated not** annotation mechanism. Moreover, explicit traceability to the source element for which a text fragment was created could be integrated in the source code by using unique identification labels (ID’s).

The tool Acceleo⁵ realizes MOFM2T almost exactly. The grammar adopts the syntax from the standard and supports modules and queries. Upfront, the incremental mode protects only regions which would be generated in the same way as in the previous execution. In contrast to the OMG standard, the current state of Acceleo does not support tracing out-of-the-box.

Xpand In contrast to Acceleo, Xpand [Kla07; Eff+04] does not realize any standard but originates from the *openArchitectureWare* (oAW) project [EV06] where it serves as a primarily template-based M2T transformation language. Xpand projects typically consist of templates, extensions written in Xtend [Bet16] and a *modeling workflow engine* MWE(2) workflow. In the templates *blocks* specify how metamodel elements are converted into text. In these blocks plain text can be intermingled with other Xpand directives, e.g., to create a file or to invoke the execution of another block. Moreover, complex instructions can be written in *extensions* which provide a subset of the Xtend language, explained in the next paragraph. These extensions are also able to call Java methods. Most importantly for the present work, Xpand allows to extend or refine blocks of a template with an aspect-oriented [Kic+97] mechanism: in *advice* templates arbitrary input elements of the source model can be mentioned and redefined or embraced by additional text. The advice template is executed by triggering it in the MWE workflow. Similarly to Acceleo, the source code of designated *protected* region can be saved from being modified when executing the transformation again. This kind of incremental mode also requires unique IDs to identify already existing objects correctly.

Xtend Although the active development of Xpand has ceased, its key elements, i.e., the syntax of template blocks, survived in the GPL Xtend [ES; Bet16]. Xtend considers itself as “*Java with*

⁵ <https://www.eclipse.org/acceleo/>

*Spice*⁶, i.e., it is a Java dialect which compiles automatically into Java 8 source code. As a consequence, the statically typed language exploiting a more concise syntax than Java integrates with all Java libraries seamlessly. However, the object-oriented language involving also functional elements, like lambda expressions, not only allows to write arbitrary programs but is categorized as hybrid M2T language because of the integration of Xpand to generate text literally from a template-like style.

II Text-To-Model Languages

Above, we mentioned that in all engineering processes in the last step (or the first one, depending on the engineering direction) an implementation model is situated. However, in the end the model needs to be translated into source code to be understood by the compiler. When reverse engineering takes place, at first the current source code has to be transformed into an implementation model. Model-driven reverse engineering tools and especially round-trip engineering tools support this functionality by applying T2M transformations. As representatives, we introduce the tools JaMoPP and MoDisco, which both transfer Java source code into a Java model.

Java Model Parser and Printer The *Java Model Parser and Printer* (JaMoPP) [Hei+09] was built for “*closing the gap between modeling and code*” [Hei+09]. As its name suggests, JaMoPP “parses” the static structure of Java files into its own Java metamodel. Thereafter, the metamodel can be again “printed” as Java code. Since JaMoPP only covers the static structure, method bodies are attached to metamodel elements representing the Java methods as plain text in annotations. The Ecore-compliant parser and printer are realized based on Java 5 and are not maintained anymore. Nonetheless, they are particularly used in model-driven software product line engineering. Although the Java metamodel does not reflect the state-of-the-art of the Java language, the framework can still be integrated in current Eclipse versions.

MoDisco Similar as JaMoPP, the MoDisco [Bru+10] framework is a Eclipse plugin to reverse engineer implementation models from source code. In the first place, MoDisco, like JaMoPP, provides an Ecore-compliant Java metamodel but is based on Java 6. MoDisco offers the extensible means to *discover* the contents of an Eclipse project in general and of Java source code in particular, i.e., transforming Java source code into an instance of the metamodel. In turn, a code generator transforms an instance of the metamodel into Java source code. Thus, the framework supports round-trip engineering between the model and the source code which only works in batch mode. The active support for the framework has discontinued at the end of 2018.

III Model-To-Model Languages

As it was stated above, the transformation mechanism can either be declarative, procedural, AI-based or a hybrid mix. Kahani et al. [Kah+19] consider graph-based approaches, which we classified as declarative approach, an own category.

In 2002, the OMG initiated a request for proposals for a *query view transformation* (QVT) language in the MOF, transforming up to n models into up to m models. Many languages and realizing tools have been developed as answer to this request. In the following, as declarative languages, we introduce QVT-R which is also part of the OMG’s QVT standard for M2M transformations, and ATL, a hybrid but primarily declarative language. QVT-O is picked in order to represent an entirely procedural one.

Furthermore, BXtend [Buc18] exemplifies a hybrid but graph-based transformation framework. Further Graph-based transformation tools are, for instance, **Henshin** [Are+10a; Str+17; Str+18a] **eMoflon** [LAS14; LAS15] and **AGG** [Tae99; Tae03; RET11], which share the fact that the left-hand side is turned into a right-hand as specified in rules. *Tefkat* [LS05] exemplifies a logic-based language which derives an execution environment from the logical rules.

⁶ <https://www.eclipse.org/xtend/>

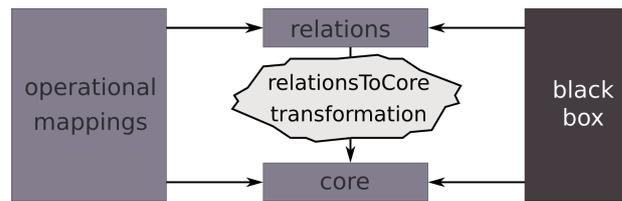


Figure 2.2.4: Parts of the MOF 2.0 Query View Transformation standard. Adapted from [Obj16]

Atlas Transformation Language The *Atlas Transformation Language* ATL [Jou+08] and the similarly named tool (provided as Eclipse plugin) is one of the answers to the OMG’s QVT request. ATL is designed to specify unidirectional, n:m transformations based on transformation rules. While *matched* rules are declaratively executed, i.e., source elements matching the input pattern are transformed, *called* rules are invoked by other rules, thus, allowing for ordering the rule execution in a procedural way. Moreover, although the batch mode works as out-place n:m transformation, an incremental mode (called “refining mode”) exists which is restricted to in-place 1:1 transformations for refining or refactoring existing targets.

ATL executes the transformation in two phases. Firstly, all target elements are created, allowing to correctly set references and attributes in the second phase making use of traced elements of the first phase. Albeit a trace is written during the execution, it is not persisted by the tool. An extension to ATL, the virtual machine based ATL/EMFTVM [Wag+12] includes the functionality to store traces after the execution and provides the transformation as bytecode model, too. However, its transformation capabilities are limited compared to ATL. For instance, an target element’s collections can only be assigned a value once. Helpers, which act like queries in Aceleo and are supported in ATL, are limited to assigning static attribute values and not to dynamically fill or invoke operations, e.g., for composing strings, in the ATL/EMFTVM. In contrast to many other transformation tools, the ATL and ATL/EMFTVM Eclipse plugins are still vivid.

MOF 2.0 Query/View/Transformation The OMG issued the *MOF 2.0 Query/View/Transformation* (QVT) language as answer to the request and as standard for establishing a multidirectional transformation language in the context of the MDA. As depicted in Fig. 2.2.4, QVT [Obj16] is divided into three main sub-languages: *QVT Relational* (QVT-R), *QVT Operational Mappings* (QVT-O) and *QVT Core* (QVT-C). While QVT-R and QVT-C are entirely declarative, as the name suggests, QVT-O creates the target in a procedural way. Since QVT-C is a declarative language residing at a lower level of abstraction than QVT-R and derivable from QVT-R, we refrain from describing it.

QVT-R allows to specify relations between up to n elements of different models and to influence the execution order by pre- and postconditions (which can be considered as assurances). The standard prescribes two different execution modes. On the one hand, the *checking* mode verifies the consistency of (already existing) participating models. For instance, a target name not corresponding with the one inferred from the corresponding relation, would be reported to be inconsistent. On the other hand, such inconsistencies are repaired automatically by executing the *enforce* mode. Since each domain in a relation (i.e., being part of up to n different models) can be marked to be enforced, multidirectional rules can be expressed in this way. Consequently, the number of real execution modes depends on the participating number of enforced directions. For instance, in a *bidirectional* transformation rules incorporate at least two domains, of source and target elements, respectively, each being enforced. Therefore, in both directions the checking and enforce mode is possible. Lastly, incremental transformations complete the OMG’s model transformation standard. Although in these transformations the standard suggests to execute the checking mode for detecting inconsistencies between the new source and the old target, many tools, realizing the standard, technically exploit trace information for detecting changed target elements.

Realizations of the standard are **medini QVT** [ikv18] and the most-recent Eclipse plugin **QVT-d⁷** [Wil17]. ModelMorf realized the first version of the QVT standard almost completely but is

⁷ [https://wiki.eclipse.org/MMT/QVT_Declarative_\(QVTd\)](https://wiki.eclipse.org/MMT/QVT_Declarative_(QVTd))

unavailable by now. It has been extensively utilized to examine the standard by Stevens et al. [Ste10; BS13]. Likewise, the active development of medini QVT stopped in 2012 but is in contrast to ModelMorf still available. medini QVT realizes incremental and bidirectional transformations based on persistent traces but misses the explicit checking mode. Finally, QVT-d is a young tool which still undergoes significant functionality changes in each new Eclipse release. QVT-d also persists trace information and does not yet support the whole amount of the QVT standard, in particular, incremental and bidirectional transformations are still not executable. [GBW16]

QVT Operational Besides the declarative transformation language QVT-R, the OMG's QVT standard encompasses the operational language *QVT Operational* (QVT-O) as well. In QVT-O a transformation is structured like a class (e.g., in UML) with properties and operations which declares mappings between source and target elements. QVT-O only supports unidirectional transformations and incorporates OCL statements with side effects. A rudimentary trace (without configuration, intermediate transformation data, nor deep copies of the involved model elements) is written during the transformation and can be used for incremental transformations (of the same source instance). [Obj16]

BXtend As a hybrid approach BXtend [Buc18; BG16a] provides the means to automatically create a *triple graph transformation system* (TGTS) [BDW08] for bidirectional, incremental transformations without explicitly specifying a *triple graph grammar* (TGG) [Sch94]. Besides the LHS and the RHS a TGG involves a third graph in between called *correspondence graph* to link elements of the LHS with those of the RHS. The correspondence graph enables to synchronize the generation of the LHS and the RHS. A TGTS resides on a lower level of abstraction and does not only incorporate production rules but consistency mechanisms to handle modifications and deletions in model synchronization scenarios. Such system is supposed to be generated from a TGG automatically. However, in practice it is hard to specify complex synchronization tasks without handcrafting fine-grained maintenance actions. For that reason, Bxtend was developed. In Bxtend two unidirectional rules form a bidirectional one. Due to the usage of Xtend, its functional syntax constructs, such as lambda expressions, intermingle declarative elements with operational ones, e.g., for prescribing the execution order. Consistency maintenance in incremental transformations is supported by exploiting the correspondence graph.

Chapter 3 Software Product Line Engineering

“Although a feature model can represent commonalities and variabilities in a very concise taxonomic form, features in a feature model are merely symbols.”

Krzysztof Czarnecki & Michal Antkiewicz [CA05]

Inspired by classical engineering sciences, where mass customization established itself by developing families of related products in a *product line* (well-known in the automotive industry), research and practice adapted the concept for creating *software product lines*. As such, *software product line engineering* is one of the answers to solve the software crisis’ problem of managing large-scale and (in this case) software-intense projects. By highlighting common and variable parts of closely related products, the discipline software product line engineering emphasizes the principles of *organized reuse* and *variability*: Parts, common to all products, should be reused to build a diverse set of product *variants* whereas the varying parts should be managed in an organized way.

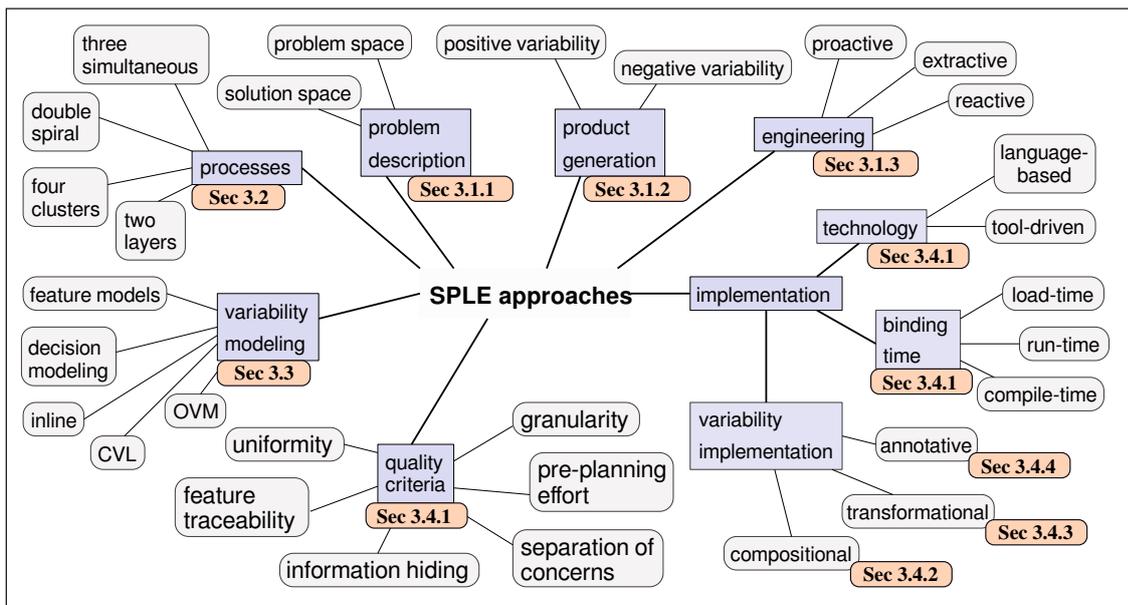


Figure 3.0.1: Overview of software product line engineering.

Fig. 3.0.1 summarizes the key contents introduced in this chapter. The chapter starts

by presenting basic terminology in Sec. 3.1, such as terms for analyzing the system that should be realized (*problem description* in Sec. 3.1.1 or the product line engineering in Sec. 3.1.2 and Sec. 3.1.3), and offers insights in engineering processes (Sec. 3.2), modeling the variability (Sec. 3.3) and implementation details (Sec. 3.4) in the following sections. In the end, the chapter closes with elaborating on product well-formedness, one key correctness criteria of a product line, and recapitulating notes.

3.1 Terminology

Before diving into the engineering principles, this section clarifies and presents basic terms and activities in the context of Software Product Line Engineering (SPLE). Initially, probably the first term used for what we (now) refer to as product line is *family of systems* [Par79]. Whenever these systems share more commonalities – so that it becomes worthwhile analyzing these systems as a whole – than distinguishing factors, developing and modeling the entire *domain* pays off. In a same manner a product line can be named *software product family*. In accordance with basic literature on this topic [Gom05], we regard the terms *software product line*, *family of systems* or *software product family* as synonyms and utilize the shortened term *product line* in the following descriptions.

3.1.1 Product Line Analysis

When commencing the development of the product line, the *scope* of the product line needs to be delineated. Typically, a *product portfolio* describes details of the products that should be supported, particularly their distinguishing and common aspects. This document incorporates – besides the technical – different perspectives, e.g., the economic point of view of selling the products. The development steps and set of products are grounded in the product portfolio which documents the needs of various stakeholders. Most importantly, the portfolio designates the *domain* of the system. According to Czarnecki and Eisenhower [CE00], a domain declares the scope of the product portfolio so that it satisfies all stakeholders’ interests maximally by utilizing the concepts and terminology common to practitioners and including the knowledge how to build a system in this domain. [Ape+13]

Problem and Solution Space Fundamentally, the *problem space* needs to be differentiated from the *solution space*. While the problem space systematically recognizes and documents the common and varying parts of the system, the solution space provides the realization artifacts of the documented product line functionalities. In the problem space a *variation point* describes a point in the product portfolio where the product characteristics diverge resulting in different *variants*. Conversely, in the solution space variation points are implemented by using configuration techniques. A variant corresponds with a product which is derived by exploiting the configuration techniques. Technically, problem and solution space are quite often connected by a mapping tracing the realization artifacts relating to the abstract problem description.

Platform Realizing the product line domain in the solution space results in a software *platform*. The platform is considered “*a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced*” [ML97]. As such, the platform lays the grounds for creating a diverse set of products (of a product line).

Feature Modeling During the analysis, *features* manifest the varying and common parts of the product line. In the first place, a feature was described as “*a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems*” in the context of the *feature-oriented domain analysis (FODA)* [Kan+90]. Further definitions of a feature were provided in the course of years and were once collected by Classen et al. [CHS08]. This collection of definitions spans a range of abstractly to technically explaining the meaning of a feature. Despite these 13 gathered definitions, for the purposes of this thesis the initial definition by Kang et al [Kan+90] suffices at the most. It could be extended by a more technical perspective, such as describing a feature as “*a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder’s requirement, to implement and encapsulate a design decision, and to offer a configuration option*” [Ape+08]. In particular, the focus on the technical aspect of localizing the realization of a single feature used as one option when configuring a product should be highlighted because it is a prominent view in *feature-oriented programming (FOP)*, a summarizing term for constructing product lines by localizing and focusing on their features. Upfront, a feature describes a binary option whether a characteristic is present in a product. The reality, however, exhibits a

higher degree of complexity. Quite frequently, dependencies between features exist and manifest as *feature constraints* [Ape+13]. For instance, selecting a wireless connection mechanism in a home automation system *requires* that at least the IEEE 802.11a standard is present in the routers. Likewise, if a finger print scanner unlocks the front door no other mechanism, like a key-card, can unlock the door, i.e., feature fingerprint *excludes* feature key-card. Several ways how to capture variability, in particular as variability models, are detailed in Sec. 3.3.

Variability in Space and Time Lastly, since variability is the essential concept in SPLE, it must be remarked that SPLE concentrates its focus on *variability in space* upfront. Variability in space refers to the fact that artifacts may exist in different forms at *one point in time*. In contrast, the concept *variability in time* refers to the evolution of a system, i.e., one artifact exists in different forms at *different points in time* [PBL05]. The research discipline *software configuration management (SCM)* [Ber84; Fel91] examines and optimizes the support for variability in time (besides aiding programming in large teams) extensively. However, managing versions of software over time, i.e., extensive support for variability in time, is not the focus of this thesis.

Summary All in all, SPLE stresses the explicit focus on *variability to reuse* parts of the created artifacts in a systematic, organized way for building a product line.

3.1.2 Product Generation

For realizing the product line, at first, a coarse-grained variability mechanism needs to be chosen¹. *Positive variability* builds one core product including elements which are ideally present in each product. The core product is extended by adding new artifacts or changing the core artifacts. A new customized product is assembled by specifying which elements need to be added to or changed in the core product. In contrast, *negative variability* means that the entire set of products is created in a *superimposition*. Consequently, in this approach products are not *assembled* but elements not belonging to the product are *removed* from the superimposition.

Feature Configuration In order to *derive* (also denoted *filter*) a customized product, a selection of the desired features needs to be provided, which is commonly referred to as *feature configuration*. A *valid* feature configuration respects all constraints which are documented in the variability model. In turn, the derivation tool should ensure that a *consistent* product is derived, i.e., resulting in a syntactically and ideally a semantically correct product.

Domain and Application Engineering In this context the distinction of *domain engineering* from *application engineering* should be stated which is a general concept for SPLE processes [WL99; Lin02; PBL05; Ape+13]. During domain engineering the platform is built in addition to recording and analyzing variability. Consequently, domain engineering realizes “*development for reuse*” ([Ape+13], p.21). In contrast, during application engineering single products are finalized based on the artifacts produced in the phase of domain engineering. Application engineering is comparable to single application development except that it explicitly reuses the already created artifacts of the domain engineering phase, thus, realizing “*development with reuse*” ([Ape+13], p.21). In contrast to domain engineering, which is a holistic process for *all* products, application engineering is performed for *each* derived product.

3.1.3 Engineering Strategies

Apel et al. [Ape+13] distinguish three main types of engineering a product line: the proactive, extractive and reactive development. As stated in the introduction (c.f., Sec. 1.3), the contribution of this thesis support proactive development, which is explained first, in the following paragraphs.

¹ Choosing a variability mechanism freely may be possible only restrictively if single products already exist. In this case, the mechanism must be aligned with its environment.

Proactive Development Depending on the company's strategy and settings, product lines are (or have to be) developed differently. Building a product line in a *proactive* way means the product line is developed from scratch. The company probably considered a new family of products in its product portfolio without any existing artifact yet. Obviously, in this case the entire platform is developed from scratch and the products are derived from the platform afterwards.

Reactive Development Conversely, frequently a set of related products already exists in a company. Many of these products may be traced back to an initial product which was copied and thereafter modified to fit specific customers' needs. An *extractive* SPLE approach aims at identifying commonalities and differences among the existing products and deriving a variability model and platform to ease maintenance of the copied products. It is an iterative process limited to the information encoded in the existing products.

Reactive Development Thirdly, a *reactive* SPLE approach initiates the development with one core product line. Incrementally, the new functionality of a not yet envisioned variant is added to the base product line. Consequently, this approach does not require as much pre-planning as the proactive approach and, thus, is more agile. On its downside, it may become cumbersome to add a new variant not at all foreseen so far. Nevertheless, developing a product line in a reactive way is a more structured process than maintaining existing products in an extractive way. [Ape+13]

3.2 Development Processes

As the field of research on SPLE is largely populated by now, different processes exist to develop product lines in an organized way. Particularly, the complexity to manage the entire set of products systematically fosters a structured development strategy, too. For that reason, this section introduces four fundamental processes which can be applied to systematically build a product line.

3.2.1 Three Simultaneous Activities

To the best of our knowledge, the first process dedicated to develop a product line in an organized way was proposed by Clements and Northrop in 2001 [CN01]. The process consists of the following three activities, which should be applied in parallel: *management*, *core asset development*, and *product development*. Each activity is linked to each other and circularly repeated continuously. Management splits in two parts: While *technical* management ensures the production of core assets by measuring the product development and applying defined processes, *organizational* management ensures the success of the product line by providing the necessities for the technical aspects, like funding or adequate personnel.

Core asset and product development are stronger intertwined. Based on the product constraints, the production constraints, the production strategy, the architectural elements (like patterns or frameworks) and the inventory of preexisting assets, the core asset development activity delineates the scope of the product line. Moreover, the core assets which represent all elements present in each product including, for example test and planning artifacts as well as a production plan, emerge from the core asset development. Finally, the product development obtains the production plan, the core assets, the specifics of one particular product as well as the scope of the product line to create this product. Due to the concrete implementation, it may be possible that the scope of the product line has to be adapted and, hence, the product development may influence the development of the core assets in turn.

3.2.2 Two-Layered Process

The process closest to the classical (iterative) waterfall process is proposed by Pohl et al. [PBL05]. The authors discern the two layers, *domain engineering* and *application engineering*. Prior to starting the product line development in the layer of domain engineering, a *product management* step

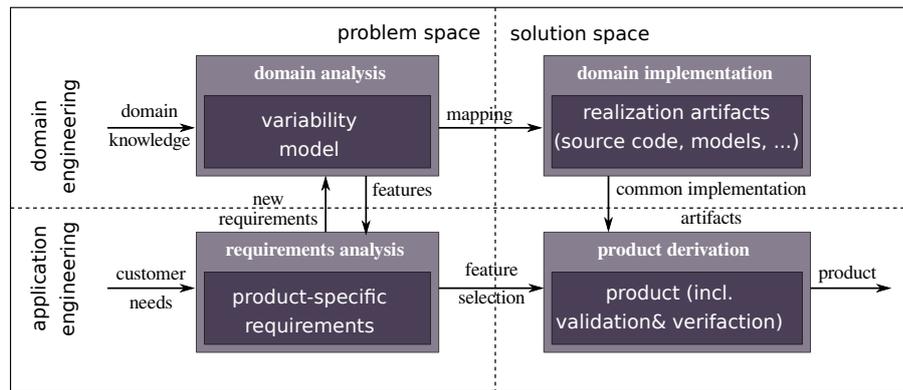


Figure 3.2.1: Four-clustered process to develop a software product line proposed by Apel et al. [Ape+13].

where the scope of the product line is defined, is undertaken. In this step, besides others, the economic aspects of the product line are considered. Thereafter, the steps *domain requirements*, *domain analysis*, *domain design*, *domain implementation*, and *domain testing* are applied in sequence in the phase of domain engineering, These development steps culminate in a *variability model* and the *platform as domain artifacts*. Particularly, the platform subsumes all realization artifacts which may be required in the varying products.

The second layer, the phase of application engineering, receives all domain artifacts and refines them in the same sequence of activities. Thus, after collecting the *application requirements*, the *application analysis*, the *application design*, *application implementation*, and *application testing* are conducted to assemble the final application. In the same way as the waterfall process can be executed iteratively, the steps in domain and application engineering may repeat after performing one complete development cycle. [PBL05]

3.2.3 Double Spiral Model

A process which incorporates a risk-driven viewpoint is proposed by Gomaa [Gom05]. The process adjusts Böhm’s spiral model [Boe88] consisting of the four steps (1) “*define objectives, alternatives, and constraints*”, (2) “*analyze risks*”, (3) “*develop (product)*”, and (4) “*plan next cycle*” to be separated into two spirals, one conducting the steps for the complete product line and one conducting them for the individual product. During the development many instances of the spirals may exist and because a product and the product line may evolve simultaneously, the spirals may be tightly intertwined. Accordingly, information obtained by processing one spiral feeds the other spiral and vice versa.

3.2.4 Four-Clustered Process

Finally, Apel et al. [Ape+13] propose a process which adapts the previously explained ones and trims them to concrete dependencies in between domain and application engineering for developing feature-oriented software product lines. The corresponding process model, depicted in Fig. 3.2.1, divides the engineering tasks horizontally in *domain* and *application engineering* tasks (as in the classical two-layered process by Pohl et al.) whereas vertically the *problem space* is separated from the *solution space*. At first, at the level of domain engineering the *domain analysis* takes place based on *domain knowledge* in the problem space. This step decides which products are covered. Thus, it delineates the “*scope of the domain*” ([Ape+13], p.21) which results in a *variability model* (the authors originally propose a feature model). The analysis artifacts are mapped onto the *domain implementation* which occurs in the solution space at the same level. It collects all realization artifacts, such as design models, source code, and tests, which together constitute the platform.

At the level of application engineering, a *requirements analysis* is performed in the problem space based on *customer needs*. Ideally one customized product is related to one particular feature

configuration. If this is not the case, a requirement not yet covered during the domain analysis, will be propagated to the level of domain engineering and integrated in the variability model. Consequently, the requirements analysis may interact with the domain analysis by adding new requirements and receiving the features of the variability model in turn. The *product derivation* finalizes the process. At this stage, common implementation artifacts are taken from the domain implementation according to the feature selection of the requirements analysis. Moreover, this last section involves the validation and verification of the eventually delivered product. [Ape+13]

3.3 Variability Modeling

This section illuminates the most common forms of *variability models* in the context of SPLE. Sec. 3.3.1 initiates by presenting *feature models* followed by Sec. 3.3.2 explaining further types such as *decision models*, the *orthogonal variability model*, the *common variability language* and variability models integrated in UML 2.0.

3.3.1 Feature Models

Basics For the first time, in 1990 Kang et al. [Kan+90] proposed to note *features* in the form of *feature models* to alleviate the *feature-oriented domain analysis (FODA)*. A feature model captures all common and varying characteristics discovered during the domain analysis as features. Typically, feature models are manifested graphically, i.e., as *feature diagrams*, which are trees with a feature root (typically mentioning the system to be built) and inner nodes building *feature groups*. *Mandatory* features form part of every product whereas *optional* features can be almost freely integrated in products. A feature group enumerates a number of child features which are either aggregated in **OR** (inclusive) or **XOR** (exclusive) groups. An **XOR** group fosters that exactly one of the features in the group can be selected at the same time. An **OR** group, instead, allows to select at least one of its grouped features. In each parent-child relationship of a feature model, the existence of a child depends on the existence of its parent. Further relationships of features can be expressed by declaring *requires* and *excludes* dependencies between the features, so called *cross-tree constraints*. Feature models, offering these properties, can be translated completely into propositional logic. For instance, the *requires* and *excludes* dependencies represent a logical implication (\Rightarrow) with a positive and a negated feature on the right side, respectively.

Feature Configuration To derive a product a *feature configuration*, selecting or de-selecting (all) features, needs to be provided. A feature configuration will be *valid* if no constraint of the model is violated. The number of valid configurations increases exponentially in relation to the number of optional features included in the feature model. Analogously, the number of valid configurations shrinks by introducing constraints or mandatory features. In contrast to a *complete feature configuration*, a *partial feature configuration* does not provide *all* features of the corresponding model with a selection state. Some selection states can simply be inferred, e.g., when de-selecting a parent feature none of its children should be selectable. Depending on the capabilities of the feature configuration tool, partial configurations can be supported and selection states be propagated. If not all feature selection states are known, either the derivation is not possible or similarly only a partial product or a product with heuristically determined elements is created.

Example 3.3.1: Feature Model and Configuration

The left hand side of Fig. 3.3.1 exemplifies a basic feature model which allows to configure databases with different contents^a. The root of the feature diagram typically mentions the systems to be built, in this case database contents (**DBContent**). Since all of the root's child features, i.e., **Person**, **Family**, **Animal** and **Media**, are optional, potentially an empty database could be built. Despite this fact, it is possible to create a database including all child features, too, since all groups are optional. If a person database is desired, a person

must carry a name at least (which is a mandatory feature). As an example of an XOR group we prohibit that a database registers wild and domesticated animals at the same time. All of the other feature groups are OR groups allowing an arbitrary number of the child features to be chosen. Two constraints require that pets can only be selected if domesticated animals, and families can only be present if relations are recorded, respectively.

The right hand side of Fig. 3.3.1 demonstrates two configurations of the feature model. The configuration in the upper left side is valid and selects a database consisting of families and songs, nothing else. The second configuration, instead, is invalid because it violates the cross-tree constraint "Pet requires Domesticated". The configuration includes pets but does not select animals, resulting in the invalid configuration.

^a This feature model may be necessary when an object-relational mapping of code to database contents should be established.

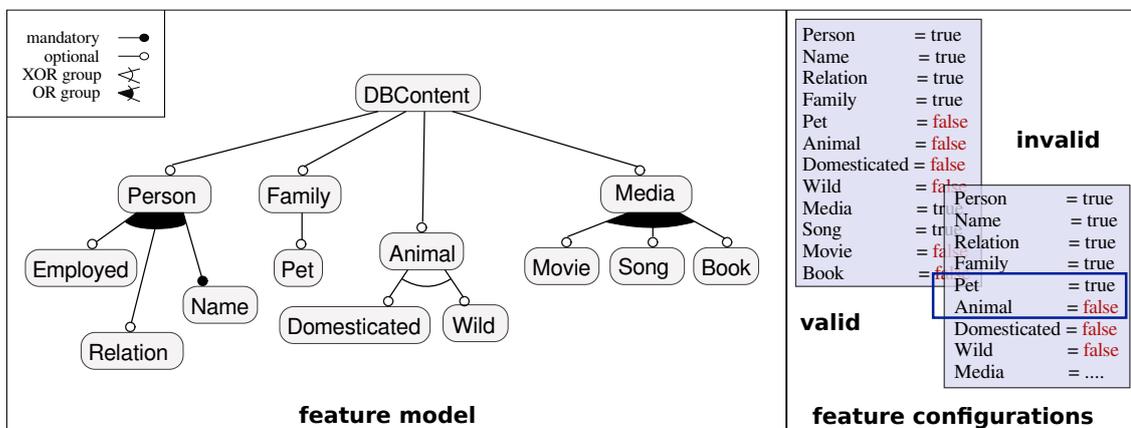


Figure 3.3.1: Feature model for *database contents*.

Extensions Due to a missing standard for variability models, many works have extended these basic descriptions of feature models. A survey and formal definition of feature models has been published by Schobbens et al. [SHT06]. As examples of extensions, *cardinality-based feature models* [CHE05] can restrict, how many of the grouped features can be selected, and *abstract* or *attributed* features can enrich the expressiveness of the feature model. Additionally, more variability information, particularly regarding the evolution of a system, could be integrated such as temporal variability in *hyper feature models* [SSA14a] or *contextual variability* which considers influences by the environment during runtime [Ape+13].

3.3.2 Further Types of Variability Models

Besides feature models, decision models, the Orthogonal Variability Model and the Common Variability Language play a role in academia and industry to express variety in the product portfolio.

Decision Models Quite like feature models, *decision models* [SRG11], originating from the *Synthesis* method [Con91], continue a long tradition of maintaining variability, particularly of industrial applications, in a systematic way. All realization approaches of decision models, such as the one by Schmid et al. [SJ04] or DOPLER [DGR11], reflect the variability in the problem space as *decisions* by taking their dependencies into account and explicitly mapping the decisions to reusable assets realizing the variability in the solution space. Instead of modeling the differences and commonalities as in feature models, decision models capture "only" the differences, i.e., the variability of the system. [Cza+12]

Orthogonal Variability Model In contrast, the *Orthogonal Variability Model (OVM)* [PBL05] explicitly documents variation points and the corresponding variants by *mandatory* and *optional* as well as *excludes* and *requires* dependencies among variants and variation points. The OVM emphasizes the orthogonality of documenting variability. Capturing the variability in this independent model, i.e., in an orthogonal way to other realization artifacts, offers its accessibility across all development artifacts (as is the case with feature models, too). Other approaches, such as the one by Gomaa et al. [Gom06], integrate the variability information explicitly in the UML artifacts hindering a cross-artifact usage. In Fig. 3.0.1 we categorize this kind of variability modeling as *inline*. [Cza+12]

Common Variability Language Lastly, in 2009 the OMG issued a request for a variability language standard. By now, this request for proposals is not accessible anymore. The *Common Variability Language (CVL)* [Hau+08] answers this request as a forerunner. Particularly, the language tries not to use already established wording, such as the term *feature* (the meaning of which varies all over computer science disciplines, e.g., in image processing it describes a characteristic in pictures). Since the CVL (framework) relies on transformations of a base model, details are explained in Sec. 3.3.2, II.

3.3.3 Endnotes

Summing it up, modeling variability is implemented in many different ways and levels of detail. Trying to establish the CVL as standardized language by the OMG has not been successful yet due to many already existing diverse realizations and a still open field of active research (e.g., conducted in form of open contemporary workshops such as the *modeling variability (MODEVAR)* series [Ben+19],[Ach+20]). As a consequence for this thesis, we pick the feature model for representing the commonalities and differences of our product lines as it is one of the most wide-spread variability models with sound tool support. A feature diagram involving the basic elements, i.e., mandatory and optional features, OR and XOR groups as well as requires and excludes cross-tree constraints suffices to describe the product lines considered in this thesis.

3.4 Variability Implementation Techniques

For implementing a product line in a proactive way (i.e., from scratch) several possibilities exist which are explained in this section. The majority of the following descriptions is based on Apel et al.'s summary of techniques [Ape+13]. The book presents (only) techniques to establish *feature oriented programming (FOP)*, i.e., realizing a product line based on capturing variability in form of a feature model, particularly focusing on how to implement single features and composing them in an organized way. Another taxonomy for categorizing the implementation of variability mechanisms was proposed by Svahnberg et al. [SGB05] which includes many technical details which we do not regard in order to rather give an overview of the general realization mechanisms. Furthermore, this section does not elaborate on classical tool-driven variability mechanisms, such as using *version control systems (VCS)* to develop products in different branches (*clone-and-own approach*) or using build systems as they are out of the scope of the thesis.

After elaborating on basic concepts when realizing variability-intense software in Sec. 3.4.1, this section introduces a *compositional* (Sec. 3.4.2) (including *aspect-oriented*), a *transformational* (Sec. 3.4.3) and an *annotative* (Sec. 3.4.4) approach to realize a system with common and varying features.

3.4.1 Basics

The description of variability implementation techniques initiates in this section with illuminating general distinctions between implementation strategies, first, and considering quality criteria afterwards.

I General Strategies

Binding Time Implementation techniques vary in two aspects: the time they bind variability and the employed technology. The variability can be bound early or late in the product development. Apel et al. [Ape+13] distinguish *compile-time* from *load-time* and *run-time* binding. Deciding early (i.e., before compiling the product) which features are part of the products implies that no other functionality is delivered to the customer. In contrast, load-time and run-time binding postpones the choice which functionality to include in the product to setup time or even to run-time, respectively. Consequently, all variability is – though hidden – part of the shipped product. While compile-time products require less resources and do not allow to detect functionality of different products, products bound at load- or run-time promise to be more flexible and easy to reconfigure. [Ape+13]

Language- vs. Tool-based Furthermore, product lines can be developed based on dedicated languages (*language-based*) or based on specific tools (*tool-based*). While in a language-based approach all variability information is placed in the source code and, thus, the variability management is also organized based on the information in the source code, a tool-based approach by separates the feature implementation from the product derivation and the variability management by using an external tool. Typical examples are the usage of runtime parameter as a language-based and the usage of preprocessor as a tool-based approach. However, many times both types of feature implementation support are intertwined or combined to some extent. [Ape+13]

II Quality Criteria

Overview To evaluate the benefits and shortcomings of implementation approaches, Apel et al. [Ape+13] install the following six quality criteria: *separation of concerns*, *uniformity*, *feature traceability*, *information hiding*, *pre-planning effort*, and *granularity*.

Separating Concerns is one of the most fundamental principles in software engineering [Par72; Dij76]. Thus, separation of concerns is crucial for leveraging the development of product lines, too. Since features are the primary concerns in SPLE, their implementation should ideally placed in one cohesive location to aid maintaining and evolving the system without huge effort. [Ape+13]

Uniformity was, to the best of our knowledge, firstly defined by Batory et al. [BSR04]. It refers to the fact that a product line is composed of different kinds of artifacts, i.e., source code in various languages, test frameworks, etc., which should all be targeted by the *same* feature implementation technique. This quality criterion addresses the problem of providing solutions specific to one language or implementation approach which does not scale due to an ever evolving tool and programming language landscape. [Ape+13]

Feature traceability refers to the possibility to detect features defined in the problem space also in the solution space. Accordingly, this ideally requires that one feature implementation is not *scattered* over different resources but located in one place which is not always possible. Additionally, variability implementation approaches may include traceability possibilities naturally, such as annotative approaches, whereas, e.g., the usage of run-time parameter may hinder easy traceability. [Ape+13]

Information Hiding Similar to the separation of concerns, *information hiding* is another key principle of software engineering. The main goal of this criterion is to write modular source code, consisting of internal and external parts. The external part of such module – such as an interface – provides a specification about its internals which can be used to reason about the internals. However, the implementation of the internals can change without affecting any of the other modules. [Ape+13]

Pre-planning describes part of the process before implementing the product line. Upfront, it must be analyzed which features are present, how they interact and how feature implementations can be reused. Due to the fact that not all features are likely to be known during the planning phase, this criterion targets the simplification of anticipating and integrating modifications at later times which is especially necessary in extractive and reactive engineering. [Ape+13]

Granularity [KAK08] refers to the level at which feature implementations are located. Each realization artifact can be considered a hierarchy of details defined by containment relationships. For instance, a coarse-grained variability mechanism allows to regard the topmost elements of the containment hierarchy (e.g., a Java class or even a file) as subjects to variation. Conversely, fine-grained feature implementation are possible, too, [Lie+10]. For instance, when employing preprocessors, almost every word in the source code (e.g., a parameter of a method) can be associated with one specific feature. Since it is hard to support fine-grained feature implementations with a coarse-grained technique, the feature realization and the variability mechanism should best reside at the same level of granularity. [Ape+13]

3.4.2 Compositional Variability

A compositional approach to implementing an SPL builds on *components*, each (ideally) realizing only one feature. A component is an *independently deployable* unit which can be *composed by third parties* and *without an observable state* [SGM02]. Therefore, it has to be clearly differentiated from its environment and hide its implementation, e.g., by offering interfaces only for its composition.

Composing In SPLE, components have to be explicitly mapped to the features they realize. In order to form a final product, a weaving mechanism selects the corresponding components according to the given feature configuration and assembles them. Typically, assembling components is not an automated but a manual process where the developer often needs to add source code (called *glue code*) to build an executable product. The compositional approach can be associated with positive variability because typically additional components will be integrated in the product if necessary. [Ape+13]

Pros and Cons Regarding the quality criteria, components support uniformity, information hiding and separation of concerns as well as feature traceability, in case each component realizes one feature. On their downside, a high amount of pre-planning is necessary to determine the right size of a component which only allows for coarse-grained variability. Moreover, having to provide manual glue code reveals a low degree of automation. [Ape+13]

Advanced Technical Realization Two advanced ways to apply FOP, which is a compositional approach to build product lines, predominated at most in the past: *collaboration-based* and *aspect-oriented* programming.

Collaboration-Based Programming On the one hand, collaboration-based FOP – as the name implies – relies on a set of classes forming a *collaboration* which ideally maps to one feature of the product line. A collaboration consists of multiple classes which play a certain *role* in the collaboration. Classes can play multiple roles in different collaborations.

Example 3.4.1: Collaboration-Based Programming

Fig. 3.4.1 highlights the key elements of collaboration-based programming by employing the example of the feature model for database contents.

*The main class representing the database contents provides base functionality to access the database contents in the basic version. In different collaborations, which each implements a single features, the class plays different rolls. In this example, the collaboration representing the feature **Song** shows the role of the class **DBContainer** which contains instances of the*

class Song and in the collaboration of the feature Book a collection of books, respectively.

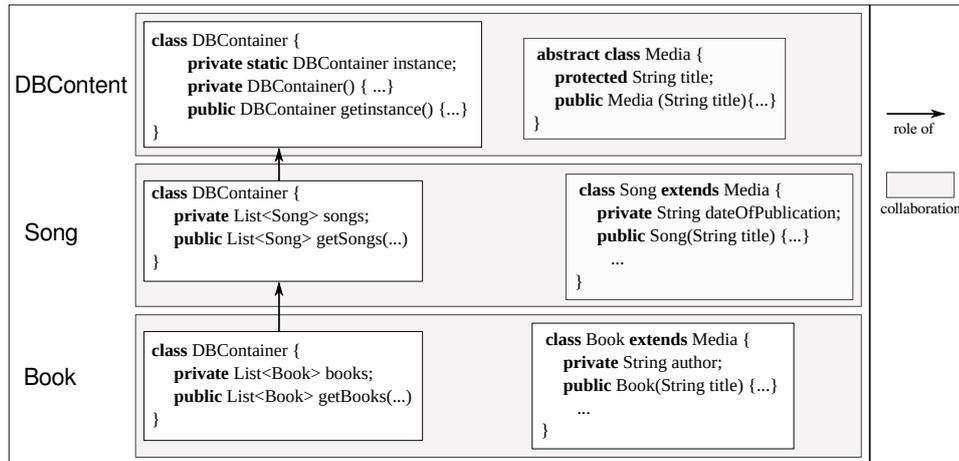


Figure 3.4.1: *Compositional* implementation of database contents using *collaborations*.

Specific languages and compilers or extension to existing GPLs, like **Jak** [BSR04] and **mixin** layers [SB02], support this realization approach. **AHEAD** [Bat04] offers a powerful and formal algebraic foundation for the approach. One of the biggest advantages is the straight-forward possibility to trace features in corresponding modules and the conceptual uniformity of applying the approach to diverse artifacts. Moreover, it requires little pre-planning and concerns are separated. However, the level of granularity hardly goes beyond method implementations and in general information hiding is no key concept. Additionally, although concerns are separated into modules, collaborations cut across different locations in the source code. [Ape+13]

Aspect-Oriented Programming On the other hand, applying *aspect-oriented programming* (AOP) [Kic+97] to realize features avoids the scattering of concerns. In AOP a base implementation is provided which can be extended and changed by aspects of which the base implementation is unaware. Similar to a variation point, a *join point* implies a point in the base implementation where an *advice* implemented in an *aspect* can change the basic implementation and, thus, adds variety. During the compilation, an *aspect weaver* integrates these changes based on the feature selection. **AspectJ**², an aspect-oriented extension of Java, supports the most powerful implementation of AOP.

Utilizing AOP for product line development typically implements one feature per aspect. In this way, AOP clearly separates concerns because the base implementation is unaware of the features and one advice is unaware of other feature implementations. Hence, the implementation of a feature is not scattered. This upside implies the downside that the base implementation is hardly accessible in the aspect implementation and that the aspect is likely to integrate errors (due to missing information about the base code) or loses its join point when the base implementation changes (called *fragile-pointcut problem* [KS04; SG05]). Further benefits are feature traceability (when each feature is implemented by one aspect), low pre-planning effort and a more fine-grained feature implementation level. While the concept is applicable uniformly, it requires specific implementations for each language and supporting tool. Moreover, hiding information (e.g., based on an interface for easy interchangeability) is no key concept of AOP. [Ape+13]

3.4.3 Transformational (Delta-Oriented) Variability

Similar to the compositional approach, a *transformational (delta-oriented)* variability mechanism relies on a *core* module and *delta modules* [Sch+10].

² <https://www.eclipse.org/aspectj/>

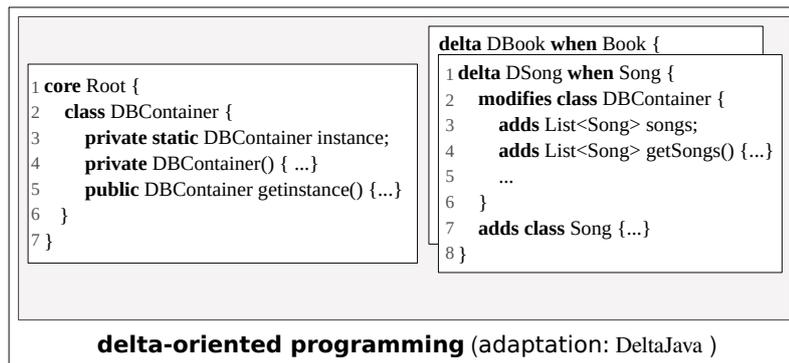


Figure 3.4.2: *Delta-oriented* implementation of database contents using a *core product* and *delta modules*.

Core and Delta Modules The core module, also referred to as *75 % model* when the product line is developed in a model-driven way, represents one valid product. As a consequence, the core module can be developed by utilizing normal single application engineering techniques. The delta modules store the change operations which are needed to realize further products and, thus, incorporate the variability implementation. Consequently, these delta operations in summary *transform* an already valid product into another customized product. In contrast to the compositional variability implementation, where components only *add* functionality, a delta module can *remove* elements from the core module yielding a potential decrease in size of the product and more effort to check the correctness of resulting products. The additional effort for ensuring consistent products due to the removal of elements can be alleviated, e.g., by including type-checking mechanisms [SBD11].

Example 3.4.2: Core Product and Delta Modules in Java

Fig. 3.4.2 presents excerpts of applying a delta-oriented approach to realize the database product line. The core product encompasses all elements to realize a basic executable product. In this example the core product is the empty database which offers as primary functionality access to its contents. Delta modules comprise the delta operations that have to be undertaken for implementing one feature. As demonstrated in the example, the delta module *DSong* which should realize the feature *Song* needs to adapt the class *DBContainer* to contain the songs. Furthermore, it needs to create the class *Song* and a corresponding implementation.

Model-Driven Transformational Variability While the initial work was performed for product lines written in Java, (ongoing) research predominantly focuses on *model-driven* product lines. For that reason, the following chapter on model-driven product line engineering provides further insights in the discipline (c.f., Sec. 4.1.2, II).

Summary Similar to compositional approaches, using delta modules facilitates feature traceability and a strict separation of concerns which can be installed at a fine-grained level and does not require much pre-planning overhead. The principles uniformly apply to different languages and tool but information hiding based on interfaces is no key concept of the general approach.

3.4.4 Annotative Variability

In contrast to the aforementioned variability implementation techniques, annotative approaches do not rely on a core implementation but implement the product line as a whole in the phase of domain engineering. As such, a superimposition [Bos99; CE00] of all products is developed and variability *annotations* serve as *presence conditions* for single code fragments by mentioning the

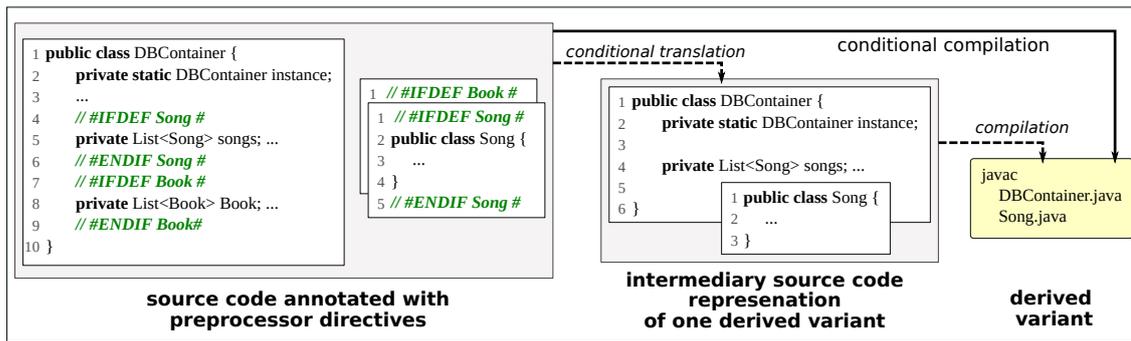


Figure 3.4.3: Annotative implementation of database contents.

feature (or an expression over features) enabling the code fragment. Accordingly, this approach realizes negative variability because the entire platform is built and artifacts corresponding with deselected features are removed when deriving a product.

Preprocessors Technically, the straight-forward implementation of annotative approaches utilizes preprocessor directives. In this case, code fragments which correspond to one specific feature (or more features) can be annotated by embracing them with a corresponding directive. When providing a feature configuration (in case of a preprocessor: a selection of the enabled preprocessor flags), *conditional compilation* is performed which only involves source code corresponding to the selected features to generate the bytecode of the product. While C/C++ includes a built-in preprocessor, other GPLs, like Java or Python, do not support preprocessors out-of-the-box. For that reason, preprocessor extensions of Java (e.g., Antenna³) exist to annotate the source code with preprocessor directives and offer conditional compilation of Java projects. Commercial tools, such as `pure::variants` [Beu13] or `gears`⁴, offer built-in solutions for conditional compilation based on preprocessor directives. [Ape+13]

Example 3.4.3: Annotative Source Code for Conditional Compilation

Fig. 3.4.3 depicts an example in which Java source code is surrounded by fictive preprocessor directives, e.g., a list of songs is only kept in the `DBContainer` when the feature `Song` is selected (c.f., lines 4-6). Based on enabled directives (representing the feature configuration), a corresponding preprocessing functionality removes all source code embraced by non-selected directives at compile-time. Thus, in this example, only the feature `Song` is selected and all source code which is annotated with different directives does not form part of the product derived by the conditional compilation. For demonstrating the “single-variant” source code the figure depicts also the intermediary state into which the complete source code is translated during the compilation before creating the final product.

Pros and Cons Since the annotations are exactly located at the corresponding realization artifact, this variability implementation approach includes native feature traceability and also supports a fine-granular feature implementation mechanism if using preprocessors. While an annotative approach requires little pre-planning and is a uniform concept, information hiding and separation of concerns can not be achieved when using preprocessors. Moreover, the level of cognitive complexity increases due to the fact that all product implementations are visible.

Virtual Separation of Concerns The complexity of maintaining preprocessor directives can be diminished, for instance, by generating *views* on the source code [Käs10]. Given a feature configuration (i.e., a selection of preprocessor directives) code belonging to other directives can be

³ <http://antenna.sourceforge.net/>

⁴ <https://biglever.com/solution/gears/>

virtually removed and is only shown as context information (“[]”). Alternatively, in case the still present preprocessor directives mitigate a clear overview, instead, background colors may be used to indicate feature annotations which then are removed artificially. The tool *Colored IDE* (CIDE) implements this functionality for representing feature annotations in Java source code [Käs07].

External Annotation Mapping As an alternative to the usage of preprocessors, annotations can be mapped onto artifacts externally. A mapping is a two-valued tuple, consisting of a presence condition and the corresponding artifact, that should be annotated. On the one hand, the presence condition, which we simply call *annotation* in the sequel and may also be referred to as *feature expression*, is a Boolean expression over the features defined in the variability model. On the other hand, the second element of the tuple, the artifact may vary with the granularity of the approach and, in case of source code-based product lines may be a text fragment. However, external mapping approaches are more prominent in model-driven product line engineering. For that reason, they are explained in greater detail in Sec. 4.1.2, III.

3.4.5 Multi-Paradigmatic Approaches

In practice, certain tools and research prototypes frequently offer a combination of the aforementioned approaches or the possibility to select one of them, implying that specific approaches may fit better for certain product lines. In this way, an implementation approach suiting the properties and the requirements of the system to be developed can be selected. For instance, a product line where most of the features are present in every product should rather apply an approach based on negative variability whereas positive variability pays off when features can be clearly distinguished and only a small number, e.g., of deltas, must be applied to assemble a product.

3.5 Product Well-Formedness

While the key motivation of SPLE is to automate the product generation, the product derivation step comes with a major requirement. Not only the feature configuration has to be valid (i.e., satisfying all constraints encoded in the variability model) but also the product corresponding with a valid feature configuration has to be (at least) syntactically correct.

Brute-Force Analysis One way to guarantee this correctness, referred to as *product well-formedness*, for the entire product line is to generate all valid feature configurations and to derive each corresponding product once in a brute force approach. However, this *product-based* analysis is only feasible for quite small feature models with a small total number of products. With an increasing number, particularly of optional features, the number of possible products grows exponentially yielding most of the times a set of products which is impossible to test in a reasonable amount of time. For instance, the variability model of the Linux kernel, KConfig, as a prominent example of a highly configurable system, was latest measured to consist of up to more than 6400 features with more than 3000 constraints, which still results in a number of configurations which is impossible to test in life time. Moreover, a product-based analysis neglects the redundancy present in products due to their reused parts. From a different point of view, a *feature-based* analysis considers each feature implementation in isolation and draws a conclusion on all products. However, since features are likely to expose different behavior when they are combined, a *family-based* analysis should consider also specific combinations of the products. [Thü+14]

Sampling To this end, different *sampling* strategies mitigate the effort to guarantee product well-formedness based on examining every product by inspecting a representative *sample* set of all products instead. The field of sampling products is largely populated and still actively researched, e.g., as community challenges [Pet+19; Fer+20]. Varshosaz et al. [Var+18] classify sampling strategies based on their input data and the sampling technique. On the one hand, the *input* data can be part of the *problem space*, which was in all investigated strategies of this survey the feature model and sometimes further domain knowledge. On the other hand, the input data can be part of

the *solution space*, i.e., test or implementation artifacts. The techniques are either *manual*, *semi-automated*, completely *automated* or based on *coverage* criteria. While in manual sampling, for instance, a domain expert selects the set of sampled products, automated sampling either employs *meta-heuristics*, i.e., a local or a population-based search (for an optimum), or a *greedy* strategy. Semi-automated strategies allow to integrate and vary different parameters, like the sampling time or a coverage degree, and typically receives a starting set from a domain expert. Finally, *coverage sampling* typically includes at least feature-wise or *t*-wise (where *t* represents the number of the different combined features) feature interactions but may also consider implementation artifacts in a white-box fashion.

3.6 Summary

This chapter presented basic terms and strategies how to develop product lines in a systematic way by adhering to the principles of variability and organized reuse. These techniques lay the grounds for incorporating the ideas of model-driven software engineering in product line engineering methods. The next chapter illuminates how the concepts explained in this chapter convey when models are the primary development artifacts.

Chapter 4 Model-Driven Software Product Line Engineering

*Ante mare et terras et quod tegit omnia caelum
unus erat toto naturae vultus
in orbe quem dixere chaos¹*

Ovid, Met. Book I, Lines 5-7

~

Both, model-driven software engineering and software product line engineering, aim at increasing the level of productivity by developing complex software systems in an organized and automated way. Combining both disciplines bespeaks an increase of their benefits. By *abstracting* the product line with models during domain engineering, not only the product derivation in application engineering is automated but model-driven software engineering allows to *automate* the generation of realization models (and in the end, source code) in domain engineering, too. Consequently, in *model-driven (software) product line engineering (MDPLE)* [Gom05; Cza+05], models are the nucleus to represent the product line artifacts at every development stage.

As a consequence, the two key elements of MDSE, *models* conforming to metamodels and *model transformations*, have to be lifted to properly address the additional dimension of variability. On the one hand, models representing a single system have to become multi-variant, meaning they have to capture a family of systems. On the other hand, to keep multiple multi-variant models consistent, automated techniques are necessary to transfer variability information from model into another. Besides creating a target model, model transformations can be lifted to address the dimension of variability.

This chapter presents the possibilities to lift models in Sec. 4.1 and their maintenance, as implemented in existing MDPLE solutions, in Sec. 4.2. Finally, Sec. 4.3 illuminates the employment of model transformation technology in model-driven product lines, which keep the variability information consistent in software families, whereas Sec. 4.4 summarizes the insights.

The chapter shares material with [GW21] and [WG20a] which lay the foundations for its contents.

¹ Before sea and earth and the sky which covers everything existed, the appearance of the nature was one in all the globe which they have named Chaos.

4.1 Multi-Variant Models

The first step of fusing MDSE and SPLE into MDPLE has to enrich *models* by the dimension of variability in space. This section delimits the scope of the considered multi-variant models, firstly, and presents how the variability mechanisms integrate variability information into models, secondly.

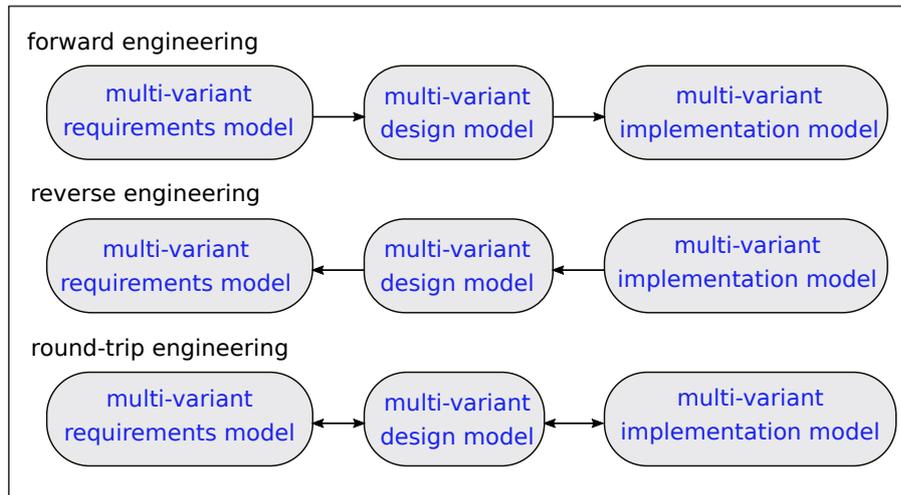


Figure 4.1.1: Engineering directions in MDPLE.

4.1.1 Preliminaries

To be used in MDPLE, in the first place, models have to represent variable content. In contrast to developing a *single* system, as in classical MDSE, models need to represent a *family of* systems (i.e., a product line). Accordingly, the general engineering directions depicted in Fig. 2.1.5 refine to those depicted in Fig. 4.1.1: instead of (single-system) models, models representing a family of systems are developed and engineered in forward, backward or round-trip direction in MDPLE. We refer to models representing more than one product as *multi-variant models* as opposed to *single-variant models* which represent only one specific product. Fig. 4.1.2 depicts a multi-variant UML class model encompassing the various database contents which correspond to the feature model presented in Fig. 3.3.1. Since such model comprises all variants in a superimposition, it is also commonly referred to as *superimposed* or *150 % model*. Derived products may only contain parts of such model.

Metamodel Restrictions In general, although almost arbitrary variable content should be integrated in the models, commonly the metamodel, to which the model conforms, restricts the properties of the model elements. Only few tools (e.g., the internal representation of SuperMod) allow for *unconstrained variability* or for merging metamodels into one. In this thesis multi-variant models are instances of a *single metamodel* with *single-variant semantics*:

Single Metamodel A multi-variant model is always an instance of only *one metamodel* and not of multiple different ones thereby conforming to Prop. 4.1.1:

Property 4.1.1: Single Metamodel

The multi-variant model is instance of one single kind of metamodel.

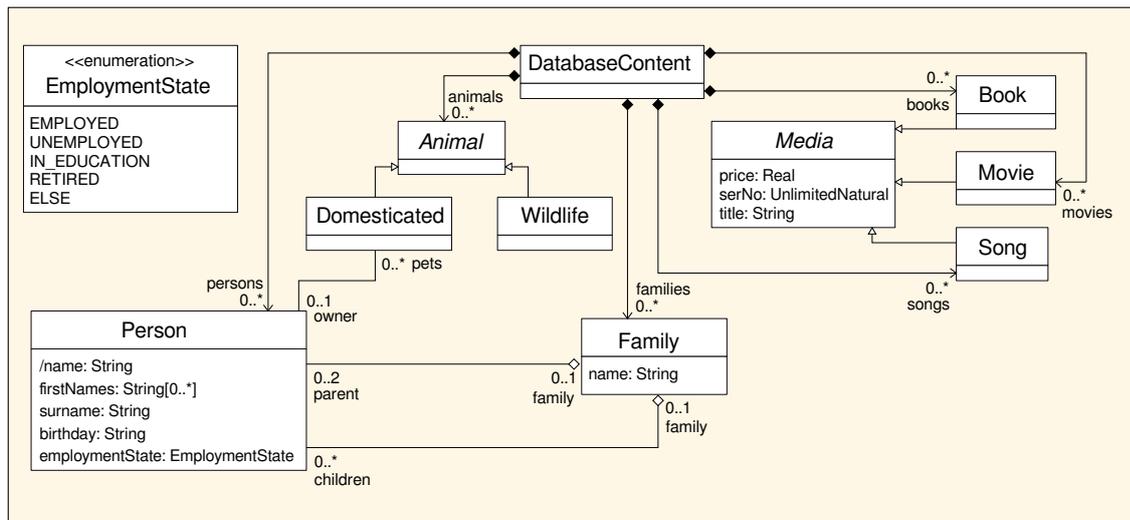
Figure 4.1.2: UML class model representing the database contents in a *multi-variant model*.**Example 4.1.1: Multi-Variant Model**

Fig. 4.1.2 depicts an example of an superimposed UML class model. The model superimposes the varying contents of the database. Accordingly, it comprises not only persons and animals but also types of media and organizes persons into families.

Thus, different configurations of this model can remove parts of the model which should not be present in a specific variant of the model. For instance, a family-database may require persons and may integrate domesticated animals but does not need media necessarily. Similarly, a media-database which could be used in a warehouse does not associate a family or animals with the media. However, all of these possibilities are integrated in the multi-variant model. Nonetheless, the model does not exceed the boundaries of the single-variant semantics of the metamodel.

In contrast, techniques exist which identify commonalities between *different* metamodels, and summarize these metamodels in one *single underlying metamodel* (**SUMM**) [ASB09], which is based on multi-view modeling [Fin+92]. An instance of a SUMM is *not* considered a multi-variant model in our context. Nevertheless, instances of different metamodels are necessary to holistically reflect a system, as also pointed out by Gomaa and Shin [GS02] who involve a different model for each development stage.

Single-Variant Semantics Furthermore, the metamodel of multi-variant models in this thesis conform to single-variant semantics as defined in Prop. 4.1.1. Thus, the models express *constrained variability* only.

Property 4.1.2: Single-Variant Semantics

The multi-variant model conforms to a metamodel with single-variant semantics.

As an example, in an Ecore or UML class model the name of a class cannot vary. In the database example it will not be possible to call the root class of the model (**DatabaseContent**) differently (e.g., **FamilyDatabase**) if a database consisting of families only is derived. If necessary, a multi-variant model conforming to single-variant semantics can be obtained by *normalizing* a “malformed” multi-variant model (i.e., a model violating the single-variant semantics of its metamodel) [RLK19].

Table 4.1: Comparison of terminology used in MDSE and SPLE.

term	MDSE	SPLE
<i>platform</i>	any technical or business details specific to a system (c.f., MDA in Sec. 2.1.3)	“a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced” [ML97]
<i>trace</i>	container of links between source and target model of a model transformation	summary of links between features and realization artifacts
<i>mapping</i>	link of corresponding source to target elements	link of a feature expression to realization artifacts
<i>derive</i>	execute a transformation to create a target representation from a given source model	create a customized product based on the platform artifacts and a given feature configuration

Terminology As another point, it must be noted that certain terms appear in both disciplines, MDSE and SPLE, but with diverging meanings. Table 4.1 collects an overview of ambiguous terms relevant in this thesis. In the following explanations of this thesis we always use the term *platform* in the meaning of the SPLE context because we do not consider platform-dependent models. Since the thesis describes the usage of traces recorded while executing a model transformation, a *trace* always refers to a collection of links between the source and target elements of a model transformation. In contrast, we take a *mapping* as a synonym for assigning an annotation to a model element. *Mapping models* capture the *variability traceability* [BBM05] of all elements in a domain model to annotations. Finally, to indicate the meaning of *derive* we clearly state the artifact which is derived, a target representation from a source model by applying a transformation or a *product* from domain engineering artifacts (as in SPLE). Due to the focus of the thesis on negative variability, the following descriptions additionally utilize the term *filter* as a synonym for deriving a product because approaches realizing negative variability always remove deselected elements from the superimposed domain artifacts for deriving a product.

4.1.2 Variability Mechanisms

As Chp. 3 explains, variability can be realized either by an annotative, a transformational or a compositional mechanism. The same principles apply to MDPLE including further specifics which relate to the nature of MDSE. Consequently, models can be lifted to realize the dimension of variability in space by exploiting the same three variability mechanisms. Most of the techniques and concepts in MDPLE are reflected in the heterogeneity of supportive tools. Therefore, the following descriptions regard the tool landscape while explaining the compositional, transformational and annotative variability mechanism applied to models subsequently in the following paragraphs. As a special form, the closing of the section sheds light on view-based editing approaches which intend to hide the complexity of developing all variant.

I Compositional MDPLE

Sec. 3.4.2 states that compositional variability mechanisms rely on the predominant concept of a *component* and a *composer* which builds products based on positive variability. Besides relying, for instance, on interfaces to encapsulate the behavior of a component, collaboration-based FOP and AOP are specific techniques for this mechanism. To mimic multi-variant contents, almost the same principles transfer from programming to modeling. Applying aspect-oriented modeling as well as composing superimposed models are two means to realize a compositional approach:

Aspect-Oriented Modeling Similar to aspect-oriented programming, where a *base program* is extended, *aspect-oriented modeling* [Wim+11b] extends a *base model* with *aspect models* which refer to join points in the base model. For creating a product line, each modeled aspect should be

related (at least) to one feature which it realizes. The *aspect weaver* will integrate the advice in a product if a feature configuration selects the respective feature.

Groher and Völter [GV09] realize *aspect-oriented MDPLE* by exploiting the concept of aspect-orientation. The authors contribute a holistic development strategy and corresponding tool to model a product line with Ecore models, to formally map models of the problem space onto corresponding ones of the solution space and to generate source code based on M2T transformations eventually. The tool environment resides in the *open Architecture Ware (oAW)* framework which offers different languages and tool support for specifying grammars of DSLs or performing model transformations. As a prominent example, **Xtend**, a powerful Java dialect, resides in the oAW.

For executing model transformations in an aspect-oriented way, the solution exploits the (former) support of the Xtend programming language for aspect-oriented programming, allowing to declare advices for methods. Furthermore, **Xpand**, another language of the oAW, supports the declaration of aspects in M2T transformation. The solution of Groher and Völter expects a variability model exported from the commercial tool pure::variants and turns it into a globally accessible variability model in its own tool environment.

Despite the fact that aspect-oriented solutions typically realize positive variability, Groher and Völter discuss ways to implement negative variability, too. For realizing this concept at the level of modeling, they offer the tool XWeave [GV07] as an integration into the oAW. They realize negative variability in the model transformation by implementing an overall generator and query whether a feature is selected before generating the corresponding element. [GV09]

Conversely, the tool **MATA** (*Modeling Aspects Using a Transformation Approach*) [Whi+09] utilizes Graph grammars to realize an aspect-oriented approach to SPLE. Every element of a UML class, state or sequence model can serve as joint point and can form part of the base model slice. A graph transformation written in MATA's own language reuses the existing Graph transformation engine of AGG [Tae99; Tae03; RET11] to compose the base model slice with aspect model slices. While the base model may contain different model types, an aspect model slice can only target a model of one kind. For example, an aspect model cannot modify state and a sequence model simultaneously. Since Graph transformations are formally founded and well-understood, MATA naturally supports formal analysis methods to detect interacting aspects by a critical pair analysis.

Superimposition-Based Approach Another solution to enrich models with variability for employing a compositional mechanisms builds a superimposition of model variants by identifying matching model fragments in the variants which can be composed [Ape+09]. In this context, an identified *model fragment* needs to conform to its metamodel and represents one part of a superimposed model corresponding with one feature. The superimposition requires a tree-structured subject language and matching names of elements, that should be unified. The tool **FEATURE-HOUSE** [AKL09; AKL13] implements this approach and offers automated generators for language integration and automated composition tools for deriving products.

II Transformational MDPLE

A second possibility to realize positive variability in MDPLE is to employ transformations. On the one hand, delta-oriented tools build on a core model which is modified by delta modules which bundle delta operations. On the other hand, different proprietary variability models and transformation approaches have been proposed at which this section looks afterwards.

Delta-Oriented Modeling Approaches Delta modeling [CHS15], as one means of transformational MDPLE, lays the grounds to transfer the delta-oriented variability mechanism from source code to models. A *delta module* assembles *change operations* in an appropriate order and applies them to the *core model*. In contrast to a general-purpose model transformation, the operations in a delta are restricted to address only elements which are allowed to vary, for instance, a delta operation cannot modify the IDs of elements. Besides the delta operations, a delta module encompasses an *application condition*, typically a Boolean expression, such as a formula in propositional logic, over the features of the corresponding variability model. One cause for errors resides

in applying different delta modules sequentially because they may compose conflicting operations. Automated tool support should address this problem by detecting and resolving these conflicts.

DeltaEcore [SSA14b; Sei17] represents an automated tool to realize the delta-oriented variability mechanism. After defining a base delta language in form of a metamodel for the target language, delta dialects can be derived based on this metamodel. The dialect defines language-specific delta operations which are used to specify the delta modules. Specific to this approach is the introduction of a *hyper feature model* [SSA14a]. Products are configured based on the hyper feature model, which allows to specify revisions of individual features. **DarwinSPL** [NES17] adapts and extends the concepts of DeltaEcore by supporting implementations that respect the *contextual variability* of a product line.

In contrast to automatically deriving delta languages, a higher degree of automation is achieved when the delta operations in form of an edit script [KKT13] are derived. The tool **SiPL** [Pie+15] generates edit scripts, i.e., a delta module, by comparing model instances. In addition, Pietsch et al. [Pie+19] offer a formal analysis and resolution of conflicting delta operations among the modules.

Class Layers Hendrickson et al. [HJH06] suggest to organize class models in different layers, each modifying a base layer. As is the case in delta-oriented approaches, a layer incorporates an “alternative” design decision by defining delta operations (add, remove) which relate to the base layer and can be composed by respecting further relationships that can be defined between the layers. The tool **EASEL** implements this concept which also involves merging and comparing models.

Transformational Variability Languages The *Common Variability Language (CVL)*, introduced in Sec. 3.3.2), and the family of *Variability Mapping Languages (VML*)* [Zsc+09], both foster a delta-oriented variability mechanism where transformations extend a base product.

CVL [Hau+08; Fon+15] realizes positive variability by relying on a *base model* incorporating *placeholders* which can be implemented differently in *variation models*. Variation models are expressed in an entirely generic language. *Resolution models* specify the selected products which can be transformed into the *variation realization* resembling one product in an iterative way. For that reason, the approach is also called *base variability resolution (BVR)*. Due to its generic nature, CVL is supposed to be integrable with arbitrary DSLs (and GPLs for modeling, such as UML).

VML* offers the creation of customized languages to specify mappings from features to model elements by using the language elements of the corresponding domain model. Moreover, the VML* metamodel incorporates elements, such as *pointcuts* or *actions* which are performed on the target model during the derivation of products. Based on the instances of the VML* metamodel, a general purpose model transformation in Xtend is generated. This language realizes the derivation of single application models. Consequently, this approach which supports positive as well as negative variability, can be categorized as transformational approach.

III Annotative MDPLE

In annotative MDPLE, *superimposed* models, such as the one depicted in Fig. 4.1.2 which represents the diverse contents of a database, express the entire variety of products in a commonly called 150% model. In contrast to the work by Apel et al. [Ape+09], in which the annotated superimposed model is composed by assembling model fragments satisfying a feature configuration, elements of an annotated model based on negative variability are removed to create a product. To derive products, annotations have to be mapped onto the elements of Fig. 4.1.2 to declare in which variants they are present. Accordingly, Fig. 4.1.2 needs to be refined as sketched in Fig. 4.1.3. In this refined version, the annotations represented in the rounded rectangles declare in which derived variants the model element onto which it is mapped is present. For instance, the two associations which connect the classes **Person** and **Family** are only present in derived variant if the features **Family** and **Relation** are selected both.

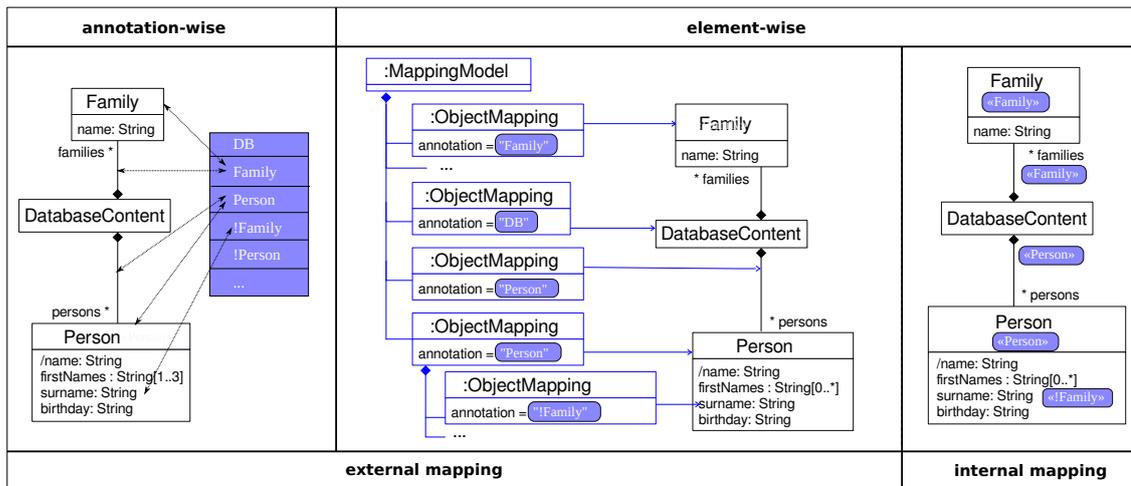


Figure 4.1.4: Mapping notations.

The *product line UML software engineering (PLUS)* method serves as one example where static stereotypes extend the basic language constructs of UML. They allow to designate, for instance, *default* elements («kernel») or *variants* («variant») [Gom05]. Thus, model elements carry annotations in form of stereotypes. Ziadi et al. [ZHJ03] introduced the concept of exploiting UML stereotypes firstly to the best of our knowledge.

In a similar way, a template mechanism assembles UML models [CA05; CP06]. The template comprises the superimposition of all variants which are annotated by utilizing UML stereotypes, too. In contrast to PLUS, the approach does not employ a static default set of stereotypes, such as «kernel» or «variant», but the stereotypes declare an annotation which is a propositional formula over features defined in a feature model. Ex. 4.1.2 demonstrates the usage of customized stereotypes for annotating a UML class model.

Example 4.1.2: Internal Mapping with Feature-Based Stereotypes

The right side of Fig. 4.1.4 sketches the internal mapping based on customized stereotypes in UML, as suggested in the template mechanism [CA05]. The example derives the stereotypes from the database example and conveys the same mapping information as is present in the middle and left side of the figure.

Please note: While in the original contribution annotations are only mapped onto classifiers which is due to the state-of-the-art capabilities of the UML specification, the example annotates properties, too. Accordingly, the annotation of the property *surname* can only be annotated if the capabilities of UML are extended.

Pros and Cons of Internal Mappings Despite the benefits of reusing existing functionality which reduces development and training cost, three general downsides of reusing existing model language capabilities for (internal) mappings stand out:

First, the *annotations are tightly intertwined with model elements*, and are, thus, scattered over the model. As a consequence, for deriving a product corresponding elements have to be searched in all realization artifacts.

Second, the *granularity of the mappings is limited* by the capabilities of the modeling language. Therefore, annotations may not be mapped onto all model elements. For instance, while UML is quite powerful, it is not possible to annotate elements of an Ecore model with customized stereotypes but as only means with the type `EAnnotation`. Overloading an existing metamodel type in this way fosters a disciplined usage based on conventions.

Thirdly, reused-based internal mappings are restricted to one type of model and, thus, not generally applicable. As an example, UML stereotypes cannot be employed to annotate an Ecore model.

Proprietary SPL Development Languages In order to address the first downside, mappings could be stored externally in separated files. To address the second and third downside of insufficient granularity and genericity, several *proprietary* languages express variability and the model elements in an intertwined way, thus, being able to adapt their language appropriately.

Clafers serve as example of a proprietary language with internal mappings. A *clafers* is a unit summarizing variability annotations (adhering to a proprietary constraint language), structural and behavioral model elements. These units are expressed in the similarly called modeling language [BCW11; Bak+16; Juo+19]. The **VML*** framework serves as another example declaring “annotations” in a proprietary language. The developer can specify a mapping language which allows to use language elements of the domain model. Accordingly, mappings of features to domain model elements are specified in this dedicated language from which a general purpose model transformation in Xtend is generated. This language realizes the derivation of single application models. Consequently, this approach which supports positive as well as negative variability, can also be categorized as transformational approach but also contributes concepts to annotative mechanisms.

External Mappings As stated in Sec. 3.4.4, mappings can as well be assigned in a file or data structure separated from the multi-variant model. Such *external* mapping offers the advantage that annotations are not scattered over the modeled artifacts but reside in a dedicated location. In this way, external mappings separate the concern of modeling from the annotation task. Furthermore, a variant can be derived more easily because the elements which should be included can be determined by looking them up at one dedicated location, the external mapping representation. Both, **FeatureMapper** [HKW08] and **Famile** [BS12], are extensions to Eclipse and support modeling product lines with eMOF-based models and external mappings. For mapping features to the domain model they include a separate mapping model which relates domain model elements with a feature. In contrast to FeatureMapper, which supports mapping feature expressions, i.e., propositional formulas over features, onto *objects* of the model only, Famile supports a finer level of annotation granularity where *structural features* can be annotated as well. Moreover, due to the capabilities of the proprietary mapping model, alternative modeling decisions which cannot be expressed by the single-variant syntax of the domain model, are possible. For instance, the name of a UML or Ecore class can vary by including an alternative mapping element for the name carrying another annotation.

Moreover, approaches which employ a correspondence model between the variability and the realization models, for instance, multi-view product lines [GS02]), fall into the external category. In the multi-view approach models at each development stage, for example activity and class models are each considered a view. The variability model is considered another view and a correspondence model not only links model elements of each view but also maps them onto features of the variability model.

Example 4.1.3: External Mappings

The left and the middle part of Fig. 4.1.4 exemplify how to realize external mappings. The left of figure demonstrates the aforementioned form of an annotation-wise mapping, which requires an external mapping artifact. The figure sketches the external annotation-wise mapping as a separate file which collects features. In multi-view approaches, this kind of mapping may be a separate correspondence view which stores links from model elements onto annotation.

*Instead, the external mapping model, depicted in the middle of Fig. 4.1.4, comprises the same classifiers and relations as used in the domain model for which it provides annotations. The figure sketches the mapping model, noted in abstract syntax, in the way the tool Famile [BS12] realizes it. The mapping model reconstructs the structure of the domain model for which it stores annotations exactly. **ObjectMappings** represent each modeled object, i.e., not only the enumeration, the classes and associations but also the properties, operations and their parameters (the latter are not depicted in the figure). The object mapping stores the annotation which is a propositional formula over features of the feature model and*

references the corresponding object of the domain model.

Notice: For simplicity and easy readability, the mapping model excerpt in Ex. 4.1.3 does not include all object mappings which form part of the mapping model in the excerpt. Moreover, the granularity of annotations which can be expressed with Famile goes beyond the granularity of many other tools because it allows to annotate the structural features, for instance the type or name of an object, and also to provide alternatives from which we abstract in this figure, too. In addition, whenever different models build the product line, the same number of independent mapping models may exist in this approach (because they reflect exactly the structure of the corresponding model).

Annotation Granularity Regarding the granularity, this property corresponds with the quality criterion as declared in Sec. 3.4.1 for product line development in general. The property assumes that all realization artifacts build a strict containment hierarchy where the overall container resides at the coarsest level of granularity. In the context of models, this could be a package or even the file containing the model. Conversely, for instance in eMOF, at the finest level of granularity reside the structural features of model elements, for instance, the name of a class or the type of an attribute. The granularity influences the derivation complexity (higher if fine-grained) and the redundancy of artifacts and possibilities for reuse (less redundancy if fine-grained).

Model Filter The term *model filter* refers to the facility which derives the product based on a feature configuration in an approach realizing negative variability. [GW19c]

Flat vs. Hierarchical Filter Firstly, the model filter may vary with respect to its capability to ensure that a well-formed product is derived and the way it incorporates structural information of the annotated model. We discern two kinds of model filters: A *flat* model filter always removes the elements the annotation of which does not satisfy the feature configuration whereas a *hierarchical* filter takes dependencies inside the model into account and may propagate selection states accordingly. As an example, if the domain model forms a spanning containment tree and a parent element is not included in a configuration but at least one of its children, a hierarchical filter will either exclude all children from the derived product or enforce the integration of the parent. Hierarchical filters allow for simpler annotations of model elements because the Boolean expression does not have to respect existence relationship while ensuring well-formed products, nonetheless.

Secondly, filters can vary by the way how they treat model elements missing an annotation. Besides others, one strategy includes elements missing an annotation in all configurations regardless of the filter a priori².

Finally, in a preprocessing step before the derivation, the consistency of the derived product can be ensured by propagating selection states, thus, avoiding the well-formedness problem. In hierarchical filters not only the obvious annotations can be determined but also can they be overridden based on knowledge about the syntax of the metamodel. For instance, in UML class models a super class should not be removed if the class inheriting from the super class remains in the product. In this example, either both classes need to be removed or both need to remain. Czarnecki et al. [CA05; CP06] offer rules to assign a more specific annotation (than `true`) in UML class and activity models. Similarly, the tool MODPL [BW14] offers a set of static rules which generate annotations by regarding dependencies inside Fujaba models [NNZ00]. More generically, the problem of generating malformed products due to the model filter can also be solved by writing repair operations, for instance, in a generic DSL [BS16].

Technically, several MDPLE solutions implement the product derivation (i.e., the model filter) as a (general purpose) model transformation [ZJ06; BOT07]. This transformation needs to understand the notation of annotations as well as the feature configuration and has to remove elements from the domain model accordingly. To avoid the necessity to learn a “general-purpose” model

² This functionality corresponds with a preprocessor where source code without an enclosing directive is omnipresent.

transformation language, simplified languages, e.g., designed with the VML*, may generate a general-purpose model transformation language from a less complex language which is specific to the applied approach.

IV View-Based Editing of Product Lines

While internally a superimposed model is maintained, several approaches hide the complexity of the complete product line from the developer. Particularly the mapping of annotations may be hidden by providing editors for manipulating a *single variant* only. We refer to these approaches as *view-based editing* because only parts of the entire product line are exposed to and edited by the product line engineer. Since view-based editing may support multiple variability mechanisms, we categorize it as a standalone group of variability mechanisms, although primarily only the way of *editing* the product line differentiates from the aforementioned mechanisms. The essential idea of view-based editing in MDPLE is similar to virtually separating concerns in annotated code by conditional compilation [Käs12] or providing views on a superimposed source code according to the choice calculus [EW11; WO14].

A special form of view-based editing is *single-variant* editing [Sch18] in which the product line developer edits one complete single variant of the product line only. Conversely, a “view” – as proposed in the multi-view approach by Gomaa and Shin [GS02] – is a synonym for exactly one kind of model. As another form, in multi-view modeling a “view” may encompass all model slices, such as a diverse UML models and a Java model, which satisfy a feature configuration [Ana+18].

Single-Variant Editing – SuperMod represents one tool which realizes *single-variant editing* [SW16; SW19]. It is grounded in the *Uniform Version Model (UVM)* [WMC01] which unifies temporal *revisions* and spatial *variants* by considering them *versions*. The model relies on configuration management concepts which populate the local workspace with a single variant by checking it out from a local repository. A checkout needs to select a revision number and a complete configuration of the feature model (called *choice*) which is present at this revision. After editing the workspace contents, a commit specifying an *ambition* (i.e., a partial configuration consisting potentially even of a single selected feature only) integrates the modifications as new revision in the internal repository which contains the superimposition of all changes applied to the feature model and the domain model. After a commit, the annotation of each element in the internal superimposed model is replaced by the previous annotation combined with the new revision number and the feature ambition (present in commit) or its negated form (absent in commit).

It must be noted, that the superimposed model may consist of instances of different metamodels without maintaining correspondences between them. Finally, during the checkout and commit, consistency checks are performed. Depending on the kind, conflicts and inconsistencies are either repaired or reported to the developer.

View-Based Editing of Multiple Models – VaVe As mentioned in the introduction to this section, approaches combining different metamodels in a *single underlying metamodel (SUMM)* exist. Corresponding tools, such as **VITRUVIUS** [KBL13], generate views on the single underlying model, an instance of the SUMM, and edit only the parts present in this view. In contrast to SuperMod, these approaches maintain consistency operations between the elements of different models to synchronize changes made to one model with corresponding elements of other models kept in the SUM.

Ananieva et al. [Ana+18] extend the single-variant development environment of VITRUVIUS to support the creation of a product line and its subsequent evolution by adding a metamodel for maintaining variability in space and time. This extension realizes a delta-oriented product derivation management and also relies on checking out a variant from the underlying workspace, and committing it again. All delta operations in the view are monitored and applied to all affected elements on commit. As effect of the commit, concurrent views are maintained as well. In addition, features are explicitly versioned, resulting in directed deltas which have to be chronologically consistent. For expressing relationships between different metamodels, SUMM approaches

commonly use proprietary delta operations which do not conform to a “general purpose” model transformation language.

Projectional Editing As another alternative to provide views but also to support switching the variability implementation mechanisms, tools may realize a *projectional editor*. Typically, a projection includes parts of the abstract syntax which is modified directly, instead of, e.g., editing the concrete syntax which is parsed into abstract syntax afterwards. While Walkingshaw and Ostermann [WO14] promote the usage of projections to allow for single-variant editing, these projections comprise concrete syntax which still has to be parsed into abstract syntax. Conversely, the following paragraphs sketch approaches that use projections of the abstract syntax.

Leviathan [Hof+10] enables the generation of views on file systems. Consequently, a checkout populates the local workspace by mounting parts of the complete file system and a local edit modifies the complete file system. This concept of *edit in isolation* proves to increase productivity despite the loss of overview of changes which have to be consistent with the hidden artifacts. This insight results from a case study with the version editor [Atk98; ABB02].

PEoPL [BPB17] is an integration to the projectional language workbench (MPS) [Voe11]. Foundational ideas of integrating variability implementations in MPS were presented by Völter et al. [Voe10]. PEoPL allows to interchange the annotative with a modular variability mechanism by the usage of projections of the abstract syntax which is modified directly.

Since PEoPL only exists in the closed environment of MPS and focuses on code-oriented product lines, as an alternative for product lines modeled in eMOF a projectional editor is proposed to address this shortcoming [Reu+20]. This approach allows to switch the projection of an annotative 150 % model to a delta-oriented representation and vice versa by using a virtual abstract syntax graph (VASG) as pivot model in between of both. The approach normalizes the superimposed representation in case it would violate single-variant model semantics and matches delta operations in modules to combine them in the virtual representation. The models in the editor are transformed into the VASG and vice versa for switching the projections. However, the product line developer still edits the concrete not the virtual abstract syntax tree.

4.2 Annotation Maintenance in Existing MDPLE Solutions

For enabling MDPLE, solutions need to support the maintenance and realization of variability information across models not only for one model as described so far. When instances of different kinds of metamodels together form the product line, the necessity to maintain the annotations of corresponding elements in these models arises. As an example, in forward engineering (c.f., Fig. 4.1.1) it will be redundant and error-prone effort to map annotations *manually* onto models at a later stage of development if another already annotated model with similar information exists at an earlier development stage. Instead, existing annotations should be propagated to corresponding elements automatically and synchronized whenever the product line evolves. Although this behavior is predominantly relevant in annotative variability mechanisms, this section illuminates how annotations are maintained across models in each variability mechanism, particularly focusing on their degree of automation.

Accordingly, the first part of this section (Sec. 4.2.1) presents a feature-based classification of annotation maintenance which we map onto an exemplary set of MDPLE solutions in Sec. 4.2.2. Thereafter, Sec. 4.2.3 analyze the results.

4.2.1 Classifying Properties of Annotation Maintenance

At first, this section inaugurates distinguishing and common criteria of maintaining annotations in one model and across models followed by examining how the MDPLE tools introduced in Sec. 4.1 match these criteria.

The feature model in Fig. 4.2.2 comprises the characteristics to classify (existing) approaches with respect to their maintenance of annotations in one model and across models, for instance by involving model transformations. The following three main paragraphs explain details of the

main categories we identified, the way annotations are mapped onto product line fragments (i.e., the *mapping*), the *propagation* of annotations, and the *consistency* maintenance, in sequence.

Mapping As discussed when explaining the annotative variability mechanism (c.f., Sec. 4.2.1, III), the *notation* and *placement* of mappings varies. Firstly, the two possibilities of annotation-wise and element-wise mappings exists and secondly, annotations are either integrated in the domain artifacts (internally) or stored in a separate file (externally), as illustrated in Fig. 4.1.4.

The mapping notation is relevant for the necessity to apply further actions when multiple models build the product line. If an annotation-wise mapping is applied, no propagation of annotations has to be performed because elements are removed or added to their corresponding annotation instead independent of the metamodel. As a consequence, a *propagation* of annotations occurs for element-wise mappings only.

Furthermore, internal mappings may diverge with respect to their representation. They can be integrated by reusing existing language constructs, such as the UML profiles, of the domain artifact or by extending the existing language, such as by defining a new language construct. In contrast, an internal mapping can be realized based on a proprietary language which combines the annotation with the realization artifact, such as in a clafer.

For applying these categories to compositional and transformational variability mechanisms as well, a third kind of mapping notation can be introduced. In these two approaches annotations are mapped onto modules. As a module collects a set of realization fragments or edit operation in compositional and transformation approaches, respectively, we consider them a special form of element-wise mappings.

Annotation Propagation Secondly, the *propagation* of annotations is either *manually* performed or executed in an *automated* way. While a manual "propagation" implies that the product line developer has to assign the annotation to each model element, an automated propagation is performed by the tool. If the tool cannot annotate all target elements automatically but requires to ask the developer in case of uncertainties, the propagation is classified as *semi-automated*.

The scope clarifies whether the propagation of annotations occurs in one model (*intra*), for instance to assign annotations to model elements still missing an annotation, or across models (*inter*). The latter property can assure that multiple models forming one product line receive annotations consistent with the annotation of a corresponding element in another model.

Annotation Consistency In SPLE, different consistency analyses have to be performed. At first, feature configurations must be consistent with the feature model (i.e., no constraints of the feature model should be violated). Furthermore, the derived products need to be well-formed (c.f. Sec. 3.5). For the maintenance of corresponding annotations in different models, which are created with model transformations, the *commutativity* criterion [Sal+14] is predominant.

Commutativity The commutativity criterion informally presented in the introduction in Fig. 1.2.1 is defined based on a single-variant model transformation (**SVMT**) and a multi-variant model transformation (**MVMT**). Both paths, MVMT-filter and filter-SVMT, need to commute: Deriving a product from the multi-variant source model and transforming it with the SVMT creates a single-variant target product. This product must be equivalent to the product that is derived from a multi-variant target model, created by executing the MVMT, when applying the same feature configuration. If this property holds for each valid feature configuration, the MVMT is consider to be correct and, thus, the annotated target model is consistent with the annotated source model. Note: The criterion can be generalized to be independent of transformations by defining a equivalence operator³ between two instances of different metamodels: After having defined the equivalence of two instances of different metamodels, the same criterion can be applied. If applying the same feature configuration to multi-variant instances of different metamodels and the filtered

³ Equivalence of two models conforming to two different metamodels is discussed in Sec. 7.4.2 where Fig. 7.4.3 depicts the adjusted commutativity criterion.

models are considered equivalent for each valid feature configuration, commutativity will be satisfied.

As another note, the underlying implicit assumption of the commutativity criterion is that the source filter generates the correct result as well as the single-variant transformation that receives the filtered source model as input. These two assumptions are, however, two fundamental correctness criteria of the product line engineering tool and the transformation engine, respectively.

By Construction As a second possibility, consistency of annotations can also be ensured *by construction*. Tools fulfilling this criterion offer processes that maintain correspondences between models. As an example, the multi-view approach by Gomaa and Shin [GS02] considers each instance of a UML model type (e.g., activity and class models) as one view where corresponding elements are linked by the means of a correspondence model. The correspondence model not only links the elements of UML models but also a corresponding feature or variation point modeled in a variability model may be linked to the respective correspondence element. As a consequence, it is ensured that corresponding elements of different models are mapped onto the same annotation, which should result in the same effect as commutativity of model transformations. Similarly, edit operations during development may be restricted in a way that they force to link elements of different models explicitly and map them onto the same annotation.

4.2.2 Annotation Maintenance in MDPLE Solutions

After having introduced different classification criteria how annotations in one and across models may be maintained, this section examines how the functionalities are supported in existing MDPLE tools and approaches. Table A.1 in the appendix summarizes to which categories the approaches discussed in Sec. 4.1.2 belong. In addition, we have performed a formal concept analysis which indicates relationships between the properties and the resulting concept lattice illustrates the result graphically in Fig. 4.2.1.

It must be noted that most of the tool descriptions are for instances of one metamodels only. Therefore, whenever the tool descriptions are silent on the maintenance of multiple models, assumptions were necessary. In the majority of solutions, we assume that an automatic propagation or maintenance of annotations cannot be supported without further modifications whereas we assume that approaches realizing an annotation-wise mapping keep annotations consistent by (manual) construction.

Compositional Approaches Although the compositional tools and approaches introduced in Sections 4.1.2 do not apply an annotative variability mechanism (i.e., they do not realize negative variability), for deriving products they also need to incorporate feature traceability, e.g., in form of annotations.

Aspect-oriented product line modeling [GV09] serves as an example (primarily) realizing a compositional approach. Elements inside the models need to be declared as join points to be extended by an aspect. Therefore, this approach realizes an element-wise internal mapping. Since most of the languages of the oAW framework used in the AOPLE approach offer aspect-oriented language constructs and corresponding tooling, the approach can be categorized to *reuse* the existing modeling language for realizing the mapping of annotations. If more than one model forms part of the product line, annotations (or join points) will be declared manually. Consequently, no propagation occurs and consistency is not ensured neither. While **MATA** applies a different approach to realize aspect-oriented behavior by using Graph transformations, it also realizes an internal manual mapping without ensuring consistency. However, MATA is a *proprietary* language to declare aspect models each modifying (parts of) the base model. Although the base model may comprise multiple types of UML models, an aspect model may only modify one kind of a model. Thus, annotations may occur multiple times for different aspect models.

The approach to superimpose UML models [Ape+09] examines composition techniques for UML models. Mappings are applied internally by reusing UML profiles. Furthermore, the annotations are applied manually per module. An automated propagation of annotations between modules does not exist.

Transformational Approaches Delta-oriented tools, such as **SiPL** [Pie+15; Pie+19], **Delta-Ecore** [SSA14b; SSA14c] and **DarwinSPL** [NES17], as representatives of a transformational approach, extend one *core* model with *delta modules*. Due to the fact, that each delta module comprises an application condition and a set of delta operations that have to be applied to realize the functionality related with the application condition, the mapping is applied module-wise (i.e., element-wise) and can be regarded as *internal* because it is integral part of the delta module. For the reason that Ecore models which are supported in these tools do not allow for specifying delta operations, the tools employ their own delta dialects and languages to specify delta modules, consequently, building a *proprietary* mapping language. Since annotations have to be assigned to each new delta module *manually*, there is no automated propagation present.

A delta-oriented mechanism is also implemented in the tool **EASEL** [HJH06]. In this tool a base layer of a UML class model can be defined and modified by composing feature-specific layers. Thus, EASEL transforms the base layer into other products. Similarly, annotations are assigned per module, internally and a propagation of annotations is not discussed.

The variability modeling language **CVL** [Hau+08] incorporates annotations as proprietary language constructs for the transformation of the base model. A propagation of annotations is not foreseen but they are associated with resolution models completely manually. A consistency criterion is not applied neither.

Lastly, **VML*** [Zsc+09] offers the functionality to map annotations onto model elements of one target model in their overall metamodel. The language realizes an external, element-wise mapping which is specific to one target model. Therefore, a propagation of annotations is not supported nor a consistency mechanism.

Annotative Approaches persist mappings internally or externally: The mechanism based on model templates [CA05; CP06] incorporates all variants in the template (i.e., it is a multi-variant model in our terminology). The template instances are single products derived in a M2M transformation based on a feature configuration. The template stores annotations element-wise based on stereotypes provided in a variability profile, which are realized internally. A propagation of corresponding annotations among models is not discussed.

The proprietary language **Clafer** [BCW11; Bak+16; Juo+19] stores annotations element-wise (or module-wise) for each *claffer* which is a unit storing structural, behavioral elements as well as its annotations. A propagation of annotations across clafers is, to the best of our knowledge, not supported. Furthermore, the tools **FeatureMapper** [HKW08] and **Famile** [BS12] both employ *mapping models* which link elements of multi-variant domain models, with annotations. Consequently, both tools employ element-wise external mappings and originally do not support an automated propagation.

Tools that realize filtered or projectional editing of models, internally maintain multi-variant models. **SuperMod** [SW16] assigns annotations element-wise as part of the internally kept superimposed model and maintains them automatically upon commits. Annotations form part of the SuperMod metamodel. Therefore, they can be considered as a part of a proprietary language. While SuperMod performs an *automatic* propagation of annotations each time a modification is committed and the propagation occurs across different models (which can be part of the superimposed model), consistency between different models is neither ensured by the propagation algorithm nor by any consistency rule inside the superimposed model. Nevertheless, SuperMod realizes an *inter-model* propagation.

Hybrid Approach In a similar way a projectional editor for model-driven product lines allows to switch delta-oriented and annotative representations of the domain model [Reu+20]. Similar as SuperMod, internally the tool maintains a multi-variant model (called *variational abstract syntax graph (VASG)*) onto which annotations are mapped element-wise as part of the *proprietary* VASG. The projection exposed to the developer, reuses the capabilities of Ecore models. Handwritten rules, transform the user-visible representations, including the annotations, into the VASG and vice versa. Accordingly, an *automatic intra-propagation* is supported but not across different models. As a consequence, consistency checks across models may be missing as well.

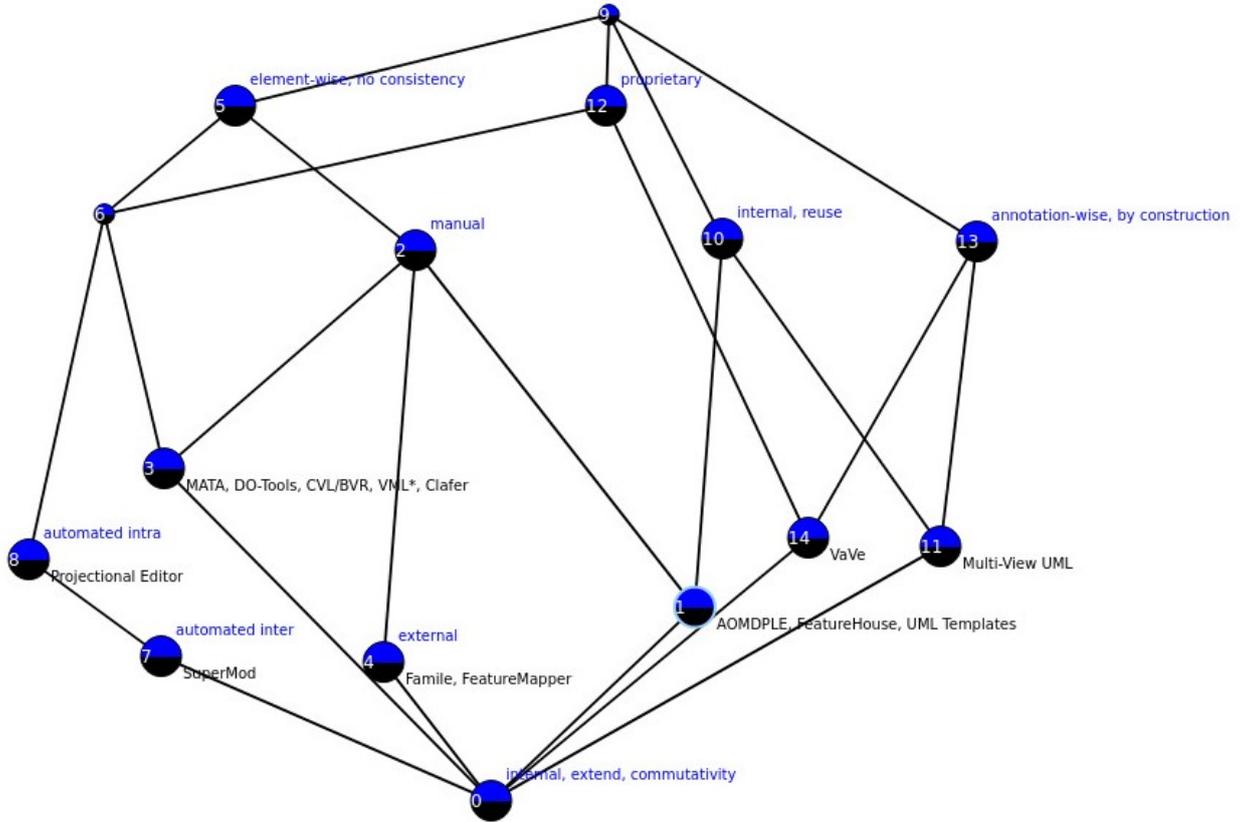


Figure 4.2.1: Formal concept lattice for mapping maintenance of MDPLE tools.

Multi-View Modeling Finally, multi-view approaches [GS02; Ana+18], explicitly deal with multiple models that are kept consistent upon modifications applied to one model. Goma and Shin [GS02] maintain a variability model incorporating variation points as one view next to other views, such as activity and class models. A correspondence model resides between all views and maps elements of various model views onto variation points. Thus, this approach provides an annotation-wise external mapping and does not need a propagation of annotations but the manual maintenance of linking the elements correctly with the variation points.

More specifically, **VaVe** [Ana+18] is an extension to the multi-view framework **Vitruvius** [KBL13]. VaVe realizes delta-oriented product line development. According to the VaVe metamodel, a feature (denoted as variant) is related with its implementation versions. Since for each variation point multiple versions may be present, it is an annotation-wise external mapping. Due to the connections of the multiple models based on correspondence rules designed in Vitruvius, a change in one model propagates automatically to corresponding models which in consequence obviates the need to propagate annotations.

4.2.3 Results

To sum the categorization of tools up, we performed a *formal concept analysis (FCA)* [GW99] which is an algebraic method to discover binary relations and logical implications of the attributes to the considered objects. In our case, the attributes are the features of the classification (c.f., Fig. 4.2.2) and the MDPLE solutions categorized in the previous section. The concept lattice depicted in Fig. 4.2.1 visualizes the result of the FCA which highlights the maximal properties shared by every solution and the minimal ones not realized by any solution.

As a result, the concept lattice and, thus, the categorization reveals that the way mappings are

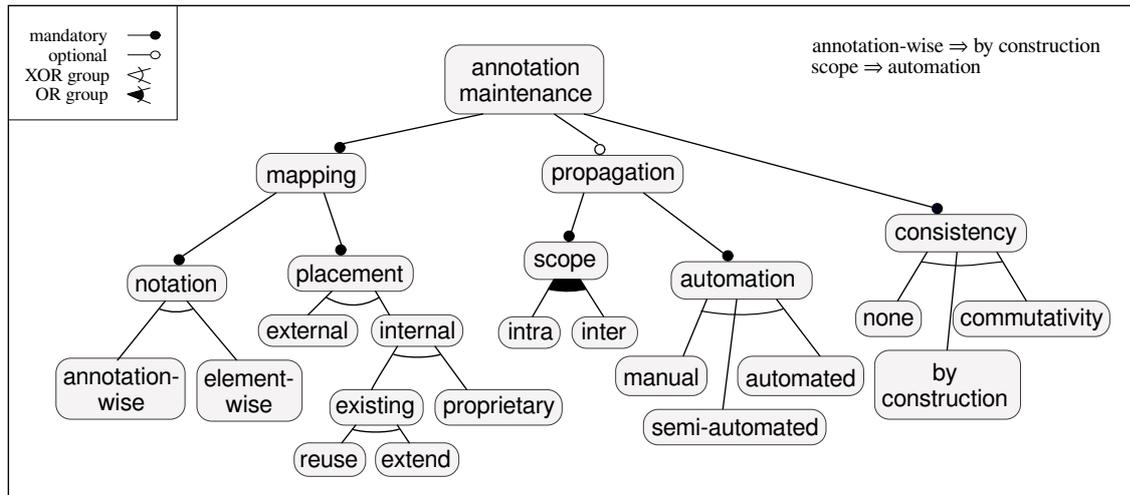


Figure 4.2.2: Feature-based classification of annotation maintenance in MDPLE.

persisted varies. Most notably, the bottom node of the lattice visualizes that no MDPLE solutions considers the criterion of commutativity but in addition only one tool, SuperMod can propagate annotations automatically across models. VaVe and the multi-view approach (on the right side of the lattice) are the only two approaches, ensuring consistency by construction which is an effect of employing annotation-wise mappings. Consequently, all solutions employing element-wise mappings do not ensure consistency (so far). However, the fact whether an internal or external mapping strategy is employed does not further affect the annotation maintenance.

On the whole, at the current state-of-the-art, none of the element-wise approaches can ensure that the annotation of corresponding model elements is consistent with an already existing annotation. On the one hand, it becomes obvious that none of the state-of-the-art solutions integrates automated means to propagate annotations across models. On the other hand, techniques, namely *multi-variant model transformations*, have been developed to solve exactly this task and are introduced in the following section.

4.3 Multi-Variant Model Transformations

As second main building block of MDSE, MDPLE should employ and reuse *model transformation* technology, for instance, by making it variability-aware. While several MDPLE solutions utilize model transformations to derive customized products from product line models or languages, hardly any regards an automated consistency maintenance of annotations across product line models as explained in the previous section. *Multi-variant model transformations* target exactly this problem.

However, the term *multi-variant model transformation* is ambiguous: Either the transformation itself becomes variable to create varying target models or the transformation is reused to propagate annotations to the (single) target model. Before Sec. 4.3.3 and Sec. 4.3.4 illuminate both meanings, respectively, this section sheds light on reusing model transformations in different use cases in general in Sec. 4.3.1. Furthermore, Sec. 4.3.2 describes properties of a feature-based classification which allows to categorize multi-variant model transformation approaches. Accordingly, each of the approaches introduced in the following two sections is classified and the results are combined and formally analyzed. The resulting concept lattice and the derived insights close the section in Sec. 4.3.5.

4.3.1 Model Transformation Reuse

Before scrutinizing multi-variant model transformations for product line models which reuse state-of-the-art technologies, the following paragraphs discuss model transformation reuse in general.

We focus on giving insights which transformation reuse scenarios exist and reference publications which offer and compare different solutions for reusing model transformations in general.

M2M Reuse Survey Model transformation reuse, in general, aims to *reuse* an existing transformation for more than one transformation scenario without the need to rewrite the transformation entirely. Kusel et al. [Kus+15] extract six essential reuse scenarios along the three dimensions *granularity*, *specificity* and *scope*. Granularity denotes whether small parts, such as single rules, or large parts, such as modules offer reusable source code. Specificity refers to the fact whether the transformation language supports a generic type system or one concrete metamodel whereas the scope allows for exchanging at least one metamodel of the transformation. The six scenarios are compared with respect to the classical phases in software reuse: *abstraction*, *selection*, *specialization* and *integration* [Kru92]. The findings of comparing the by then state-of-the-art in reusable model transformation technology for realizing the aforementioned software scenarios reveal that the mechanisms support generic reuse only restrictively. For instance, solutions are trimmed for certain kinds of metamodels or one specific transformation language and specialization support is hardly offered. Extending a transformation to allow for propagating annotations in a product line is not considered but may only restrictively and not generically be possible with the considered transformation approaches.

Feature-Based M2M Reuse Classification A recent classification on reusing transformations across metamodels [Bru+20] offers further classification criteria in form of a feature model and takes surveyed community needs into account. In the same way as reuse mechanisms in classical programming can rely on language-specific support, such as subtyping or genericity [Che+16], similar concepts can be found or integrated into model transformation languages as well. Still, these mechanisms do not allow for transforming a product line model into another or to create a family of products by regarding variability which is the most relevant form of reuse for the purposes of this thesis.

Model Transformation Product Line Research on *model transformation product lines* recognizes the variety in input and output models and the resulting variety in the model transformation definitions [Lar+18]. Therefore, the varying transformation can be represented as a product line. However, a model transformation product line does not solve the problem targeted with this thesis.

4.3.2 Classification

Transformations addressing the additional dimension of variability inherent in product line models have in common that they *reuse* existing model transformation engines to create the target model. The way, how annotations are attached to target model elements varies and can be classified according to the features presented in Fig. 4.3.1. This section illuminates each of the features from left to right and elaborates on the level of automation as prerequisite.

Automation Before discussing the capabilities of different solutions, the degrees of *automating* the maintenance of annotations, as defined in Fig. 4.2.2, have to be recapitulated. The bottom line of automating the annotation of target models is an entirely *manual* process. Consequently, neither the model transformation is made variability-aware nor another (customized) technique is utilized. In automated solutions the propagation of annotations across models works completely automatically. The product line developer does not have to intervene. As a consequence, a semi-automated propagation tries to compute annotations automatically first, followed by involving the product line developer in case of uncertainties, such as ambiguities or missing information, to determine the annotation for specific model elements.

Genericity On the left of the feature model depicted in Fig. 4.3.1, the first feature *scope* defines to what extent the solution is generically applicable with respect to the transformation *language* and the transformation *definition*.

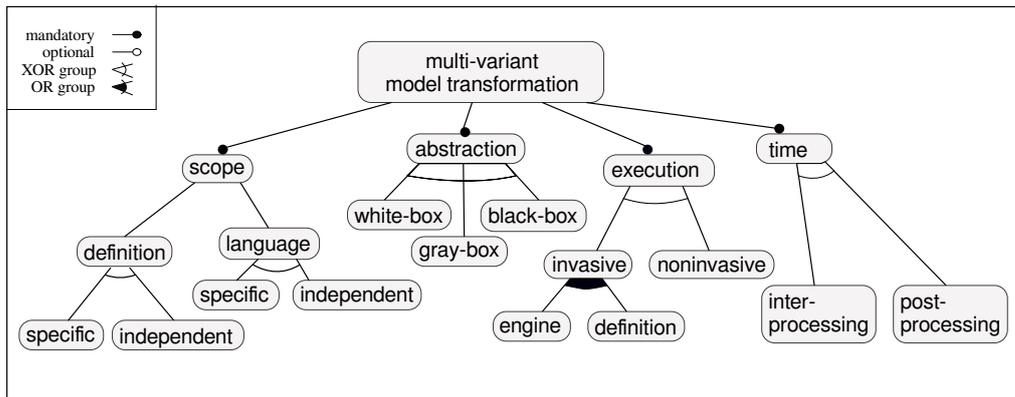


Figure 4.3.1: Feature-based classification of multi-variant model transformations.

Firstly, definition-specific (metamodel-specific) solutions are able to propagate the annotations of one kind of metamodel to another one. Thus, these solutions are able to support all transformations creating instances of these metamodels independent of the used transformation language. Such approach can be realized, for instance, by employing a DSL to specify corresponding elements of the source and target metamodel and iterating the input and output models accordingly when assigning annotations.

Secondly, a *language-independent* solution can propagate annotations independent of the model transformation language which creates the target model whereas a *language-specific* solution is trimmed for supporting the propagation of annotations for a specific transformation language only. Language-specific solutions either extend or manipulate the transformation language, the specification or the execution environment.

Abstraction As different degrees of abstracting the multi-variant transformation, *white-box* solutions are distinguished from *gray-box* and *black-box* solutions.

A white-box solution interacts with the transformation specification directly either by extracting information of the specification, manipulating it or by adapting the execution environment. In contrast, a gray-box solution does not analyze the transformation specification but exploits the artifacts created during or for the execution. For instance, a trace written during the transformation allows to draw conclusions on the specification but exploiting this information does not interfere with the execution engine nor requires manipulations or an analysis of the transformation specification.

The least invasive solution does not interfere with the transformation engine and specification. Instead, the transformation is treated as a black-box. Based on other knowledge, e.g., the types of transformed models, corresponding elements have to be determined and annotations propagated accordingly.

Execution As another point, the way, how the *execution* is performed, varies. Either the execution is performed in a *noninvasive* way, i.e., the execution environment is not manipulated at all. Differently, in *invasive* solutions either the transformation *specification* is changed (e.g., by extending the syntax to become variability-aware) or the execution engine is manipulated to become variability-aware. Black-box solutions imply a noninvasive execution because they consider neither the transformation specification nor the execution engine and consequently do not manipulate them.

Point in Time Last but not the least, *multi-variant model transformation* solutions can be classified by the point in *time* at which they assign annotations to target elements. On the one hand, the solution can attach annotations already during the execution of the reused transformation which we refer to as an *inter-processing* approach. On the contrary, particularly most of the noninvasive solutions assign the annotations after the single-variant transformation has been performed in a *post-processing* step.

Note: *Preprocessing* of annotations, for example in form of a static analysis of the transformation specification, can be considered one means to compute annotations but the actual assignment to target elements can occur earliest when the target elements are created.

4.3.3 Variation in Transformation

While this thesis offers methodologies to reuse single-variant transformation specifications as they stand, related work focuses on extending existing model transformation languages by variability-aware rules. This class of “multi-variant” model transformations varies the transformation execution. Respective solutions offer possibilities to express the variability inside the transformation rules themselves. During the execution of the transformation the variability is bound resulting in different target models. Although this functionality does not solve the problem of propagating the annotations to the target model and thus does not create an annotated multi-variant target model, this section introduces different solutions for incorporating variability in the model transformation. The main approaches which we identify in this research area are variability-based rules and higher order transformations. The upper part of Table 4.2 summarizes how the three main approaches can be mapped onto the multi-variant model transformation categories which is explained subsequently in this section. Since these approaches do not propagate annotations but bind variability before the execution, the *time* category is not applicable.

Variability-Based Rules The variability-based rules [SS16] serve as first example where variability is explicitly encoded in the model transformation. Those rules are implemented as a tool extension to *Henshin* [Are+10b; Str+18a]. Although the approach supports filtered editing of the rules, variability has to be expressed and considered explicitly in the transformation definition. On receiving a feature configuration, the variability is bound and a corresponding target created. In contrast to others, the matching of a variability-based rule exploits the variability information to speed up the execution by first matching the common parts and the distinguishing parts thereafter [Str+17]. The rules and their matching are formally specified [Str+18b] as algebraic graph transformations [Ehr+08].

Although variability-based rules do not propagate annotations to the target model, they are aware of the variability of product lines and can be categorized as a language-dependent, definition-independent white-box solution to handle variability because they require to adapt the transformation engine and to extend the definition.

Aspect-Orientation Employing aspect-oriented programming techniques to realize model-driven product lines is discussed in Sec. 4.3.3, I. Particularly, the concept of employing an aspect for each feature represents one means to compose a transformation yielding customized products when binding the variability. However, the solution of Groher and Völter [GV09] does not aim for generating a multi-variant model. Employing a generic aspect to generate multi-variant source code is a contribution of this thesis presented in Chp. 8.

Higher-Order Transformation Alternatively, variability rules can employ *higher order transformations*. A higher-order transformation denotes a model transformation creating another model transformation [Tis+09].

Variability Rules in ATL Similar to other approaches that employ a higher-order transformation to *derive products* [OH07; BOT07] (from the same model), the solution by Sijtema et al. [Sij10] creates varying target models by extending ATL rules to integrate conditional compilation directives. In contrast to, for instance, the lifting approach (Sec. 4.3.4), the execution semantics of the state-of-the-art ATL execution environment remains unchanged. Instead, the variability-aware transformation definition is transformed by a higher-order transformation into a “default” ATL definition by a transformation written in ATL. On receiving a feature configuration and a source model, an ATL transformation translates the variability-based transformations into state-of-the-art ATL transformations where only positively selected parts are included. The respective

Table 4.2: Classification of multi-variant model transformation approaches in related work based on features in Fig. 4.3.1. The first part of the table enumerates approaches which vary the transformation to yield customized products whereas the second part regards solutions which propagate annotations. All approaches are white-box solutions because they change execution semantics of existing single-variant model transformation approaches.

Ref	Description	Scope	Realization	Time
[Str+15]	Variability-Based (VB) Rules	definition-independent, language-specific,	invasive, engine	–
[Sij10]	HOT: variability rules in ATL	definition-independent, language-specific	invasive, definition & engine	–
[Kav+11]	HOT: MTS	definition-independent, language-specific	noninvasive	–
[Sal+14; Fam+15]	Lifting	definition-independent, language-specific,	invasive, engine	inter
[Tae+17]	Formal Framework Based in Category Theory	definition-independent, language-specific	invasive, engine	inter
[SPJ18]	Staged-Strategy (Lifting + VB Rules)	definition-independent, language-specific	invasive, engine	inter

publication does not discuss limitations of the approach, for instance, the kind of annotations and product line scenarios which are supported.

The solution works language-specific but independent of the definition. While it does not require to modify the default ATL execution engine, it extends the ATL syntax (i.e., in an invasive way) and offers a preprocessing engine that binds variability and converts the extended ATL into the default ATL transformation. Thus, it is a inter-processing white-box solution which does not propagate annotations but builds varying target product models. The transformation definition is extended and the engine, too.

MTS The *model transformation system (MTS)* [Kav+11] employs higher order transformations to generate customized transformations, too. Similar to variability-based rules, the purpose is to create platform-dependent models from platform-independent ones. After defining commonalities and the variability in the transformation, a *variability metamodel* is created which captures parameterized rules in the form of a metamodel. Instances of this model represent a customized but still parameterized transformation which is input to a second transformation. The second higher order transformation creates the concrete transformation for deriving one specific product. On the whole, this approach does not target and therefore not support the propagation of variability information among models.

As classified in Table 4.2, the MTS higher-order transformation solution works independently of a specific definition but is implemented in one specific language. It is a white-box solution which requires to integrate the higher order transformations in the execution engine of GReaT [Kar+03] and to extend the default syntax.

4.3.4 Annotation Transformation

The second way to parse multi-variant model transformations considers the propagation of the annotations to the target model. Besides the contribution of this thesis, lifting and two formal frameworks represent this kind of transformation and are explained in this order. The lower part of Table 4.2 summarizes the results of mapping the approaches in this section onto the features of multi-variant transformations as defined in Fig. 4.3.1.

Lifting of Transformation To the best of our knowledge, Salay et al. [Sal+14] proposed the first solution applying a single-variant model transformation to a product line in order to include annotations in the target models. The authors formalize a *Lifting* algorithm for a single-variant *in-place Graph* transformation involving negative application conditions (NAC) to become variability-aware and create an annotated target model.

The lifting algorithm, which describes how a single rule can propagate annotations, behave as follows: The product line, the set of constraints, the rule and a matching site inside the domain model of the product line are input to the algorithm and modify the input product line. Applying the lifted rule requires that the constraints of the product line as well as the application constraint ϕ_a , composed of a negated disjunction of the constraints' NACs combined with the constraints of the elements that are maintained and deleted. The algorithm, then, maps the annotation ϕ_a to each added element and $\neg\phi_a$ is combined in a conjunction with the elements' constraints to each deleted element. If the resulting constraint together with the set of constraints of the product line is not satisfiable, the element will be removed from the product line.

The algorithm is formally proven to be correct for typed graphs and in-place transformations. However, Lifting requires to modify the execution semantics of a rule application which is implemented as extension to Henshin [Are+10b; Str+18a]. As a consequence, supporting the lifting of a rule requires to change the transformation engine. Moreover, the solution is restricted to in-place Graph transformations. A transformation, for example specified in ATL, is not guaranteed to be lifted by employing this algorithm.

Despite the fact that Lifting is defined for in-place Graph transformations, the algorithm was integrated in the execution engine of **DSLTrans** [Bar+10], a graph-based domain-specific out-place transformation language. The solution was successfully applied to transform one product line model into another one by retaining the original product line model in the context of the automotive industry [Fam+15]. Similarly to the basic lifting algorithm, the integration in DSLTrans requires to change the semantics of its execution environment.

As depicted in the first line of Table 4.2, both inter-processing solutions are language-specific but propagate the annotations independently of a concrete definition. Since they require to modify the execution semantics by manipulating the engine, they are categorized as white-box solutions.

Category Theory Taentzer et al. [Tae+17] propose a formal framework specified in category theory for transforming one product line into another, including the evolution of the variability model. For instance, the framework represents the Lifting algorithm as a special case (without evolution). In the formal framework not only a product line serves as input but also the transformation rules involve variability. Therefore, the negative application condition, the left hand-side and the right-hand side may specify the set of features and constraints and may encompass annotated domain model elements. Due to its restriction to injective morphisms, only in-place Graph transformations and simplified feature model evolution scenarios are possible. For instance, splitting of assets [BTG12] cannot be expressed.

Even though an implementation of the framework is not yet published, it can be considered as language-specific, definition-independent white-box solution because it either requires to extend an existing Graph transformation engine or to develop a proprietary tool satisfying all conditions.

Staged Transformations On the contrary, a *staged strategy* of combining variability-based rules and the lifting algorithm can be applied to transform product lines [SPJ18]. The strategy mitigates the drawbacks of lifting and variability-based rules to enumerate all rules or all products, respectively, to transform a product line. It behaves as follows: first, a common base rule is matched with the annotated domain model, followed by a search for the specific rule that needs to be applied. The respective rule is lifted to the product line as described in [Sal+14]. The equivalence of the resulting products, i.e., commutativity, is proofed based on algebraic graph transformations. Accordingly, the method is applicable to rudimentary Graph transformations (without negative application conditions or amalgamation). Furthermore, features need to be expressed explicitly in the rule set. To this end, the user effort is higher due to the cognitive complexity introduced in variability rules. Nonetheless, avoiding the need to iterate over single products or rules confirms expected runtime savings. Since the solution is a combination of lifting

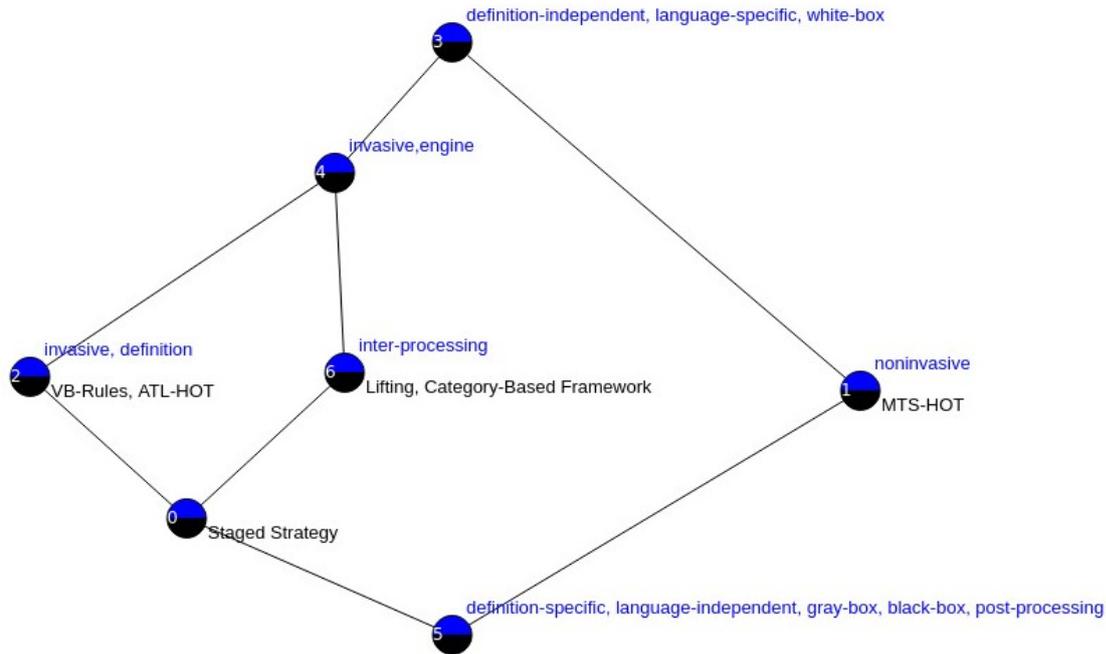


Figure 4.3.2: Formal concept lattice for multi-variant model transformation properties of related solutions.

and variability-based rules, it is grouped in the same classification categories. Consequently, the staged strategy is a white-box, definition-independent but language-specific solution which is performed during the transformation in an modified transformation engine.

4.3.5 Results

Table 4.2 collects mappings of the related multi-variant model transformation solutions onto the feature-based classification criteria. This section presents a formal concept analysis which was performed in the same way as in Fig. 4.2.1 but for these mappings of multi-variant model transformations. At the bottom, the lattice shows the properties which cannot be mapped onto any of the approaches whereas the top node shows the properties which are shared among all approaches.

Annotation Propagating Approaches The descriptions in this section reveal that in total only three related solutions realize the automatic propagation of annotations based on a reused single-variant model transformation. Moreover, these solutions, namely lifting, the category-based framework and the staged strategy, require to manipulate the execution semantics of the reused transformation engine. Consequently, these invasive solutions realize the assignment of annotations during the actual model transformation execution (inter-processing).

Further Properties On the upside, all solutions are definition-generic, meaning that the propagation of annotations can be performed on instances of arbitrary metamodels. On their downside, they work for one transformation language or class of languages (specific graph transformations) only. Except for MTS, all of the approaches represent white-box solutions which modify either the transformation engine or its semantics (i.e., they are invasive).

On the contrary, several contributions, such as variability-based rules, explicitly incorporate the variability dimension in the transformation rules and, thus, expose the product line developer to a higher level of cognitive complexity. Moreover, the variability in these rules is bound to a specific configuration during the execution, thus, yielding single-variant models and not a superimposition of them which would be required for adequate development and evolution support in MDPLE.

The lattice further demonstrates that only MTS does not invade in the execution environment but cannot be considered a black- or gray box approach because it requires to manually derive and configure a variant of the transformation that should be executed. Not analyzing the transformation but using one of its artifacts, namely the trace, for propagating annotations is one of the main contributions of this thesis.

4.4 Bottom Line

All in all, this chapter illuminates the consequences and possibilities of fusing model-driven software engineering and software product line engineering which means to develop product lines in a model-driven way.

On the one hand, models have to be enriched by variability information which is realized in varying ways in MDPLE solutions. On the other hand, model transformations can be employed in MDPLE solutions not only for deriving customized products but also to keep different models (originating from developing the product line in different stages and for different customer concerns) in one product line consistent. However, only few existing solutions consider an automatic maintenance of variability information across different models. In addition, the already existing solutions are not employed in any of the considered MDPLE and typically require to modify existing model transformation execution semantics or require to declare the variability in the model transformation specifications directly. This results in adaptation costs and higher complexity due to the additional dimension of variability which may be the reason why an integration is still missing.

Part III

Trace-Based Propagation of Variability Annotations

Chapter 5 Informal Properties of Trace-Based Propagation

*Ideas that you'll never find
All the inventors could never design*

Coldplay, Speed of Sound

Many transformation approaches and software development tools record corresponding elements of different artifacts as trace(ability) information. Given this observation, we utilize the persistent information about corresponding elements of a source and target model to propagate the annotation of source to corresponding target elements. Therefore, this chapters presents an informal example illustrating an out-place model-to-model transformation which records a *trace* of corresponding elements while being executed. We employ the persisted trace information to propagate annotations from the source to the target model of the example.

The example (Sec. 5.1) demonstrates model-to-model transformation rules which create a Java model from a class model. First, it illustrates the transformation of single-variant models and extends the models with variability information to become multi-variant, thereafter. Before Sec. 5.3 explains how traces can be used to propagate annotations and which conditions have to be satisfied for a correct result, Sec. 5.2 illustrates different types of traces and classifies them to derive a common trace meta-model which we employ for propagating annotations.

The chapter shares material with [GSW17], [WG18], [WG20a] and [WG20b] which lay the foundations for its contents.

5.1 Example of Trace-based Transformation

For exemplifying the formalization presented in the next chapter with a concrete scenario, this section introduces a multi-variant UML class model to Java transformation. The UML class model for the database contents product line, depicted and introduced in Fig. 4.1.2, serves as source model in a reduced form.

Instead of considering all classes and, thus, all variants, in the first step (Sec. 5.1.1) the example presents the transformation of one class and one package into corresponding elements of a Java model. Thereafter, Sec. 5.1.2 presents the transformation rules that are necessary to create the target Java model. Finally, Sec. 5.1.3 extends the example to represent a small multi-variant model in the same scenario and the result of applying the same rules to this example.

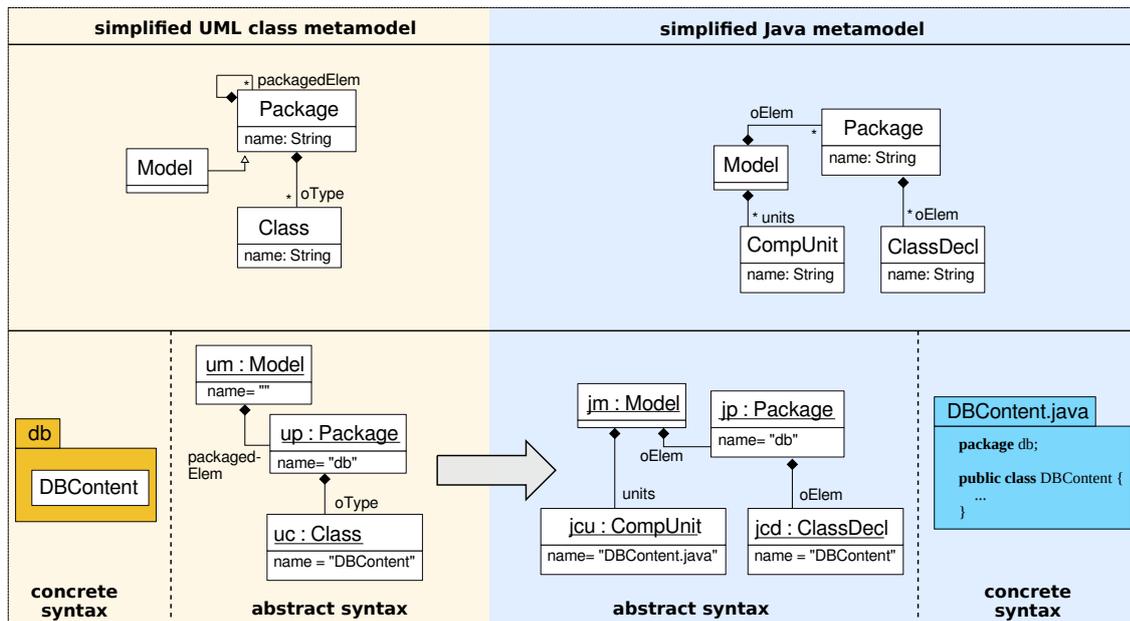


Figure 5.1.1: UML class and Java MoDisco model excerpt representing the database contents.

5.1.1 Single-Variant Model

Fig. 5.1.1 presents the contents of the example transformation which transforms a UML class model into a corresponding Java model. The top of the figure introduces the simplified UML class and Java metamodels to which the models, depicted at the bottom, conform.

On the left side, the top depicts the simplified UML metamodel for class models consisting of a package hierarchy. The `Model` is designed as a special kind of package and classes are stored in the packages as `o(wned)Types`. Packages as well as classes are named elements in the UML metamodel. The bottom depicts an instance of this metamodel in concrete and abstract syntax. The model comprises two packages, one being an instance of a `Model` (object name: `um`), and the other one holding the class named `DBContent`.

The right side of the figure illustrates the simplified Java metamodel which is an adaptation of the Java MoDisco metamodel [Bru+10]. The class `Model` serves as root which contains packages but also all compilation units (`CompUnit`) in which type declarations are actually implemented. The only kind of type declaration are class declarations (`ClassDecl`) stored in packages which corresponds with the UML side where only classes are present. In the original metamodel, a reference between compilation unit and the type declaration exists as well as an indirection via access classes (`TypeAccess`) which are left out from this example for the reason of easier readability. At the bottom, the right side presents an instance of this metamodel which corresponds to the UML class model (and can be created by a model transformation). The UML model (instance

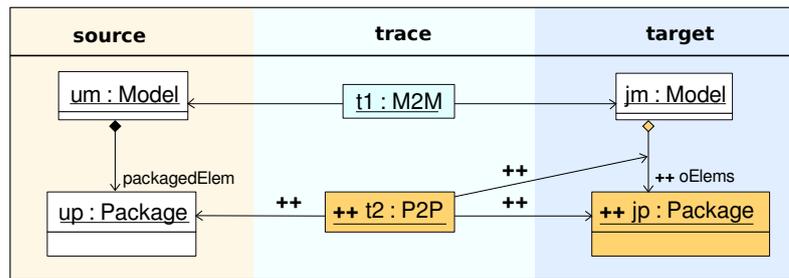


Figure 5.1.2: Graphical rule which transforms UML packages into Java packages. The container `Model` has already been created. `++` signs mark the target elements created by the rule.

name: `um`) is represented by a Java model (instance name `jm`). Both models contain a package with the same name whereas additionally the Java model comprises a compilation unit (instance name: `jcu`) for the class declaration (instance name: `jcd`) corresponding to the UML class `DBContent`.

5.1.2 Example Transformation

This section describes the transformation rules to create a Java model from a given UML class model. The subsequent paragraphs describe all rules at the level of objects while also establishing links between these objects. Names of the elements are assigned correspondingly in the real-world transformation but not explicitly mentioned in the example. Furthermore, the rules compose a source and a target side as well as a trace section. Trace elements map source objects onto the created target objects and are maintained by trace-generating transformation engines, such as the Eclipse QVT-O plugin or the ATL/EMFTVM, in addition to the target model.

Transformation of Model A forward batch transformation behaves as explained in the following paragraphs. The initial step transforms the UML `Model` into a Java model. This rule is not explicitly shown because of its simplicity. A trace element created by this rule references the UML model as source and the Java model as target element, respectively.

Package To Package After having transformed the UML model, the transformation creates a Java package for each UML package. In this example, the rule does not construct the package hierarchy but integrates all packages in the single model of the respective transformation side. Fig. 5.1.2 presents the source and the target graph corresponding to the example. The rule marks elements and references that are created anew with `++` symbols and highlights them in orange color.

To generate containment references properly, it is essential that the Java model corresponding to the UML model as well as a corresponding trace element have already been created in the first execution step of the transformation. Consequently, the transformation rule depicted in Fig. 5.1.2 creates not only a similarly named Java package for the UML package but also a containment reference originating from the Java model and ending in the Java package for storing the package in the Java model.

Moreover, in a trace-generating transformation not only the target side is extended but also the trace section by adding a new trace element. Accordingly, the figure shows that the `P2P` object is added, too. This trace element references the UML package as source element and the java package as well as the containment reference as target elements.

Class To ClassDeclaration The third transformation rule, depicted in Fig. 5.1.3, transforms each UML class into a class declaration and a compilation unit. While the class declaration is stored in the Java package, the Java model serves as container for the compilation unit. In this example the transformation rule refrains from creating a link between the compilation unit and the class (which is necessary in the real-world scenario) as well as from regarding the possibility to store more than one type declaration in a compilation unit. In addition, the rule refrains from creating an interface for each UML class for the sake of easy readability of the example.

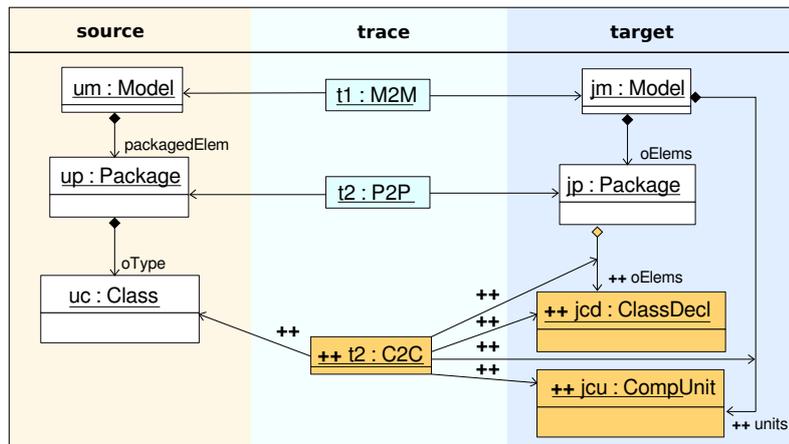


Figure 5.1.3: Graphical rule which transforms UML classes into Java class declarations and compilation units.

Trace Access Most notably, after having performed the step of creating the Java package, the transformation may access the new trace element between the UML and Java package created by the rule transforming packages. In this way, the transformation can add new elements to previously created elements in the target model, namely the class declaration to the package and the compilation unit to the model. Besides accessing existing trace elements, this rule extends the trace graph with a new trace element referencing the four elements created anew and the corresponding source element.

Please note: Although the rules need to access already existing trace information and existing target elements to properly store the elements added by the rules, these access relationships are not made explicit in the rules (yet) for the sake of easier readability of the example.

5.1.3 Multi-Variant Model

After having explained the basic (single-variant) rules in the previous section, the following paragraphs demonstrate an extended input model with annotated varying contents.

The example is extended by the classes **Person** and **Family** added to the UML class model. In addition, we simplify the feature model of the database product line example (Fig. 3.3.1) such that it encompasses the mandatory feature **P**(erson) and the optional feature **Fa**(mily) which are part of the database (DB) root feature. The left of Fig. 5.1.4 depicts the extended UML class model. This figure shows that the three classes are turned into corresponding class declarations and compilation units. In contrast to the transformation rules, the names of the class declaration and compilation units are assigned, too, to distinguish the objects more easily.

From the figure, two points become obvious: Firstly, in contrast to the above rule descriptions the trace references also *context* elements besides the target elements created by the corresponding rule application. As an example, the rule to create a class (Fig. 5.1.3) only references the class declaration and the compilation unit as well as the containment references as target elements. However, for adding the containment references (**types** and **units**) to the correct container, also the package and the model have to be accessed by the rule. For that reason, accessing already existing elements in the target model (i.e., elements that have been created in a previous execution step) is marked in the trace element by a reference to these *context* elements sketched as dashed lines in Fig. 5.1.4. Accordingly, execution traces may vary with respect to the granularity by which they reference elements in the target model. Different kinds of traces written during a model transformation execution are discussed in Sec. 5.2.

Secondly, although the source model is annotated, the rules presented in Sec. 5.1.2 are unaware of the variability occurring in product lines and offer no means for propagating the annotations to the target model. Consequently, a multi-variant target model (without annotations) can be created by employing a state-of-the-art (single-variant) model transformation. However, to automatically annotate the target model, further extensions are required. As explained in the subsequent de-

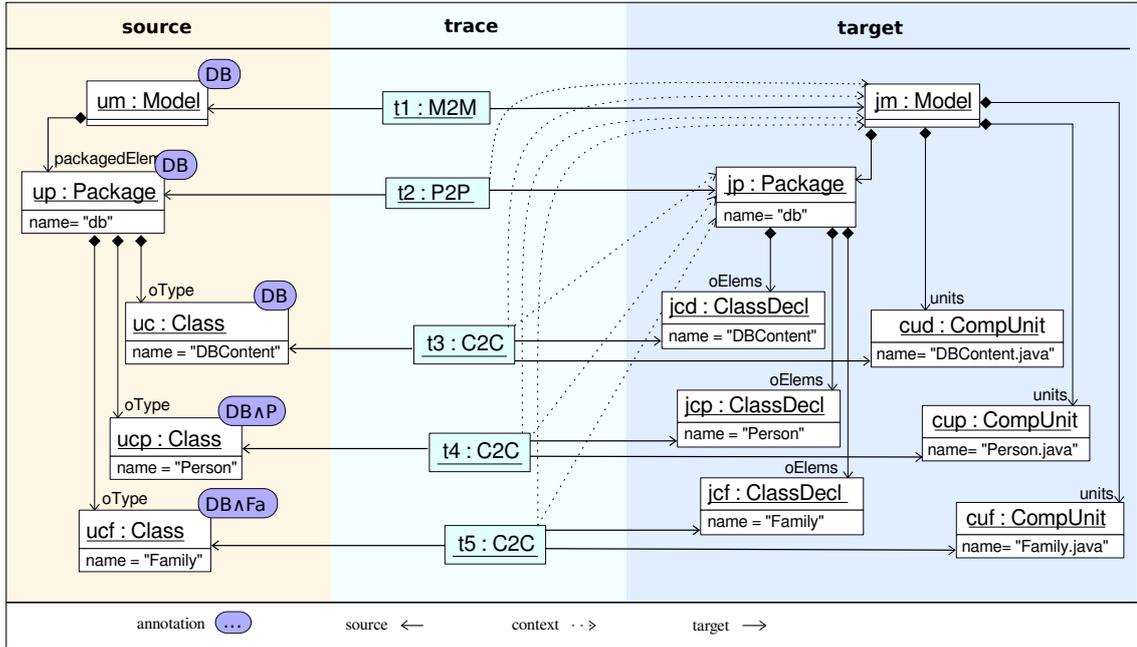


Figure 5.1.4: Rule applications to multi-variant model in triple-graph representation.

scriptions, the generated trace elements can be employed to propagate the annotations without a need to change the single-variant model transformation nor its execution environment. Important requirements to successfully and beneficially propagate annotations by employing trace information are summarized in Sec. 5.3.3.

5.2 Properties of Transformation Traces

Due to the plethora of model transformation languages and tools, various kinds of traces exist in M2M transformation execution engines. By examining trace-generating model transformation solutions in Sec. 5.2.1 and deriving a feature-based classification (Sec. 5.2.2), this section gives an overview of their differences and commonalities. Based on these properties, a common *trace model* is derived (Sec. 5.2.3) which is trimmed to be suitable to define a generic algorithm for propagating annotations. Arbitrary traces can be transformed into the common trace model to be used in the propagation algorithm.

Please note: Here, we only consider traces, which result from applying a M2M transformation and record corresponding source and target elements. *Traceability frameworks* [Aiz+06; BBM05; WP10], particularly those customized for MDSE [Anq+10; SHG12], go beyond the interests of the trace-based propagation by relating multiple kinds of artifacts and, for instance maintained in a database, and by allowing for querying the traceability information and further maintenance activities. For that reason, they not considered in the following discussions.

5.2.1 Traces in Existing Model Transformation Solutions

Several transformation solutions maintain traces during their execution. Due to the multitude of model transformation solutions (c.f. Sec. 2.2.2), the information persisted in the maintained traces is of manifold granularity, too. In this section, we examine the traces of exemplary model transformation engines as representatives of different possibilities to note information of corresponding source and target elements.

QVT [Obj16] The OMG proposed the QVT framework (c.f., Sec. 5.2.1, III) as standard for incremental bidirectional M2M transformations. The QVT-R specification establishes the declarative

language in this framework. Even though the standard does not prescribe a trace for realizing the incremental transformation, the stable tool realization *medini QVT* [ikv18] builds traces during the execution and persists them thereafter. Traces encompass a sequence of instances of relations (which are comparable to rules in other model transformations solutions). A rule instance records the relation which it instantiates as well as the source elements for which it was triggered and the resulting target elements. The list of target elements consists not only of the elements generated anew but also of already existing (context) elements in the target element which had to be accessed to create the new target elements.

In contrast, the Eclipse plugin which realizes the procedural language QVT Operational Mappings (QVT-O) manifests trace information, too. The trace may record context elements if they are provided as input to the mappings (i.e., rules) and all target elements which are explicitly defined as output parameters. The persistent information may vary with respect to how the rules are defined as Sec. 5.2.1, I discusses in more detail.

ATL (EMFTVM) [Wag+12] As a second model transformation language, ATL [Jou+08] allows for specifying in-place and out-place unidirectional model transformations which are realized by different virtual machines. The default virtual machine of the ATL distribution supports out-place transformations and incremental in-place transformations.

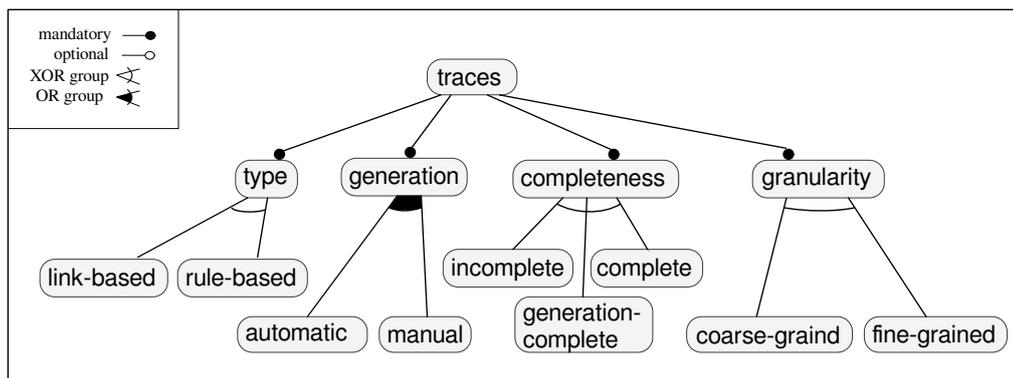
Although during the execution of a transformation specification a trace is maintained and can be accessed when specifying the rules, the trace is not persisted after the execution. For that reason, in this work we employ the ATL/EMFTVM as virtual machine which offers to persist the execution trace and additionally provides an instruction model of the bytecode. Despite manifesting the trace, still the available capabilities of this virtual machine are more restrictive than in the default ATL virtual machine. Particularly, ATL/EMFTVM does not support incremental transformations nor the usage of helpers hindering the specification of complex transformations.

An example of a ATL/EMFTVM traces is presented in Sec. 5.2.1, II. These traces record the source and created target objects of each applied matched rule. The trace elements do not record context elements which are required to create these element nor mappings of structural features. Similarly, applications of lazy and called rules are not persisted.

BXtend [Buc18] is a framework offering to implement bidirectional, incremental transformations in a way strongly inspired by triple graph grammars. Rules, in which the transformation developer has to explicitly maintain a correspondence graph between the source and the target model, are implemented in the object-oriented Java dialect Xtend [ES]. Accordingly, the correspondence graph represents the trace information which in its default version maps only one source onto one target element regardless of how many elements are created or accessed to create the new elements. Therefore, the trace consists of correspondences which are 1:1 links.

eMofflon [LAS14; LAS15] serves as an example of a language and corresponding tool which persists two kinds of trace information. Triple graph rules may express incremental bidirectional model transformations. Consequently, similar to BXtend, a correspondence graph is maintained between the source and the target model in which each node (i.e., a link) stores exactly one source node and one target node. Besides the correspondence graph, however, eMofflon builds a *protocol* during the execution which is persisted thereafter. This data structure records a partially ordered set of rule applications which store the match as well as the elements that are created including context elements that were necessary for the new creation. We consider the protocol and the correspondence graph as two different types of traces.

Further M2M Transformation Solutions Finally, several other transformation solutions exist which permanently store trace information. Kahani et al. [Kah+19] distinguish tools that automatically create trace information, such as *medini QVT* or the *DSLTrans* [Bar+10], from those allowing to manually define trace links in the transformation rules, such as in *Henshin* [Str+18a]. Few tools, such as eMofflon, allow for both, manual trace generation, i.e., the correspondence graph, and automatic trace generation in form of the aforementioned protocol. But the details and properties of the traces apart from their generation are not discussed in this article.

Figure 5.2.1: Feature-based classification of M2M transformation *traces*.

The granularity of the trace written by the transformation engines may depend on the type of employed rule (e.g., in ATL and QVT-O (c.f., Sec. 10.2.3)) and, for instance in Henshin, on the developer who has to specify the creation of trace elements. Accordingly, target elements can be stored by potentially differentiating context from target elements at the level of objects. In contrast, if the correspondence graph is extendable, the granularity of mapping source onto target elements may be refined. Accordingly, the possibility of specifying sublinks to relate structural features of objects can and has been used to offer more fine-grained tracing [Bec+07]. The customization of traces is also offered by the BxtendDSL [BBW21] framework, which extends the basic Bxtend framework with higher automation. However, the customization also requires manual maintenance of these specific kinds of links and is not examined by the tool developers yet.

5.2.2 Feature-Based Trace Classification

As a consequence from the previous descriptions, we classify traces maintained in M2M transformations based on the categories depicted in Fig. 5.2.1. We apply this feature-based taxonomy to the traces resulting from the M2M transformation languages described in the preceding section. Table 5.1 collects the resulting mapping.

Trace Type Firstly, traces are either *link-based* or *rule-based*. While rule-based traces result from applying rules and maintain a link or reference to the applied rule, a link-based trace stores links between up to many elements of the source model and up to many elements of the target model without necessarily mentioning the rule in charge for their creation. ATL/EMFTVM and medini QVT offer rule-based traces whereas Bxtend and eMoflon require the transformation developers to specify the creation of the correspondence graph during rule execution. The correspondence elements per se are not related with the rule creating them. eMoflon’s protocol originates from applying the transformation and mentions the corresponding rule. Therefore, it can be classified as rule-based trace.

Trace Generation The second feature differentiates an *automatic* from a *manual generation* of the trace. This feature can also be related with the type of the trace. While link-based traces are frequently built manually by the transformation developer in the transformation rules, rule-based traces are an artifact automatically created by the transformation execution engine. However, as discussed by Kahani et al. [Kah+19], ATL traces can be maintained and accessed by the transformation developer. Depending on the ATL rule type, the developer may create custom trace information in case no automatic generation is supported. However, in any case, the default virtual machine does not persist the ATL traces after the execution.

Trace Completeness The third criterion, *completeness*, refers to the elements that are stored per rule application. *Incomplete* traces only store a single target element and a single source

Table 5.1: Categorization of an exemplary set of M2M transformation traces.

Tool	Trace Data Structure	Type	Generation	Completeness	Granularity
medini QVT	trace model	rule-based	automatic	complete	coarse
ATL/EMFTVM	trace model	rule-based	automatic	generation-complete	coarse
BXtend	correspondence model	link-based	manual	incomplete	coarse
eMoflon	correspondence graph	link-based	manual	incomplete	coarse
eMoflon	protocol	rule-based	automatic	complete	fine

element regardless of the fact how many elements are created for the source(s) element(s) and how many context elements are required for their creation. In contrast, a *generation-complete* trace stores all target elements created by the corresponding rule application but no context elements. The latter are persisted by *complete* traces in addition to the created target elements. Fig. 5.2.2 exemplifies the different kinds of traces for the transformation scenario of converting a UML class into a Java class declaration and compilation unit. The top row shows the excerpt of an *incomplete* trace: Only the UML class and the Java class declaration are referenced by the trace element although the compilation unit is created as well by the same rule application corresponding to the trace element. The excerpt in the second row demonstrates a *generation-complete* trace element referencing not only the class but also the compilation unit. However, the Java model and package, which are necessary to store the class declaration and the compilation unit, are referenced only in the trace elements of *complete* traces as depicted at the bottom of the figure. In addition, depending on the granularity of the trace, not only the objects are stored in the trace but also the links between them or even the attributes of objects. In the figure we show a fine-grained complete trace where also the links between objects are referenced as source, target and context elements by the trace element.

The classification of the completeness of traces is in accordance with the *scope of transformation steps* proposed by Cuadrado et al. [CM09] for the modularization of a model transformation. The authors distinguish a *local to local*, *local to global*, *global to local* and *global to global* scope of performing transformations steps. The local to local scope corresponds with an incomplete trace, where only one target element can be created from one source element, which is called *pivot element* by the authors. In a local to global transformation step more than one target element is created which may require to collect information from different parts of the target model (i.e., for instance context elements). In contrast, a global to local transformation step requires more (globally available) information from the source model than the pivot element to create a single target element. We represent that case by incorporating more than one source element in generation-complete and complete traces. Finally, a global to global transformation step combines the cases of a global to local and a local to global transformation steps and is represented in a complete trace.

Notice that the type and the completeness are orthogonal criteria. Completeness refers to the amount of information stored in the trace whereas the type describes the fact how the trace is created.

Trace Granularity Lastly, traces may either be *coarse-grained* or *fine-grained*. A coarse-grained trace records elements of the target model at the level of objects only. Conversely, a fine-grained trace also links the attributes of and references between objects to corresponding source elements. The ATL/EMFTVM, medini-QVT and BXtend traces as well as the correspondence graph of eMoflon represent coarse-grained information at the level of objects. In contrast, the protocol of eMoflon maintains a more fine-grained trace, which stores the references between objects but neither their attributes.

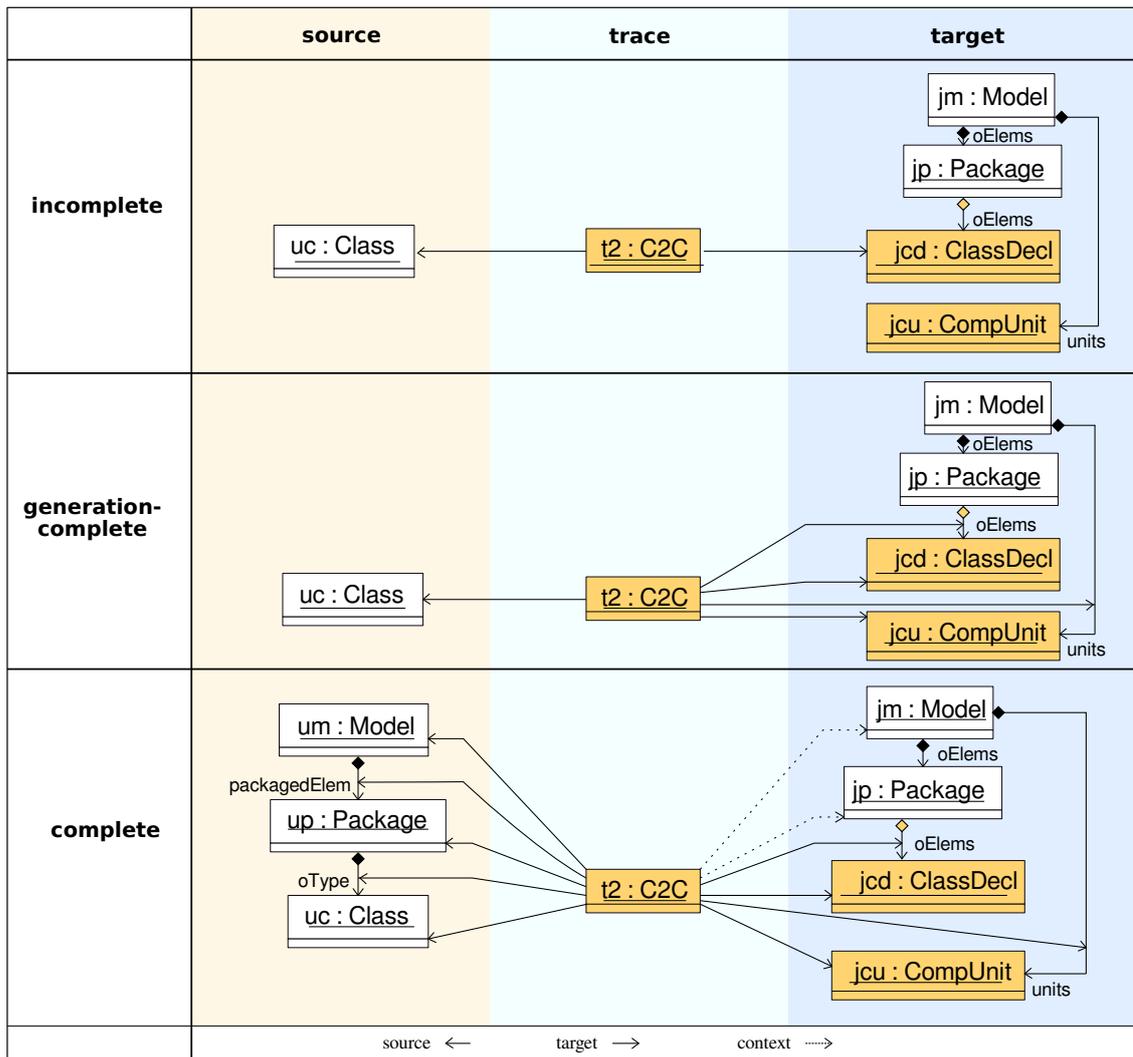


Figure 5.2.2: Trace completeness levels.

5.2.3 Common Trace Metamodel for Annotation Propagation

Based on the criteria classifying the different transformation traces, we derive a common trace metamodel into which transformation traces can be turned. Fig. 5.2.3 depicts the generalized model. The trace model encompasses an ordered set of trace elements. The order results from the sequence of applying transformation rules.

Each trace element references up to many source and target as well as context elements. Despite the fact that the trace elements enumerate sets of source, target and context elements (i.e., representing a complete trace), incomplete traces which reference only one source and one target element are covered as special instances of this metamodel: The set of context elements is completely empty and the sets of source and target elements include one respective pivot element.

5.3 Trace-Based Annotation Propagation

This section presents the key characteristics of the trace-based propagation of annotation at an informal level. Sec. 5.3.1 initiates with giving a schematic overview of the propagation whereas Sec. 5.3.2 explains how to process a trace model and to compute the target annotations based on this information. Finally, Sec. 5.3.3 enumerates the properties to which the transformation type, the transformation rules and the recorded trace must conform in order to satisfy commutativity.

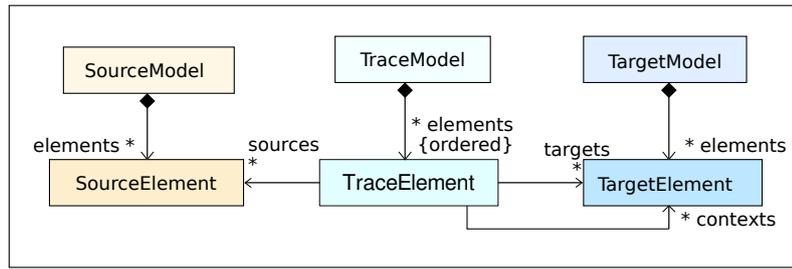


Figure 5.2.3: Common trace metamodel.

5.3.1 Schematic Overview

The knowledge about different trace kinds and the common trace metamodel lay the grounds for propagating the annotations of multi-variant models to the target model without having to adapt the reused single-variant transformation.

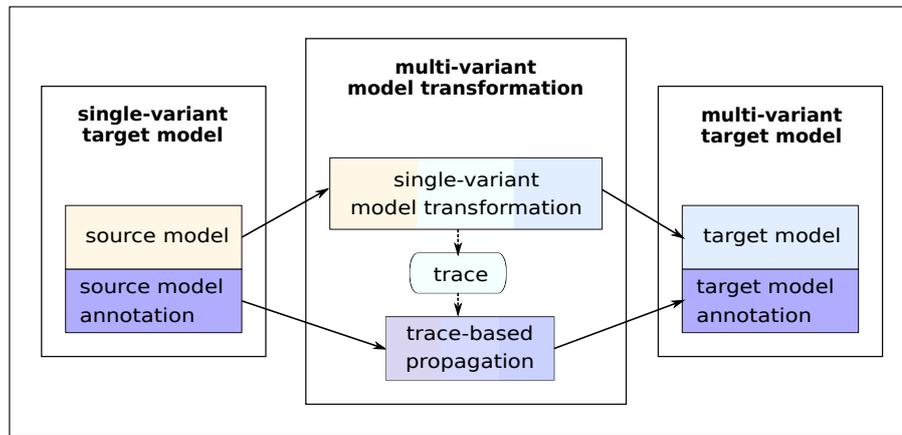


Figure 5.3.1: Schema of trace-based annotation propagation.

The schematic overview in Fig. 5.3.1 illustrates the concept of utilizing the trace for propagating annotations. The single-variant model transformation first turns the multi-variant source model into a multi-variant target model (without regarding annotations). As an artifact of the transformation, a trace is generated and serves as input to the trace-based propagation algorithm transferring the annotations from the source model to the target model. Details of the propagation algorithm are given next.

5.3.2 Annotation Propagation Procedure

For propagating annotations to the target model, the trace elements `elements` of the common trace metamodel (Fig. 5.2.3) are processed sequentially by their order. For the algorithm it is important that the order corresponds with the application of transformation rules and, thus, the order of creating elements in the target model.

According to the common trace model, each trace element enumerates a set of source, target and context elements, which are abbreviated as `SRC`, `TRG` and `CTX`, respectively, in the following explanations. For each trace element, the annotations of the source elements in `SRC` are queried as well as the annotations of the context elements in `CTX`. These annotations are combined in a conjunction and attached to each target element `trg` \in `TRG` in the following way:

$$\text{trg.ann} := \left(\bigwedge_{s \in \text{SRC}} \text{s.ann} \right) \wedge \left(\bigwedge_{c \in \text{CTX}} \text{c.ann} \right)$$

Please note: For demonstration reasons the formula notes the annotation of the source and target

elements as one of their attributes. As explained in Sec. 4.1, however, mapping the annotations onto model elements may be realized in varying ways in reality. Moreover, the expression has to be simplified before attaching it to the target elements to avoid redundant clauses and transitively growing expressions.

In the transformation example of the beginning of this chapter (Fig. 5.1.4), the annotations are propagated by the trace-based propagation as follows. The first trace element corresponds with the creation of models. Accordingly, the only referenced target element `jm` receives the annotation of the recorded source element `um` which is `DB`.

Next, the trace element `t2` mapping the packages is processed. Here, the Java model `jm` is referenced as context element and `up` as the single source element. Accordingly, the annotation attached to the package would be $DB \wedge DB$ which can be simplified to the annotation `DB` for the Java package `jp`.

Thirdly, the elements `t3`, `t4` and `t5` are processed subsequently. As an example, the annotation computed based on `t4` for the class declaration, compilation unit and the containment references originating from the model and package receive the annotation $(DB \wedge P) \wedge DB \wedge DB$ where the first clause stems from the source element and the second and third from the context elements. Only the simplified expression $DB \wedge P$ is applied to the target elements.

5.3.3 Computational Model

To propagate the annotations to target elements based on the informally described algorithm correctly, the transformation and their traces have to adhere to certain properties. The *computational model* for trace-based annotation propagation summarizes these properties. If the computational model is satisfied, commutativity can be guaranteed as explained in Chp. 6. The following paragraphs introduce each property at a conceptual level starting with general properties for the transformation and continuing with concrete properties for the rules and the resulting traces.

Transformation Properties First of all, the transformation, in general, has to conform to the following properties.

Property 5.3.1: Batch Transformation

A batch transformation has to be performed, thus, creating a new target model.

The computational model assumes that no target model exists yet. Accordingly, a new target model without any annotation is created. Since incremental transformations are not regarded in the computational model, Sec. 7.5 presents an extension how to address them.

Property 5.3.2: Rule-Based Traces

The transformation has to be composed of rules. The rule-based transformation execution applies each rule to each match exactly once and the trace records the rule generating the trace element.

Traces have to record rule applications in an ordered way. As a consequence, the computational model requires a rule-based transformation which apply all rules to all matches exactly once.

Notice: Link-based traces are subsumed in rule-based traces as long as each link corresponds exactly with one rule application. Furthermore, this property does not cover explicit *control structures* which prescribe the order of applying rules. Instead, the model assumes that in a global control structure the transformation execution environment resolves dependencies between target elements and applies them in the correct order. The order of the trace reflects the order in which the execution environment creates elements in the target model.

Property 5.3.3: Out-Place Transformation

The transformation has to be performed out-place, which means it creates a target model at a physically different location from the source model.

Prop. 5.3.3 prescribes that the source and target model are physically separated. As a further implication, it assumes that the source model remains the same and is not modified by the forward transformation. Consequently, all source model elements (including their annotations) are pertained as they are.

Rule Application Properties Besides the transformation in general, the application of transformation rules has to satisfy the following properties, in particular.

Property 5.3.4: Functional Rule Application

Rules have to be functional: Applying a rule to a given match determines the result uniquely.

Commonly, rule-based transformation languages are of functional behavior. In the computational model, the functional behavior of rules carries over to the entire transformation. Non-functional single-variant model transformations would result in different output model in different executions. Therefore, non-functional transformations could prevent the multi-variant model transformation to commute with respect to the filters and single-variant transformations.

Property 5.3.5: Monotonic Rule Application

Rules have to be monotonic: Applying a rule adds elements to the target model but does not remove or change any already existing element in the target model.

Due to the batch behavior, deletions and modifications of target elements are not supported by the computational model which also holds for single rules. The common trace metamodel (Sec. 5.2.3) groups elements of the target model into context and (created) target elements but not in modified or deleted ones.

Property 5.3.6: Local Rule Application

Rules have to be local: The effect of applying a rule depends on the match only.

Locality is essential to support commutativity because it guarantees that the rule is applicable to the same match in the unfiltered (multi-variant) as well as in filtered (single-variant) models. For that reason, the context of matching the rule in the source model has to be irrelevant to the rule. If rules were not local and, thus, the context of the match were relevant, an application condition could cause a rule being applicable in the multi-variant model but not in a filtered model or vice versa.

Trace Properties Finally, the trace for the transformation has to be complete and fine-grained:

Property 5.3.7: Complete Traces

A complete trace, which enumerates the subsequently executed rule, all source, context, and target elements, has to be recorded.

If target elements are missing from the trace, they cannot receive variability annotations by the

propagation algorithm. Consequently, the target elements missing in the trace, miss an annotation. Then, assumptions or conventions have to be made whether these elements are visible in every variant of the product line. Missing source and context elements in the trace may also cause annotations that are not specific enough and may result in a violation of commutativity.

Property 5.3.8: Fine-Grained Traces

The trace has to be as fine-grained as the mapping mechanism and the model filter.

Although in the example (Fig. 5.1.4) we illustrate annotations at the level of objects, the computational model postulates the usage of fine-grained traces. Only if the annotations of all involved model elements (i.e., of objects, attributes and links) are considered, the annotation for the target elements is computed correctly.

Please note: To guarantee commutativity, the trace needs to be as fine-grained as the mapping and the filter. While the granularity of the mapping mechanism determines up to which level annotations can be assigned to source elements (e.g., to their structural features), the granularity of the filter determines up to which granularity these annotations affect the derived variant. If the filter works at the level of objects only, it cannot remove structural features of the objects. In that case, a trace at the level of objects only will suffice to satisfy commutativity.

5.4 Summary

This chapter illustrates a single-variant UML class to Java model transformation and explains how to lift it to propagate annotations. We exemplify how a trace-generating rule behaves and employ this information together with an examination of traces persisted by different transformation engines, to derive the concepts for propagating annotations from the multi-variant source to the multi-variant target model. Based on a derived common trace model, annotations from recorded source and context elements are mapped onto recorded target elements. Finally, at a conceptual level Sec. 5.3.3 explains which properties the transformation specification and execution engine need to satisfy in order to guarantee commutativity.

Chapter 6 Formal Foundations

*A pack of wolves, a bunch of grapes,
or a flock of pigeons are all examples of sets of things.*

Paul Halmos

Given the informal explanations of trace-based propagation in Chp. 5, this chapter presents how to formally note multi-variant model-to-model transformations and defines the important properties to guarantee commuting multi-variant and single-variant transformations. The main purpose of this chapter is to demonstrate which conditions have to be satisfied to prove commutativity of multi-variant model transformations. Instead of repeating the complete proof, we refer the interested reader to the corresponding publications [WG18; WG20a] for details and present the background knowledge and key ideas of the proof here.

For defining multi-variant M2M transformations, this chapter employs a formalism based on Graph theory. This formalism considers models as *graphs* consisting of *nodes* and *edges*. Model transformations are *Graph transformations* where a left-hand side and right-hand side graph constitute the rule which replaces matches of the left-hand side in the host graph by the right-hand side.

By employing the informal example of the preceding chapter, at first, Sec. 6.1 and Sec. 6.2 demonstrate the graph formalism for expressing *single-variant models* and *multi-variant models*, respectively. Next, Sec. 6.3 introduces trace-generating in- and out-place graph transformations on single-variant models, denoted as *derivations* and *STT derivations*. Last but not least, Sec. 6.4 contributes the algorithm which employs the generated trace to propagate annotations to the target model in multi-variant model transformations and sketches the proof of commutativity based on the properties of the computational model for trace-based propagation. Sec. 6.5 closes the chapter by summarizing its content.

[WG20a], [WG20b] and [WG18] lay the foundations for this chapter.

6.1 Models as Graphs

The formal notion of multi-variant transformation represents models as Graphs which consist of nodes and edges. Accordingly, we employ *Graph theory* [Ehr+15] to formalize transformations. The foundations of Graph theory lie in set theory [Hal17] which we utilize throughout the formalization.

Before defining model transformations, this section starts by noting models in Graph formalism (Sec. 6.1.1) and defining relationships between graphs in form of morphism operators (Sec. 6.1.2).

6.1.1 Single-Variant Models

In this thesis, models form spanning containment trees and their objects are linked by typed references. As a consequence, it is possible to express the model as a graph which requires the following mapping:

Graph Each object is represented by a node in the graph and each link between objects by an edge. Therefore, a *graph* encompasses *nodes* and directed *edges* as typed *elements*. We refrain from explicitly including the attributes of objects, such as the name or their type, in the graph. Consequently, we employ a *typed graph*, where nodes and edges are labeled but without attributes nor an explicit graph schema prescribing details of the structure of the graph.

Please note: the following definition of a graph permits – besides a unique source and target node – an edge as target of an edge, which is also denoted as *higher order edge*. Higher order edges will be used for defining traces where the target of an edge may be an edge of the source or target graph.

The set of nodes N and edges E of a graph are summarized as the disjoint set of typed graph elements $EL = N \dot{\cup} E$ and used for defining a typed graph in Def. 6.1.1:

Definition 6.1.1: Graph

Let T_N and T_E be finite sets of node types and edge types, respectively. A graph over T_N and T_E is a tuple $G = (N, E, l_N, l_E, s, t)$, where

- N is a finite set of nodes,
- E is a finite set of edges, where $(N \cap E = \emptyset)$,
- $l_N : N \rightarrow T_N$ is a node labeling function,
- $l_E : E \rightarrow T_E$ is an edge labeling function,
- $s : E \rightarrow EL$ is a source function, and
- $t : E \rightarrow EL$ is a target function.

An edge e is *self-referential* if the source or the target of the edge is the edge itself, i.e., if $s(e) = e \vee t(e) = e$.

Ordered Graph More specifically, the models in our thesis are ordered implying that an *ordered graph* represents them. The nodes and edges in an ordered graph satisfy an ordering function as defined in Def. 6.1.2.

Definition 6.1.2: Ordered Graph

Let $G = (N, E, l_N, l_E, s, t)$ be a graph over T_N and T_E . G is ordered with respect to its edge set E if and only if an ordering function $ord : EL \rightarrow \mathbb{N}_0^+$ which maps graph elements onto natural numbers exists and which satisfy the following conditions:

$$\forall n \in N : ord(n) = 0 \tag{6.1}$$

$$\forall e \in E : ord(e) = \max(ord(s(e)), ord(t(e))) + 1 \tag{6.2}$$

As a consequence, the ordering function guarantees the following property:

$$\forall e \in E : ord(e) > ord(s(e)) \wedge ord(e) > ord(t(e)) \quad (6.3)$$

Consequently, an ordered graph prohibits the existence of self-referential edges. The following descriptions assume ordered graphs.

Ex. 6.1.1 demonstrates how the UML class model introduced in Fig. 5.1.1 is expressed in graph notation.

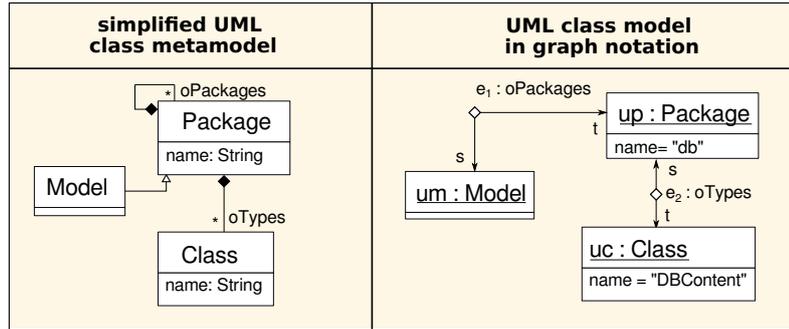


Figure 6.1.1: UML class model in graph notation.

Example 6.1.1: UML Class Model as Graph

The UML class model depicted on the right-hand side of Fig. 6.1.1 serves as an example throughout this chapter. It is represented as a graph $G = (N, E, l_N, l_E, s, t)$ over T_N and T_E and constitutes in the following way:

- $T_N = \{Model, Package, Class\}$
- $T_E = \{oPackages, oTypes\}$
- $N = \{um, up, uc\}$
- $E = \{e_1, e_2\}$
- $l_N = \{um \mapsto Model, up \mapsto Package, uc \mapsto Class\}$
- $l_E = \{e_1 \mapsto oPackages, e_2 \mapsto oTypes\}$
- $s = \{e_1 \mapsto um, e_2 \mapsto up\}$
- $t = \{e_1 \mapsto up, e_2 \mapsto uc\}$

G is ordered because the source and target of an edge are nodes only. Accordingly, the ordering function returns 0 for the nodes and 1 for the edges e_1 and e_2 , thus, satisfying Def. 6.1.2.

All of the following examples refer to the example of Fig. 5.1.1 in graph notation as exemplified for the UML class model in Fig. 6.1.1.

6.1.2 Graph Morphisms

In model transformations, the application of a rule is determined based on a *match* of input model elements with the source elements declared by the rule. Similarly, in graph transformations matches are determined based on the structure of the input graph and the one specified in the rule.

Graph morphisms define relations between graphs, particularly the equivalence of graphs or of sub-graphs, and are type and structure preserving mapping functions for graph elements. In the sequel, we refer to graph morphisms as *morphisms*:

Definition 6.1.3: Morphism

Let $G_1 = (N_1, E_1, l_{N_1}, l_{E_1}, s_1, t_1)$ and $G_2 = (N_2, E_2, l_{N_2}, l_{E_2}, s_2, t_2)$ be graphs over T_N and T_E with the disjoint sets of graph elements $EL_1 = N_1 \dot{\cup} E_1$ and $EL_2 = N_2 \dot{\cup} E_2$, respectively. A morphism $m : G_1 \rightarrow G_2$, $m = (m_N, m_E)$ comprises two functions: a node mapping function $m_N : N_1 \rightarrow N_2$ and an edge mapping function $m_E : E_1 \rightarrow E_2$ retaining node and edge types as well as the source and target of an edge. Furthermore, $m_{EL} : EL_1 \rightarrow EL_2$ denotes the element mapping function $m_{EL} = m_N + m_E$ applying either m_N or m_E depending on the kind of input. The morphism guarantees the following conditions:

$$\forall n_1 \in N_1 : l_{N_2}(m_N(n_1)) = l_{N_1}(n_1) \quad (6.4)$$

$$\forall e_1 \in E_1 : l_{E_2}(m_E(e_1)) = l_{E_1}(e_1) \quad (6.5)$$

$$\forall e_1 \in E_1 : s_2(m_E(e_1)) = m_{EL}(s_1(e_1)) \quad (6.6)$$

$$\forall e_1 \in E_1 : t_2(m_E(e_1)) = m_{EL}(t_1(e_1)) \quad (6.7)$$

An identity morphism is a morphism $id = (id_N, id_E)$ with identities on nodes and edges.

If both, m_N and m_E , are *injective*, *surjective*, or *bijective* the morphism is called *monomorphism*, *epimorphism*, or *isomorphism*, respectively. These definitions carry over to graphs which adhere to the following properties:

Definition 6.1.4: Graph Relations

Let $G_1 = (N_1, E_1, l_{N_1}, l_{E_1}, s_1, t_1)$ and $G_2 = (N_2, E_2, l_{N_2}, l_{E_2}, s_2, t_2)$ be graphs over T_N and T_E . We employ the following graph relations:

- *sub-graph*: $G_1 \subseteq G_2$ if and only if $id = (id_N, id_E)$ is an identity morphism from G_1 to G_2 .
- *less or equal*: $G_1 \lesssim G_2$ if $m : G_1 \rightarrow G_2$ is a monomorphism.
- *isomorphic*: $G_1 \simeq G_2$ if $m : G_1 \rightarrow G_2$ is an isomorphism.

Furthermore, *morphism operators* construct a new graph from existing morphisms. The following explanations consider the *composition*, *inverse*, *restriction* and *range* of up to two morphisms incorporating up to three graphs.

Definition 6.1.5: Morphism Operators

Let $G_i = (N_i, E_i, l_{N_i}, l_{E_i}, s_i, t_i)$ ($1 \leq i \leq 3$) be graphs over T_N and T_E .

- *Composition*: Let $m_1 : G_1 \rightarrow G_2$ and $m_2 : G_2 \rightarrow G_3$ be morphisms, where $m_1 = (m_{N_1}, m_{E_1})$ and $m_2 = (m_{N_2}, m_{E_2})$. The morphism $m = m_1 \circ m_2 : G_1 \rightarrow G_3$ with $m = (m_{N_1} \circ m_{N_2}, m_{E_1} \circ m_{E_2})$ is called the composition of m_1 and m_2 .
- *Inverse Morphism*: Let $m : G_1 \rightarrow G_2$, with $m = (m_N, m_E)$ be an isomorphism. The isomorphism $m^{-1} : G_2 \rightarrow G_1$ with $m^{-1} = (m_N^{-1}, m_E^{-1})$ is called the inverse of m .
- *Restriction*: Let $m : G_1 \rightarrow G_2$, with $m = (m_N, m_E)$ be a morphism, and let $G_3 \subseteq G_1$ be a subgraph of G_1 . The morphism $m|_{G_3} : G_3 \rightarrow G_2$ such that $m|_{G_3} = (m_N|_{N_3}, m_E|_{E_3})$ is called the restriction of m onto G_3 which only maps the elements of the subgraph G_3 onto elements of G_2 .
- *Range*: Let $m : G_1 \rightarrow G_2$ with $m = (m_N, m_E)$ be a morphism. The range of m , $ran(m)$ is a subgraph $G_3 \subseteq G_2$ with node set N_3 and edge set E_3 such that $N_3 = ran(m_N)$ and $E_3 = ran(m_E)$.

The following two properties of morphisms, which follow from their definitions, are essential for ensuring commutativity:

- The composition m_3 of two monomorphisms m_1 and m_2 , such that $m_3 = m_1 \circ m_2$, is a monomorphism, too.
- For a morphism $m : G_1 \rightarrow G_2$ the morphism $m' : G_1 \rightarrow \text{ran}(m)$ is an epimorphism iff

$$\forall n \in N_1, \forall e \in E_1 : m'_E(e) = m_E(e) \wedge m'_N(n) = m_N(n).$$

6.2 Variability in Graphs

As the graphs defined so far represent single-variant models, this section introduces the extension of graphs to represent the dimension of spatial variability present in SPLE. In this way, it starts with defining annotations and their effect in filtering models and closes with formalizing multi-variant graphs and model filters.

Feature Models In this thesis we employ feature models, as one kind of variability model, to express variability in a product line. Boolean *features* represent the common and distinguishing parts of the product line. The finite set of *features* $F = \{f_1, \dots, f_n\}$, ($n \geq 1$) represents them. For the proof of commutativity and for representing multi-variant model transformations we refrain from considering constraints among the features. The correctness of the feature model and the validity of corresponding feature configurations is supposed to be analyzed by specific SPLE tool support.

Annotations For annotating models, as mentioned in the previous chapters, typically not a sole feature is used but a Boolean expression over features to which we refer as *annotation*:

Definition 6.2.1: Annotation

Let F be a set of features. An annotation over F is an arbitrary Boolean expression in propositional logic over features from F . The set of all Boolean expressions over F is denoted by A_F ; we write $a_F \in A_F$ for one annotation.

A *feature configuration* describes a product variant by determining the features to be incorporated in this variant. In contrast to an annotation, in a (complete) feature configuration every feature of the set of features of the product line must be assigned a selection state which mentions the feature either positively or in a negated form.

Definition 6.2.2: Feature Configuration

Let $F = \{f_1, \dots, f_n\}$, $n \geq 1$, be a set of features. A feature configuration over F is a conjunction of bindings $fc_F = b_1 \wedge \dots \wedge b_n$, where $b_i \in \{f_i, \neg f_i\}$. The set FC_F denotes the set of all valid feature configurations.

Negatively bound features imply that they are not represented in the derived variant whereas positively bound ones are included. We do not consider *partial* configurations in this thesis. For that reason, a feature configuration always provides either a positive or negative selection state to each feature in the feature model (i.e., it is *fully bound*).

Furthermore, the constraints and dependencies in a feature model declare which feature configurations are *valid*. We refrain from formalizing valid feature configurations but assume in the following definitions that only valid feature configurations are included in the set of feature configurations.

Visibility Evaluation Annotative approaches require to map annotations onto model elements to encapsulate the knowledge which elements to remove in the product derivation process. For deriving products we employ a *visibility evaluation function* which expresses whether a given annotation satisfies a given feature configuration.

Definition 6.2.3: Visibility Evaluation Function

Let F be a set of features, FC_F the set of feature configurations and A_F be the set of annotations over F . A visibility evaluation function over F is a function $v_F : A_F \times FC_F \rightarrow \mathbb{B}$, where $\mathbb{B} = \{true, false\}$, denotes the set of Boolean values. v_F guarantees the following property:

$$v_F(a_F, fc_F) = true \Leftrightarrow fc_F \Rightarrow a_F \quad (6.8)$$

Example 6.2.1: Features, Annotations and Configuration

As an example, let assume $F = \{f_1, f_2, f_3\}$ is a set of the features representing the features *DB*, *P* and *Fa* of the database example, respectively. Given an annotation $a_F = f_1 \wedge \neg f_2$ and a configuration $fc_{F,1} = f_1 \wedge f_2 \wedge f_3$, the visibility evaluation function $v_F(a_F, fc_{F,1})$ returns *false* whereas it returns *true* for the same annotation and the feature configuration $fc_{F,2} = f_1 \wedge \neg f_2 \wedge f_3$.

Multi-Variant Model To represent multi-variant models, we employ *multi-variant graphs*. The elements of the multi-variant graph are associated with annotations by a mapping function which together with the graph defines the multi-variant graph:

Definition 6.2.4: Multi-Variant Graph

Let F be a set of features. A multi-variant graph is a pair $MG_F = (G, map_F)$, where $G = (N, E, l_N, l_E, s, t)$ is a graph, $EL = N \cup E$, and $map_F : EL \rightarrow A_F$ is a mapping annotation function assigning an annotation to each element of G .

map_F must ensure referential integrity, thus, the following constraint must be satisfied for each edge in E :

$$\forall e \in E : map_F(e) \Rightarrow map_F(s(e)) \wedge map_F(t(e)) \quad (6.9)$$

The set of all multi-variant graphs for the set of features F is denoted as \mathcal{MG}_F .

Referential integrity assures that a single-variant graph derived from the multi-variant graph is well-formed with respect to an absence of dangling edges. Furthermore, depending on the properties of the mapping function, either the graph is annotated completely, i.e., the function can determine an annotation for each graph element in EL , or not. In this chapter, the function is assumed to be total for the source graph of the transformation, thus, each element is assigned an annotation.

Example 6.2.2: Multi-Variant Graph

Let $F = \{f_1, f_2, f_3\}$ denote the set of features in this example. The multi-variant graph $MG = (G, map_F)$ consists of the single-variant graph and the mapping function. Fig. 6.2.1 illustrates the single-variant graph G , which comprises a model, a package and three classes together forming the five nodes of the graph. The model, the package and one class carry the annotation *DB* (i.e., f_1), the second class (node *ucp*) the feature *DB* \wedge *P* (i.e., $f_1 \wedge f_2$) and the third class (node *ucf*) the annotation *DB* \wedge *Fa* (i.e., $f_1 \wedge f_3$). Accordingly, the mapping function map_F returns these annotations for the respective input nodes.

The edges of this example satisfy referential integrity, thus, conforming to the definition of multi-variant graphs. As an example, the nodes connected by e_3 will be visible besides the edge itself if the annotation of e_3 , $f_1 \wedge f_2$ (i.e., $DB \wedge P$), is implied by the feature configuration. The same statement holds for the edges e_1 , e_2 and e_4 .

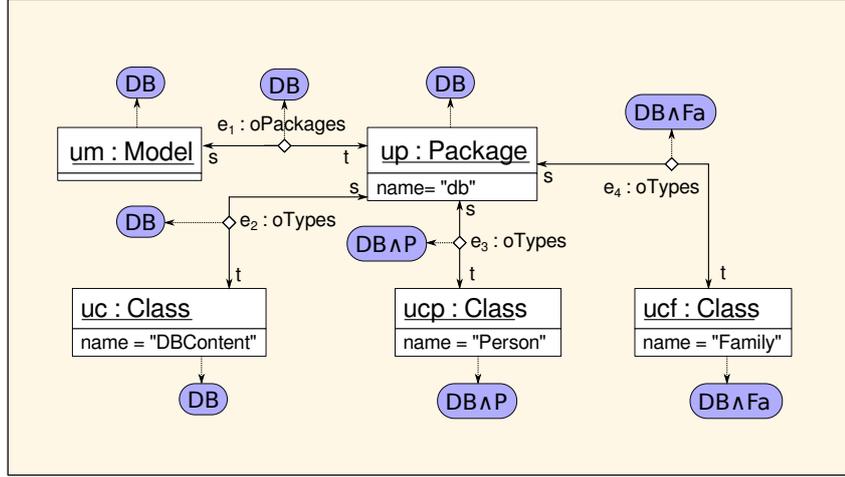


Figure 6.2.1: Multi-variant model in graph notation.

Model Filter For deriving products, in annotative approaches elements of the multi-variant model have to be removed to form the customized product. In the graph formalism a filter function accomplishes this task:

Definition 6.2.5: (Flat)Filter

Let F be a set of features and FC_F be the corresponding set of feature configurations. Let \mathcal{MG}_F and \mathcal{G} denote the sets of all multi-variant graphs and single-variant graphs, respectively. A filter is a function $filter_F : \mathcal{MG}_F \times FC_F \rightarrow \mathcal{G}$ defined as follows: For a given multi-variant graph $MG_F = (G, map_F)$ and a feature configuration fc_F , the function $filter_F(MG_F, fc_F) = G'$, with $G' \subseteq G$, adheres to the following properties:

$$N' = \{n \in N \mid v_F(map_F(n), fc_F) = true\} \quad (6.10)$$

$$E' = \{e \in E \mid v_F(map_F(e), fc_F) = true\} \quad (6.11)$$

Example 6.2.3: Filtered Graph Variants

As an example of filtered graph variants, Fig. 6.2.2 demonstrates three variants of the multi-variant graph presented in Ex. 6.2.2. We assume that the feature DB is mandatory whereas P and Fa are optional yielding four valid feature configurations.

The top row of the figure declares the feature configuration and presents the corresponding graph variants derived by the filter functions below. On the left, Fig. 6.2.2 presents the variant encompassing the class $DBContent$ and its containers only. Since the features Fa and P are deselected in the corresponding configuration all other elements are removed from the multi-variant graph by the filter function. Accordingly, the second and third column comprise variants where P and Fa are each solely selected, respectively. The fourth feature configuration where P and Fa are selected simultaneously, incorporates all elements of the

multi-variant graph as depicted in Fig. 6.2.1.

6.3 Graph Transformations

This section introduces a rule-based formalism for transforming (single-variant) graphs and at the end multi-variant graphs. The section starts with the definition of in-place transformations applied to one graph (Sec. 6.3.1). After summarizing the properties of such transformation in Sec. 6.3.2, Sec. 6.3.3 extends the in-place to out-place transformations. An out-place transformation is simulated by splitting a single graph in three mutual exclusive sub-graphs representing a *trace graph* in between *source* and *target* graphs.

6.3.1 (In-Place) Rules and Derivations

For generating a target model transformation rules are executed. Here we start with defining in-place transformation rules and their application for deriving a target representation of the input graph. In the following the input graph G to which rules are applied is referred to as *host graph*. A transformation rule consists of a left-hand side graph and a right-hand side graph where the left-hand side is replaced by the right-hand side.

Definition 6.3.1: Rule

Let L and R be graphs over T_N and T_E , such that $L \subseteq R$. The pair $\rho = (L, R)$ represents a rule over T_N and T_E . L and R are called the left-hand side and the right-hand side graphs of the rule, respectively.

The finite set of all rules ρ is denoted as P .

To apply rules to the input graph, a *matching site* (*match*) needs to be present in the input graph. Accordingly, the elements contained in L have to match the structure of the parts of the input graph exactly. Therefore, an injective mapping (i.e., a monomorphism) represents a *match*:

Definition 6.3.2: Graph Match

Let G and $\rho = (L, R)$ be a graph and a rule over T_N and T_E , respectively. A *match* for ρ in G is a monomorphism $m_L : L \rightarrow G$.

The application of a rule to a match is called a *derivation step* because applying the rule to the match derives the right side. Consequently, we call the application of multiple rules (i.e., a

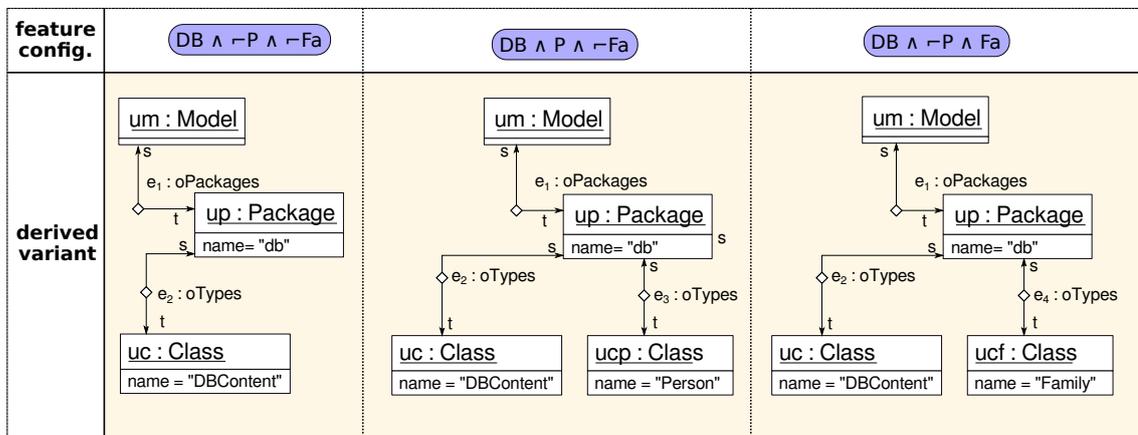


Figure 6.2.2: Filtered variants in graph notation.

transformation) a *derivation*. In short, we write the derivation step, involving the rule ρ and the match m as a pair (ρ, m) .

If there are two graphs G and H obtained by employing two monomorphisms m_L and m_R to the left-hand side and right-hand side, respectively, and two identity morphisms mapping L onto R , and G onto H , H can be *derived directly* from G by satisfying the following definition:

$$\begin{array}{ccc} L & \xrightarrow{id_L} & R \\ \downarrow m_L & & \downarrow m_R \\ G & \xrightarrow{id_G} & H \end{array}$$

Figure 6.3.1: Rule application diagram for match in G and direct derivation of H .

Definition 6.3.3: Direct Derivation

Let G , H and $\rho = (L, R)$ be two graphs and a rule over T_N and T_E , respectively. Let $m_L : L \rightarrow G$ be a match for ρ in G and let $id_L : L \rightarrow R$ and $id_G : G \rightarrow H$ denote identity morphisms.

H is directly derivable from G via ρ if H satisfies the following properties:

- A monomorphism $m_R : R \rightarrow H$ exists such that the diagram of Fig. 6.3.1 commutes: $id_L \circ m_R = m_L \circ id_G$.
- For each graph H' with the same properties, $H \lesssim H'$.

We note the derivation as $G \xrightarrow{\rho, m_L} H$ if the match m_L is of importance and $G \xrightarrow{\rho} H$ otherwise. Furthermore, applying a rule to a graph is denoted as derivation step.

Lastly, let P be a rule set. If a rule $\rho \in P$ exists, such that $G \xrightarrow{\rho} H$ holds, H is directly derivable from G via P ($G \xrightarrow{P} H$).

Accordingly, a derivation step (i.e, the application of a rule to a match) extends the host graph with elements of the right-hand side, which are not present in the left-hand side.

To conform to the computational model introduced in Sec. 5.3.3, the transformation rules satisfy the following properties:

1. ρ is *functional* (Prop. 5.3.4): after fixing the match m_L and applying ρ , the resulting graph H is unique up to isomorphism.
2. ρ is *monotonic* (Prop. 5.3.5): ρ adds nodes or edges to G but does not remove elements from G .
3. ρ is *local* (Prop. 5.3.6): the condition allowing to apply ρ as well as the effect of the application depend only on the match m_L .

Based on the application of single derivation steps, a *derivation*, transforming the input graph into a target graph, is defined in the following way:

Definition 6.3.4: Derivation

Let G , H and P be two graphs and a rule set over T_N and T_E , respectively. H is derivable from G via P ($G \xrightarrow{P^*} H$) if and only if a sequence of rules $\rho_0, \dots, \rho_{n-1}$, of matches m_0, \dots, m_{n-1} and of graphs G_0, \dots, G_n exists, such that $n \in \mathbb{N}_0^+$ and the following conditions hold:

- $G = G_0, H = G_n$

- $G_i \xrightarrow{\rho_i, m_i} G_{i+1}$ for $0 \leq i < n$
- A rule is applied only once to the same match:
 $(\rho_i, m_i) = (\rho_j, m_j) \Rightarrow i = j$ for $0 \leq i, j < n$

A sequence of derivation steps satisfying the conditions stated above is called a derivation.

A derivation $G \xrightarrow{P^*} H$ is complete if it cannot be extended any more. An empty derivation ($n = 0$) does not apply any rule.

A complete derivation is a *rule-based transformation*, thus, satisfying Prop. 5.3.2. Moreover, each rule is applied to a corresponding match exactly once and therefore, exposes the following properties.

6.3.2 Properties of Derivations

For guaranteeing commutativity not only the rules but also the derivations (i.e. the rule application steps) must satisfy the computational model. As stated above, rules have to exhibit functional, monotonic and local behavior. Except for locality, these properties carry over from single rule applications to the derivation. These properties are important to embed the single-variant transformations into the multi-variant transformation without generating contradicting results.

Functional Behavior Firstly, complete derivations have to exhibit *functional behavior*. Thus, the result of a complete derivation is *unique up to isomorphism*. Accordingly, if $G \xrightarrow{P^*} H_1$ and $G \xrightarrow{P^*} H_2$ are *complete* derivations H_1 and H_2 are unique up to isomorphism ($H_1 \simeq H_2$). Functional derivations guarantee the following property: if two rules r_1 and r_2 are applicable to the host graph at the same time, the order of applying r_1 and r_2 can be exchanged and delivers the same result, nonetheless. Accordingly, r_2 is applicable after having executed r_1 and vice versa resulting each time in the same graph.

This property (carrying over from rules (Prop. 5.3.4)) ensures that each time a transformation (derivation) is applied to the same host graph, the transformation results in the same derived target graph. This is necessary to safely embed single-variant into multi-variant transformations.

Monotonicity Secondly, derivations have to be *monotonic*. Consequently, the transformation only adds elements to the target graph but does not remove or modify already existing elements in the target graph which is in accordance with Prop. 5.3.5 for transformation rules.

As a result of this property, in a sequence of monotonic derivation steps via a rule set P , $G_0 \xrightarrow{P} G_1 \xrightarrow{P} \dots \xrightarrow{P} G_{n-1} \xrightarrow{P} G_n$, $n \in \mathbb{N}^+$, each derived “predecessor” graph is a sub-graph of the graph derived in the next application step, such that $\forall i \in [0 \dots n] : G_i \subseteq G_{i+1}$.

Furthermore, if H' is the result of a complete derivation via P ($G \xrightarrow{P^*} H'$) and H the graph resulting from a (partial) derivation ($G \xrightarrow{P} H$), H can be mapped onto H' injectively by a monomorphism, such that $H \lesssim H'$. This is the case, if there is a monomorphism $G \rightarrow G'$ and $G' \xrightarrow{P^*} H'$ because then the derivation $G' \rightarrow H'$ encompasses each of the applied derivation steps in $G \rightarrow H$. Due to locality, this holds for $G_0 \rightarrow G'_0$ and can be proven by induction for the remaining steps.

Termination In general, the definition of applying rules to matches (in in-place transformations) does not guarantee to terminate. Since it is possible that rules generate new matches during a derivation step, an *infinite sequence of derivation steps* may result. However, if a complete derivation starting from graph G' exists, any sequence of derivation steps starting on a graph $G \lesssim G'$ will terminate which is a consequence of above properties.

Consequently, if H' is the result of performing a complete derivation ($G' \xrightarrow{P^*} H'$) via a rule set P and H is the (intermediary) graph resulting from a partial derivation ($G \xrightarrow{P} H$) via P , the number

of derivation steps to create H is less than or equal as the number of derivation steps necessary to perform the complete derivation, yielding H' .

6.3.3 Out-Place Rules and Derivations

So far we have defined *in-place transformations*: Rules are applied based on matches in a single graph and modify exactly this graph. By decomposing a graph into three sub-graphs, a *source graph*, a *trace graph*, and a *target graph* and employing *trace-generating source-to-target rules*, we simulate forward *out-place transformations*. A *batch* transformation is defined as a *complete* derivation, initiated with a graph consisting of a source sub-graph (only) and empty trace and target sub-graphs.

To this end, this section adds all definitions of properties of the computational model for trace-based propagation (Sec. 5.3) which are still missing because they do not apply to in-place transformations.

STT Graph To begin with, a *source-to-target graph*, defined in Def. 6.3.5, records the information about the transformation:

Definition 6.3.5: Source-to-Target Graph

A source-to-target graph (**STT graph**) is a graph G typed over node types $T_N = T_{N_S} \cup T_{N_{TR}} \cup T_{N_T}$ and edge types $T_E = T_{E_S} \cup T_{E_{TR}} \cup T_{E_T}$, comprising three mutually exclusive sub-graphs $G = G_S \dot{\cup} G_{TR} \dot{\cup} G_T$ and mutually exclusive edge sets connecting these sub-graphs.

Accordingly, it is composed of the following elements:

- $G_S \subseteq G$: the source graph, typed over T_{N_S} and T_{E_S} .
- $G_T \subseteq G$: target graph, typed over T_{N_T} and T_{E_T} .
- $G_{TR} \subseteq G$: trace graph typed over $T_{N_{TR}}$ and $T_{E_{TR}}$ whereby
 - trace nodes are typed over rule identifiers, i.e., $T_{N_{TR}} = ID_P$ for some rule set P .
 - the edge type set $T_{E_{TR}}$ contains a single edge type: $T_{E_{TR}} = \{use\}$.
- Trace-to-source edges of type *src* from trace nodes N_{TR} to elements of the source graph.
- Trace-to-target edges of type *ctx* or *trg* from trace nodes N_{TR} to elements of the target graph.

We write $G = G_S \leftarrow G_{TR} \rightarrow G_T$ to indicate that G is an STT graph comprising the components as defined above.

According to Def. 6.3.5, the STT graph composes the source, the target and a trace graph. The nodes of the trace graph maintain links between corresponding source and target elements. Ex. 6.3.1 demonstrates how the STT graph represents the transformation rule which converts a UML package into a corresponding Java element.

Example 6.3.1: Source-To-Target Graph

Fig. 6.3.2 illustrates an example of an STT graph by showing the simplified UML class model and Java model from Fig. 5.1.1 in graph notation. The state of the graph is reached after applying the rule to generate the package after the model has been created.

Accordingly, two trace nodes exist in G_{TR} , τ_1 and τ_2 , which reference the corresponding

target elements. Most importantly, the edge between the model and the package is listed as a target element of τ_2 , too. Therefore, the trace is fine-grained. Furthermore, τ_2 enumerates three source elements. Besides the package which is the pivot element of this trace node, the reference `oPackages` and the model are referenced as *src* elements because both are required to correctly integrate the Java package in the Java model. The Java model is referenced as a *ctx* element of τ_2 because it was already created by the M2M rule but is necessary to add the package correctly. In addition, the dependency of the P2P rule of the M2M rule is marked by a use edge from τ_2 to τ_1 in the trace sub-graph.

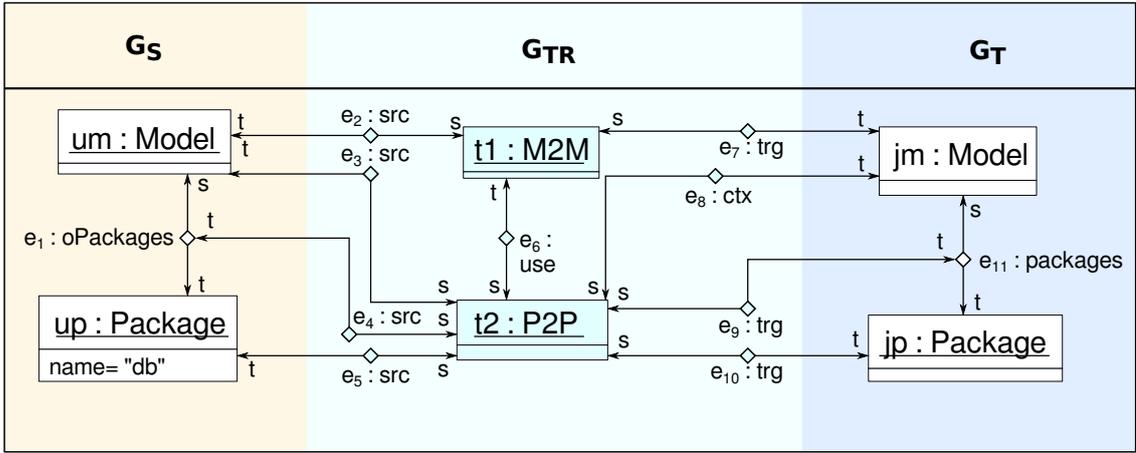


Figure 6.3.2: Source-to-target rule of P2P rule.

For easier readability, the following definitions refrain from mentioning node and edge types explicitly and from expanding graphs into components. Instead, per convention indices indicate to which sub-graph an element belongs. For instance, N_{TR} is the set of trace nodes present in the trace sub-graph G_{TR} .

STT Rule As a consequence of the definition of STT graphs, the definition of a rule is extended to address the sub-graphs. The left-hand side graph L and right-hand side graph R are both STT-graphs in the following explanations. To ensure monotonicity (Prop. 5.3.5), a rule only extends the target sub-graph while the source and trace sub-graphs remain unmodified. These considerations allow to define a Source-To-Target Rule in Def. 6.3.6.

Definition 6.3.6: Source-to-Target Rule

Let $\rho = (L, R)$ be a rule, where $L = L_S \leftarrow L_{TR} \rightarrow L_T$ and $R = R_S \leftarrow R_{TR} \rightarrow R_T$ are STT graphs. ρ is a source-to-target rule (STT rule) if the following conditions hold:

$$\forall el \in EL_R : el \notin EL_L \Rightarrow el \in EL_{R_T} \quad (6.12)$$

$$\forall el \in EL_{L_T} : |\{e \in E_L \mid t_L(e) = el \wedge l_{E_L}(e) = trg\}| = 1 \quad (6.13)$$

Accordingly, each element of the right-hand side graph R not contained in L is present in R_T only. Moreover, each element in L_T must possess *exactly one* incoming *trg* edge. The existence of the incoming *trg* edge for elements of the left-hand side's target graph (Equation 6.13) ensures the completeness of dependency information which will be required for the complete transformation of STT-Graphs(Def. 6.3.8). From Equation 6.12 follows that a rule does not add elements to the *source* and *trace* graph, such that the corresponding left-hand and right-hand side sub-graphs encompass exactly the same elements.

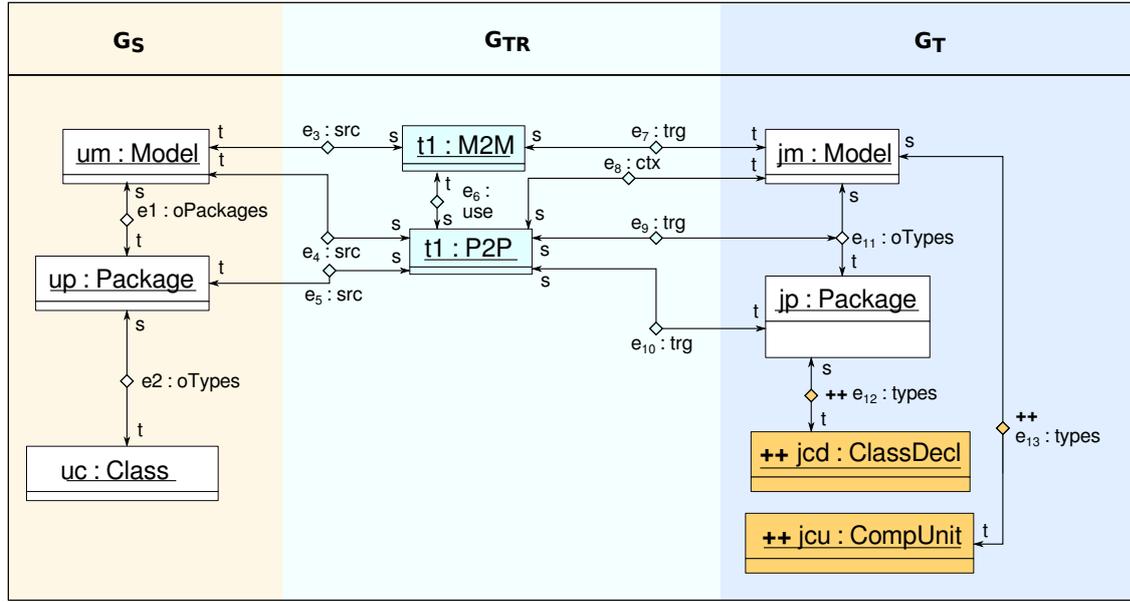


Figure 6.3.3: STT rule representing C2C rule.

Example 6.3.2: Source-To-Target Rule for Class Transformation

Fig. 6.3.3 demonstrates an STT rule creating a Java class and compilation unit for a UML class which corresponds to the informal rule of Fig. 5.1.3 except that no trace element is generated, yet. The ++ markers as well as the orange color indicate the elements which are added to the target graph. Accordingly, these elements are part of R_T only. All the other elements form part of L and R in the same way.

Trace-Generating Rule In order to capture trace information, STT rules defined by the user are turned into *trace-generating STT rules*, which are exploited by the transformation engine during the execution of the transformation. Model transformation solutions, for instance ATL/EMF-TVM and QVT-O, apply a similar mechanism. Thus, the user may define STT rules accessing trace information but is not required to specify the generation of trace information as is the case in link-based traces. Instead, as a side effect, the trace is created during execution automatically. The following definition prescribes how a (general) STT rule is extended to automatically add trace information to the right-hand side trace graph.

Definition 6.3.7: Trace-Generating STT Rule

Let $\rho = (L, R)$ be an STT rule, where $L = L_S \leftarrow L_{TR} \rightarrow L_T$ and $R = R_S \leftarrow R_{TR} \rightarrow R_T$ are STT graphs. Executing the following steps subsequently creates the trace-generating STT rule $\rho' = (L', R')$ where $L' = L'_S \leftarrow L'_{TR} \rightarrow L'_T$ and $R' = R'_S \leftarrow R'_{TR} \rightarrow R'_T$:

1. Initialize ρ' with ρ : $\rho' := \rho$.
2. Add a single trace node n' to the trace sub-graph R'_{TR} . n' is typed by the identifier id_ρ of ρ :

$$l'_N(n') = id_\rho \quad (6.14)$$

3. For each trace node $n \in L_{TR}$, create a trace edge e' of type *use* from n' to n :

$$l'_E(e') = use \wedge s'(e') = n' \wedge t'(e') = n \quad (6.15)$$

4. For each source element (node or edge) $el \in L_S$, create a trace edge e' of type *src* from n' to the source element el :

$$l'_E(e') = src \wedge s'(e') = n' \wedge t'(e') = el \quad (6.16)$$

5. For each already created target element $el \in L_T$, create a trace edge e' of type *ctx* from n' to the old target element el :

$$l'_E(e') = ctx \wedge s'(e') = n' \wedge t'(e') = el \quad (6.17)$$

6. For each new target element $el \in R_T$, create a trace edge e' of type *trg* from n' to the target element el :

$$l'_E(e') = trg \wedge s'(e') = n' \wedge t'(e') = el \quad (6.18)$$

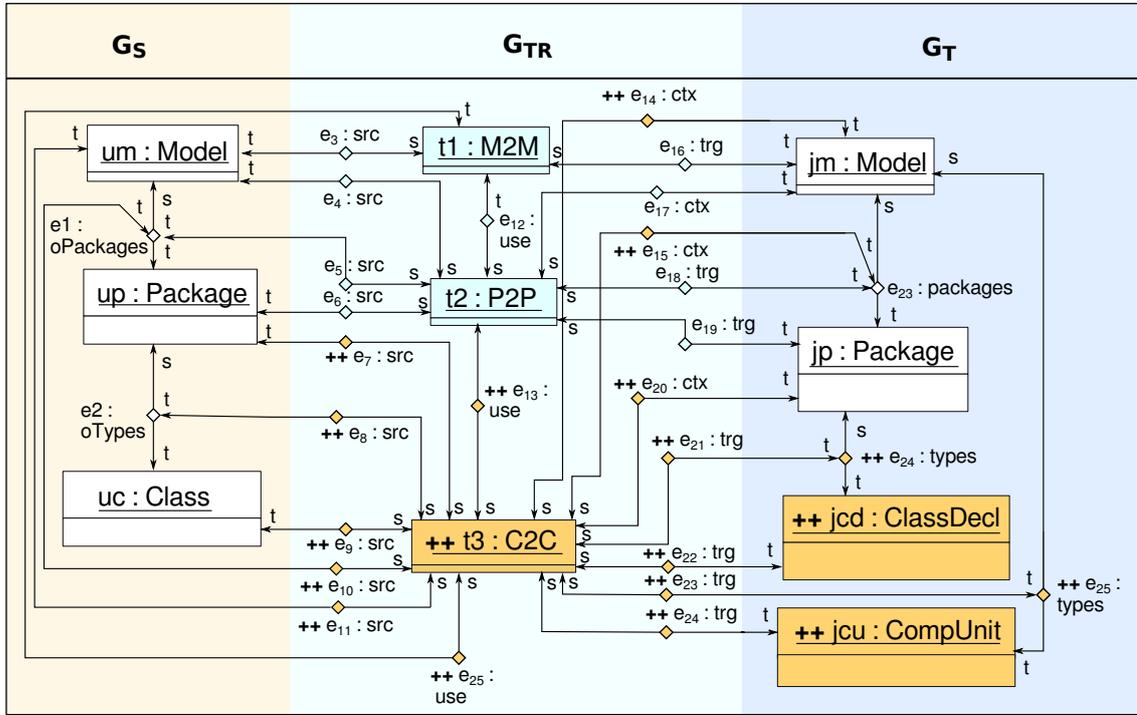


Figure 6.3.4: Trace-generating STT rule of C2C rule.

Example 6.3.3: Trace-Generating STT Rule

Fig. 6.3.4 depicts the trace-generating STT rule that is generated for the rule creating a Java class declaration and a compilation unit for a given class depicted in Fig. 6.3.3. Node $t3$ is added to G_{TR} according to the generation rules prescribed in Def. 6.3.7. Not only one source and target edge are added which reference the pivot elements, the UML class and Java class declaration, respectively, but also to all source elements and all created elements in the target graph. Moreover, use edges originating from $t3$ reference the trace nodes $t1$ and $t2$ and *ctx* elements reference the target elements, the *Model* and *Package* as well as their connecting references, which are necessary to add the created target elements in the right container.

Properties of Trace-Generating Source-to-Target Rules The construction of trace-generating STT rules allows to infer that a trace created by trace-generating STT rules conforms to the following properties of the conceptual model. The trace is

- **rule-based** (Prop. 5.3.2): Since each trace node represents the application of a rule by which the node is typed, the link to the rule is persisted.
- **complete** (Prop. 5.3.7): Trace elements record all source, context and target elements.
- **fine-grained** (Prop. 5.3.8): Besides nodes, trace elements also reference edges between the nodes and, thus, persist the finest level of granularity possible in the formalism.

For proving commutativity these properties are essential. Generating trace elements for each rule application covers all creations of target elements. Moreover, the completeness of the trace guarantees that the annotation information for its target elements can be computed correctly, as required by the definition of the annotation propagation algorithm (Sec. 6.3.3). Lastly, the ordering by *use* edges ensures the partial order necessary to process trace information in the correct transformation order.

Derivation via Trace-Generating Rules Derivations via trace-generating STT rules, called *STT derivations*, are derivations similar to those defined in Def. 6.3.4, which is extended in the following way.

Definition 6.3.8: STT Derivation

Let $G = G_S \leftarrow G_{TR} \rightarrow G_T$ and $H = H_S \leftarrow H_{TR} \rightarrow H_T$ be STT graphs. Furthermore, let P be a set of trace-generating STT rules. H is STT-derivable from G if $G \xrightarrow{P^*} H$ holds.

The notion of *completeness* carries over from derivations to STT derivations. In addition, since the STT rules do not modify the source graph, $G_S = H_S$ always holds.

Properties of STT Graphs The following facts about trace-generating STT rules are essential for propagating annotations based on the trace. In below statements the symbol \emptyset denotes an empty graph because the set of graph elements is empty. Furthermore, we employ binary relations to note edge types t_E : If an edge of type t_E from el_1 to el_2 exists, we will write the relation $(el_1, el_2) \in t_E$ for the typed edge or use arrow notation: $el_1 \xrightarrow{t_E} el_2$.

The following properties hold true for the STT-graph $H = H_S \leftarrow H_{TR} \rightarrow H_T$ derived via P from $G = G_S \leftarrow \emptyset \rightarrow \emptyset$, such that $G \xrightarrow{P^*} H$, :

1. Each target element of the target sub-graph H_T possesses exactly one incoming *trg* edge:

$$\forall el \in EL_{H_T} : |\{e \in E_H \mid t_H(e) = el \wedge l_{E_H}(e) = trg\}| = 1 \quad (6.19)$$

This property ensures that an annotation is assigned only once by the propagation algorithm presented in Sec. 6.4.

2. The trace sub-graph H_{TR} is acyclic with respect to *use* edges: Let $use^+ \subseteq N_{H_{TR}} \times N_{H_{TR}}$ denote the transitive closure over *use* edges. No trace node uses itself:

$$\forall n_{H_{TR}} \in N_{H_{TR}} : \neg(n_{H_{TR}} \xrightarrow{use^+} n_{H_{TR}}) \quad (6.20)$$

Due to this property, a trace node cannot be processed more often than once for propagation to the target elements when H_{TR} is iterated.

3. Each create/use dependency between rule applications is explicit: If a trace node $n_{H_{TR}}$ encompasses an outgoing ctx edge to a target element t , $n_{H_{TR}}$ will possess an outgoing use edge to another trace node referencing the target element t with an outgoing trg edge:

$$\begin{aligned} \forall n_{H_{TR}} \in N_{H_{TR}}, \forall el_{H_T} \in EL_{H_T} : \quad n_{H_{TR}} \xrightarrow{ctx} el_{H_T} \Rightarrow \\ \exists n'_{H_{TR}} \in N_{H_{TR}} \setminus \{n_{H_{TR}}\} : \quad n_{H_{TR}} \xrightarrow{use} n'_{H_{TR}} \wedge n'_{H_{TR}} \xrightarrow{trg} el_{H_T} \end{aligned} \quad (6.21)$$

Due to the construction of the trace graph and of use and trg edges, this property ensures that the information in the trace graph is inherently consistent.

Based on above definitions and the properties of STT graphs, the single-variant derivation which represents a single-variant model transformations is defined as follows:

Definition 6.3.9: Single-Variant Derivation

Let P be a set of trace-generating STT rules and S and T be a source and a target graph, respectively. T is single-variant derivable from S via P ($S \xrightarrow{P^*} T$) if STT graphs $G = G_S \leftarrow G_{TR} \rightarrow G_T$ and $H = H_S \leftarrow H_{TR} \rightarrow H_T$ exist, satisfying the following conditions:

$$G_S = S \wedge G_{TR} = \emptyset \wedge G_T = \emptyset \quad (6.22)$$

$$G \xrightarrow{P^*} H \quad (6.23)$$

$$H_T = T \quad (6.24)$$

$G \xrightarrow{P^*} H$ is a complete STT derivation, thus, completeness carries over from general derivations to single-variant derivations.

Example 6.3.4: Transformation Derivation

In Fig. 6.3.4, the target graph G_T has been derived from the source graph G_S according to Def. 6.3.9.

Thus, by employing relations between STT graphs (Def. 6.3.8), Def. 6.3.9 specifies a relation between source and target graphs and allows to formalize *out-place batch transformations* (Properties 5.3.3 and 5.3.1, respectively) in this way.

Summing it up, the definitions of rules and derivations in Sec. 6.3.1 and Sec. 6.3.3 conform to all properties of the computational model postulated in Sec. 5.3.3.

6.4 Trace-Based Annotation Propagation

After having defined single-variant transformations and multi-variant graphs, this section describes how the mapping function for a target graph is created from the mapping function of the source graph. Effectively, these steps propagate the annotations from the source to the target graph.

6.4.1 Propagation Algorithm

Alg. 1 describes how the mapping function for the multi-variant target graph can be computed: Please note: for the sake of a consistent representation, Alg. 1 is noted in the graph formalism. Its implementation is in accordance with the common trace model (Sec. 5.2.3) and the informally stated algorithm for trace-based propagation (Sec. 5.3.2).

Applying the algorithm to a (completely derived) graph $H = H_S \leftarrow H_{TR} \rightarrow H_T$, where a mapping function $map_{F_{H_S}} : EL_{H_S} \rightarrow A_F$ for H_S exists, results in a mapping function for H_T which conforms to the definition of multi-variant graphs (Def. 6.2.4). As a consequence, $MH_{S_F} = (H_S, map_{F_{H_S}})$

Algorithm 1 Trace-based propagation of annotations.

```

1: procedure PROPAGATE( $H, map_{F_{H_S}}, map_{F_{H_T}}$ )
2:   in  $map_{F_{H_S}} : EL_{H_S} \rightarrow A_F$            ▷ Annotation function for the source graph  $H_S$ 
3:   in  $H = H_S \leftarrow H_{TR} \rightarrow H_T$          ▷ Target STT graph, derived from  $H_S \leftarrow \emptyset \rightarrow \emptyset$ 
4:   out  $map_{F_{H_T}} : EL_{H_T} \rightarrow A_F$          ▷ Annotation function for the target graph  $H_T$ 
5:
6:   var  $SRC \subseteq EL_{H_S}, CTX \subseteq EL_{H_T}, TRG \subseteq EL_{H_T}$    ▷ Sets of source, context, and target
   elements
7:   var  $W_{H_{TR}} \subseteq N_{H_{TR}}$                                ▷ Working set of trace nodes
8:   var  $n_{H_{TR}} \in N_{H_{TR}}$                                ▷ The current trace node to be processed
9:   var  $a_F \in A_F$                                          ▷ Annotation to be assigned to target elements
10:  var  $te \in EL_{H_T}$                                        ▷ The target element to be annotated
11:
12:   $W_{H_{TR}} := N_{H_{TR}}$                                    ▷ Initialize working set of trace nodes
13:  while  $W_{H_{TR}} \neq \emptyset$  do
14:     $n_{H_{TR}} := \text{SELECT}(W_{H_{TR}})$                    ▷ Select trace node in topological order
15:     $W_{H_{TR}} := W_{H_{TR}} \setminus \{n_{H_{TR}}\}$          ▷ Remove trace node from working set
16:    ▷ Determine source, context, and target elements (using arrow notation for edges):
17:     $SRC := \{se \in EL_{H_S} \mid n_{H_{TR}} \xrightarrow{src} se\}$ 
18:     $CTX := \{ce \in EL_{H_T} \mid n_{H_{TR}} \xrightarrow{ctx} ce\}$ 
19:     $TRG := \{te \in EL_{H_T} \mid n_{H_{TR}} \xrightarrow{trg} te\}$ 
20:     $a_F := \bigwedge \{map_{F_S}(src) \mid src \in SRC\} \wedge \bigwedge \{map_{F_T}(ce) \mid ce \in CTX\}$ 
21:    ▷ The annotation is a conjunction of source and context element expressions
22:     $a_F := \text{SIMPLIFY}(a_F)$                                ▷ Simplify the annotation
23:    for  $trg \in TRG$  do                                   ▷ Process all target elements
24:       $map_{F_{H_T}}(trg) := a_F$                              ▷ Annotate the target element
25:    end for
26:  end while
27: end procedure
    
```

is the multi-variant source graph from which the mapping function for the multi-variant target graph $MH_{T_F} = (H_T, map_{F_{H_T}})$ is computed by employing the algorithm. Due to the following properties, the mapping function is guaranteed to be consistent with the source mapping function.

1. Alg. 1 builds a mapping function which is *total*:

A complete derivation processes all derivation steps. The trace-generating STT derivation creates a trace element in each derivation step. Alg. 1 iterates all trace nodes while assuming that each element of H_T is referenced by exactly one target edge trg which is declared as property of STT graphs in Equation 6.19.

2. The computed annotation for each target element is *well-defined*:

According to Equation 6.20 the trace graph is acyclic which allows to perform a topological sort. Consequently, the SELECT operation in Alg. 14 retrieves a trace node from the set of open trace nodes only after all of its used trace nodes have been processed. In addition, according to Equation 6.21, context elements of a trace node possess exactly one incoming trg edge from a used trace node which is guaranteed to be processed beforehand. Therefore, the annotation of context elements is always accessed after it was assigned its annotation by the used trace node. As a consequence, the annotation computed in Alg. 24 is well-defined.

3. The mapping function assigns annotations which satisfy *referential integrity* as postulated as property of multi-variant graphs (Def. 6.2.4, Equation 6.9) such that:

$$\forall e_{H_T} \in E_{H_T} : \quad map_{F_{H_T}}(e_{H_T}) \Rightarrow map_{F_{H_T}}(s_{H_T}(e_{H_T})) \wedge map_{F_{H_T}}(t_{H_T}(e_{H_T})) \quad (6.25)$$

For an edge e in E_{H_T} two cases can occur: Either its ends are created in the same derivation step or at least one end in a previous derivation step. If both ends are created by the same derivation step, the same annotation will be assigned to the edge and its ends, satisfying Equation 6.25 trivially. If one end of the edge e is created in a previous derivation step and annotated with some annotation a'_F , the annotation mapped onto e will be combined with the annotation of its ends because those are context elements. As a result, the annotation function comprises the annotation for the edge (e.g., a_F) and the one of the ends, which altogether is $map_{F_{H_T}} = a'_F \wedge a_F$.

Both cases satisfy referential integrity.

Based on the previous definitions, it is possible to define a *multi-variant derivation*, which relates a multi-variant source graphs with a multi-variant target graph in one STT graph:

Definition 6.4.1: Multi-Variant Derivation

Let P be a set of trace-generating STT rules and $H = H_S \leftarrow H_{TR} \rightarrow H_T$ be the STT graph resulting from a complete STT derivation starting with $H_S \leftarrow \emptyset \rightarrow \emptyset$. Let further $MH_{S_F} = (H_S, map_{F_{H_S}})$ be a multi-variant source graph and let $map_{F_{H_T}}$ be the annotation function resulting from employing the trace-based propagation algorithm (Alg. 1) with H and $map_{F_{H_S}}$ as input.

Then, the multi-variant target graph $MH_{T_F} = (H_T, map_{F_{H_T}})$ is multi-variant derivable from the multi-variant source graph MH_{S_F} ($MH_{S_F} \xrightarrow{P_F^*} MH_{T_F}$).

6.4.2 Commutativity of Derivations

This section defines the commutativity criterion informally introduced in Sec. 1.2 to the graph notation and sketches the important properties to prove its satisfaction with the trace-based propagation.

Before explaining the commutativity criterion, we sum up the facts about single- and multi-variant derivations in relation with the computational model for trace-based propagation introduced so far:

- Since both, the single-variant transformation and trace-based propagation, exhibit *functional* behavior (c.f., Sec. 6.3.2), the result of a *terminating multi-variant* transformation is *unique up to isomorphism*.

Please note: this property will be guaranteed if the SELECT operation (Alg. 1, Line 14) retrieves the trace nodes in the partial order defined by the *use* edge creation in trace-generating STT rules.

- Executing a single-variant model transformation on a filtered source graph will terminate if the multi-variant model derivation performed on the multi-variant source graph terminates (c.f., Sec. 6.3.2). Similarly, the resulting graph is unique up to isomorphism.
- The filter function terminates and produces a single-variant graph which is unique up to isomorphism.

According to these properties, the successful execution of the *transform-filter* path in the commutativity criterion always terminates and produces a result unique up to isomorphism. The commutativity criterion compares this result with the outcome of the *filter-transform* path where the same feature configuration is provided to the filter function.

The commutativity theorem, visualized in Fig. 6.4.1, states that the graphs created by the *transform-filter* and *filter-transform* paths for the same feature configuration deliver equal results (up to isomorphism):

$$\begin{array}{ccc}
 MG_{S_F} = (G_S, \text{map}_{F_{G_S}}) & \xrightarrow{P^*_F} & MH_{T_F} = (H_T, \text{map}_{F_{H_T}}) \\
 \downarrow \text{filter}_F(MG_{S_F}, fc_F) & & \downarrow \text{filter}_F(MH_{T_F}, fc_F) \\
 G'_S & \xrightarrow{P^*} & H'_T \simeq H''_T
 \end{array}$$

Figure 6.4.1: Commutativity in graph formalism.

Theorem 1 (Commutativity). *Let P be a set of trace-generating STT rules, F and FC_F be a set of features and feature configurations over F , respectively, and $\text{filter}_F : \mathcal{G}_F \times FC_F \rightarrow \mathcal{G}$ be a graph filter function.*

*Let further $MG_{S_F} = (G_S, \text{map}_{F_{G_S}})$ be a multi-variant source graph and let $MH_{T_F} = (H_T, \text{map}_{F_{H_T}})$ be a multi-variant target graph, which is multi-variant derived from MG_{S_F} ($G_{S_F} \xrightarrow{P^*_F} H_{T_F}$).*

Then, the following proposition is satisfied for each feature configuration $fc_F \in FC_F$:

Let $G'_S = \text{filter}_F(MG_{S_F}, fc_F)$ and $H''_T = \text{filter}_F(H_{T_F}, fc_F)$ be the filtered (single-variant) source and target graph, respectively. Finally, let H'_T be single-variant derivable from G'_S .

Then, H'_T and H''_T are equal up to isomorphism: $H'_T \simeq H''_T$.

The proof utilizes the following statements declared before: Let $G \xrightarrow{P^*} H$ be the complete STT derivation, where $G = G_S \leftarrow \emptyset \rightarrow \emptyset$ and $H = H_S \leftarrow H_{TR} \rightarrow H_T$ are the multi-variant graphs (without the mapping function). Similarly, let $G' \xrightarrow{P^*} H'$ be the underlying complete STT derivation for the single-variant graph, where $G' = G'_S \leftarrow \emptyset \rightarrow \emptyset$ and $H' = H'_S \leftarrow H'_{TR} \rightarrow H'_T$. Per definition the source graphs remain the same during derivations, such that $G_S = H_S$ and $G'_S = H'_S$, because the source graph is not modified by a STT derivation (Def. 6.3.8). This property is a consequence of the definition of STT rules (Def. 6.3.6).

$$\begin{array}{ccc}
 G & \xrightarrow{P^*} & H \\
 \uparrow m_G & & \uparrow m_H \\
 G' & \xrightarrow{P^*} & H'
 \end{array}$$

Figure 6.4.2: Monomorphism preservation by complete derivations.

As a second fact, the filtered graphs are sub-graphs of the multi-variant graphs, such that $G' \subseteq G$, which implies that the identity on G' is a monomorphism $m_G : G' \rightarrow G$. Since monomorphisms are preserved by complete derivations (which is proven in [WG20a]), such that the diagram depicted in Fig. 6.4.2 holds, a monomorphism $m_H : H' \rightarrow H$ exists as well. Applying this property to the (transformations of the) commutativity diagram results in the diagram depicted in Fig. 6.4.3, which is still silent on the properties of H''_T .

To prove commutativity, the restriction m_{H_T} of the monomorphism m_H to H'_T , i.e., $m_{H_T} = m_H|_{H'_T}$, is utilized which is sketched in Fig. 6.4.3, too. It is shown that the range of m_{H_T} comprises exactly the graph H''_T such that the following equation holds:

$$\text{ran}(m_{H_T}) = H''_T \tag{6.26}$$

From this equation, it can be deduced that $m_{H_T} : H'_T \rightarrow H''_T$ is surjective resulting in H'_T and H''_T

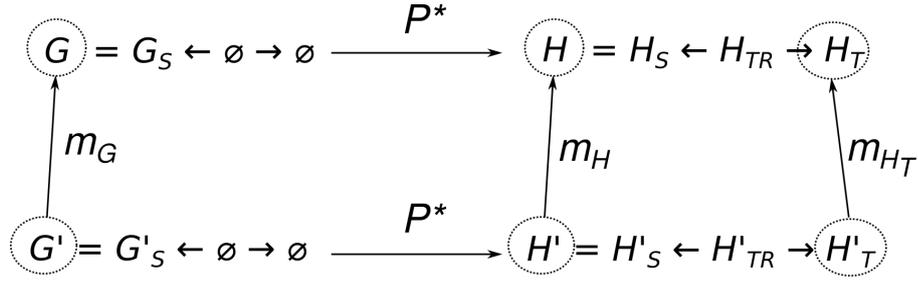


Figure 6.4.3: Application of the monomorphism preservation to commutativity diagram.

being isomorphic: $H'_T \simeq H''_T$.

Rephrasing Equation 6.26 uses the mapping morphism for the elements of H_T which is denoted as $m_{EL_{H_T}}$ and the knowledge that elements of H''_T are only visible if their mapping function is implied by the feature configuration:

$$\forall el \in EL_{H_T} : el \in \text{ran}(m_{EL_{H_T}}) \Leftrightarrow v_F(\text{map}_{F_{H_T}}(el), fc_F) = \text{true} \quad (6.27)$$

For the proof of Equation 6.26 and Equation 6.27, the derivation sequence of G is employed as well as the knowledge that only elements, the annotation of which is implied by the feature configuration fc_F , pass the filter function to become parts of G'_S and H''_T . The monomorphism which maps the filtered graph onto the multi-variant graph reflects the latter property. We state the property here for the elements of the source graph where it is guaranteed to be satisfied

$$\forall el \in EL_{G_S} : el \in \text{ran}(m_{EL_{G_S}}) \Leftrightarrow v_F(\text{map}_{F_{G_S}}(el), fc_F) = \text{true} \quad (6.28)$$

For the derivation sequence G_0, \dots, G_n we know that it is increasing monotonically, each source graph $G_{S,i}$ in the sequence, where $0 \leq i \leq n$, is the same as G_S , such that $G_{S,i} = G_S$ and that the resulting graph G_n equals H , such that $G_n = H$. Furthermore, the derived target sub-graph of G is the same as the target graph of H , such that $G_{T,n} = H_T$.

To this end, for an element to be present in H''_T the annotation of the element has to be visible given the feature configuration. According to Alg. 1, the annotation is computed from the source and context elements stored in the trace. We know that the annotation of an element is visible if and only if the annotations of all its source (*SRC*) and context elements (*CTX*) are visible as well. Based on this knowledge and the fact that all elements in G_T which are present in the range of $m_{EL_{H_T}}$ have to be visible in the given feature configuration, the following equation holds in each derivation step i :

$$v_F(a_F, fc_F) = \text{true} \Leftrightarrow \forall s \in \text{SRC}_i : s \in \text{ran}(m_{EL_{G_S}}) \wedge c \in \text{CTX}_i : c \in \text{ran}(m_{EL_{H_T}}) \quad (6.29)$$

If and only if matches (ρ'_i, m'_i) in the complete single-variant derivation $G' \xrightarrow{P^*} H'$ and (ρ_i, m_i) in the complete multi-variant derivation $G \xrightarrow{P^*} H$ exist, satisfying the following two properties, all source and context elements are guaranteed to be members of the respective ranges as postulated in Equation 6.29:

1. $\rho_i = \rho'_i$
2. The monomorphism from H' to H , m_H , maps the elements of match m'_i onto those of match m_i .

As a consequence, if and only if the matches (ρ_i, m_i) and (ρ'_i, m'_i) satisfy these properties, the annotation assigned to the target elements in derivation step i evaluates to true. Matches correspond with each other if and only if the elements created by applying the respective rule correspond with each other. This knowledge, results in the following fact:

$$v_F(a_F, fc_F) = \text{true} \Leftrightarrow t_i \in \text{ran}(m_{EL_{H_T}}) \quad (6.30)$$

Since the new target elements which evaluate to true would be part of the next derivation step it can be concluded that the rephrased property to prove commutativity (Equation 6.27) is satisfied by the construction of the propagation algorithm in accordance with the definition of the single- and multi-variant derivations.

6.5 Summary

Altogether, this chapter formally defines multi-variant M2M transformations by employing Graph theory. By defining graphs and their transformation, firstly, by a set of in-place rules and secondly, by out-place rules, the chapter formalizes the concept of trace-based propagation, subsequently. Sec. 6.3.2 and Sec. 6.3.3 explain to what extent the definitions conform to the computational model for trace-based propagation before presenting the propagation algorithm in Sec. 6.4.1. Finally, Sec. 6.4.2 summarizes the essential steps for proving commutativity of the proposed propagation approach.

Despite proving commutativity for the trace-based annotation propagation conforming to the computational model, existing transformation languages or their engines may violate the computational model in different respects: Transformation engines exist which do not store complete, fine-grained trace information permanently. The corresponding languages may be more powerful, for instance, they may support non-local or non-functional rules. Similarly, transformations may be performed incrementally to prevent redundant computations. Therefore, the following part of the thesis presents annotation propagation strategies for transformation scenarios which violate the computational model.

Part IV

Extensions to Trace-Based Annotation Propagation

Chapter 7 Missing Trace Information

*Untersuchen was ist, und nicht was behagt*¹

Johann Wolfgang von Goethe
(The Attempt as Mediator of Object and Subject, 1792)

~

As foreshadowed by the concluding summary of the previous chapter, the computational model for the trace-based propagation may be too restrictive to be applicable in several real-world transformation scenarios. On the one hand, as a consequence of different granularity, the trace information may not be complete, rendering a trace generation-complete or incomplete. On the other hand, some transformation engines do not generate or persist a trace at all. For instance, the default ATL virtual machine constructs a trace for a subset of the rule types, the *matched rules*, internally but does not store the trace permanently after the execution. For this reason, this chapter offers solutions to scenarios in which the level of trace completeness and granularity is less than the one expected by the computational model for trace-based propagation.

Fig. 7.0.1 presents the organization and relationships of the sections of this chapter. The computational model for trace-based propagation postulates *complete* and *fine-grained* trace information to satisfy commutativity. Accordingly, the first section discusses the impact of a generation-complete trace on the commutativity criterion. As this information suffices to create a completely annotated target model automatically, the trace-based propagation is not modified.

Conversely, Sec. 7.2 offers a solution for the situation in which either a generation-complete or complete trace is available but the persisted information is more coarse-grained than the granularity at which annotations can be assigned to model elements. For instance, while the trace only stores corresponding source and target *objects*, annotations may also be mapped onto their *structural features*, such as their names. Then, the trace-based propagation can be executed as preliminary step to compute annotations for the corresponding elements recorded in the trace. A following *analysis* of the transformation definition, represented as bytecode instruction model, serves to determine the missing fine-grained annotations.

If the trace is incomplete instead, Sec. 7.3 offers heuristic completion algorithms to determine missing annotations in the target model automatically whereas Sec. 7.4 presents two possibilities to automatically reconstruct trace information without analyzing the transformation in the case that a trace is unavailable.

¹ Investigate what is, and not what pleases.

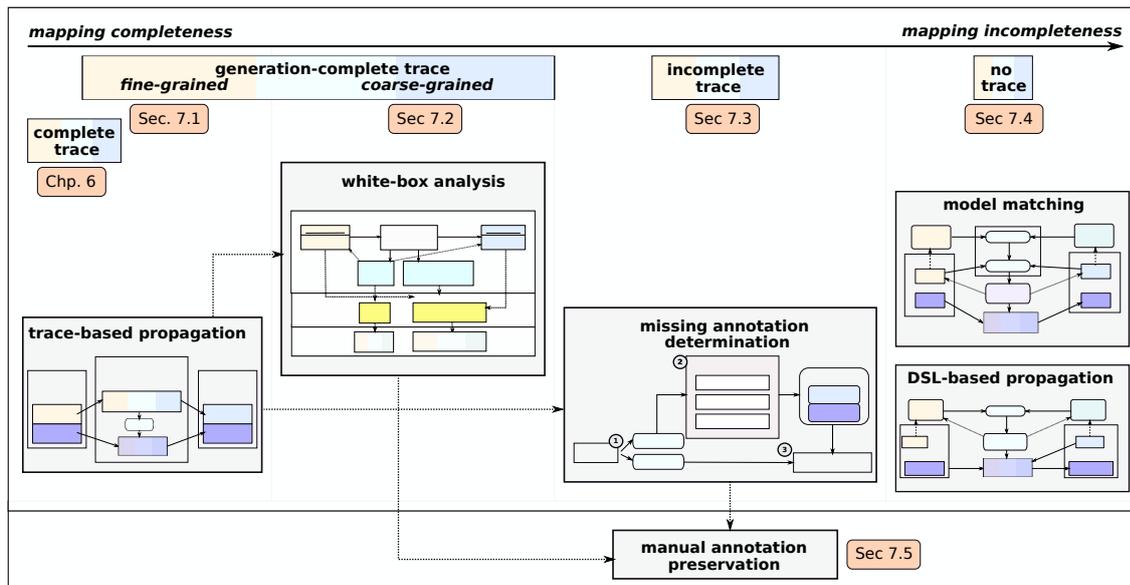


Figure 7.0.1: Overview and interplay of strategies to maintain missing trace information.

To this end, due to the missing mapping information, we apply heuristic strategies to complete annotations. As a result, some elements in the target model may not be annotated correctly, such that it may be necessary to fix the annotations manually. The requirement to repair target annotations may occur if the trace is generation-complete or too coarse-grained and if the annotation completion relies on heuristics. Manually assigned annotations, however, should be pertained in consecutive transformations of the same source model, to preserve the manual effort. Therefore, Sec. 7.5 describes how to preserve manual annotations in incremental transformations and how to maintain them in an incremental propagation.

[GSW17], [GW18c], [BG18], [GW19c], [GW20] and [GNS22] lay the grounds for the contents of this chapter.

7.1 *Generation-Complete* Traces

This section illuminates the situation when the reused transformation engine persists *generation-complete* instead of *complete* traces. At first, Sec. 7.1.1 describes the problem provoked by generation-complete traces: the derived target model may comprise too many elements which are not created by the single-variant transformation performed on the derived source model because the context element is missing. While an annotation propagation based on a generation-complete trace still yields a completely annotated model (Sec. 7.1.2), it may impact the correctness of the propagated annotations, as an example in Sec. 7.1.3 illustrates. The closing section draws a conclusion of these observations.

7.1.1 Problem Description

Per definition, a generation-complete trace consists of an ordered list of trace elements which record source elements and the target elements which are created and added to the target model by applying a rule. As opposed to a complete trace, the generation-complete trace does not store context elements, i.e., already existing target elements which are *necessary* to create a new target element. Similarly, this trace does not store further source elements which were responsible to create the context elements.

According to the computational model for trace-based propagation, propagating annotations based on a *generation-complete* trace cannot guarantee commutativity, in general. By postulating complete traces, the computational model regards the fact that a transformation rule requires the presence of the context element to construct the new target elements. Thus, the annotation of the context element has to be satisfied in order to create the target element because otherwise the context element would be missing and the target element could not be created by the single-variant model transformation.

However, even though generation-complete traces violate the computational model for trace-based propagation, this section discusses the implications of propagating annotations based on a generation-complete trace: On the one hand, a trace of such granularity ensures a completely annotated target model, nonetheless. Since the generation-complete trace records each created element in the target model, the propagation algorithm can map an annotation onto each of them. On the other hand, the effect on commutativity may depend on the contents of the rule, particularly on the reason why another target element is considered a context element.

7.1.2 Completely Annotated Target Model

Although a generation-complete trace does not record context elements, the trace-based propagation can combine the annotations of the source elements in a conjunction and assign them to the elements which are created by applying the corresponding rule in the target model. Since a generation-complete trace records all elements of the target model, which were created by the transformation, such that Equation 6.13 (each element in the target graph is referenced by exactly one *trg*-edge) still holds, an annotation will be mapped onto all target elements after the trace-based propagation has finished.

Consequently, the result of a trace-based propagation based on a generation-complete trace behaves differently than a propagation based on incomplete traces, which is discussed in Sec. 7.3. The information in incomplete trace cannot ensure a completely annotated target model which leaves room for uncertainties about the presence of target elements in derived products. Conversely, propagating annotations based on a generation-complete trace does not require to compute missing annotations and ensures that each target element carries the annotation of its corresponding source elements.

7.1.3 Correctness of Propagated Annotations

In general, the missing information of a context element may provoke the situation that a target element is not created by the single-variant model transformation due to the missing context

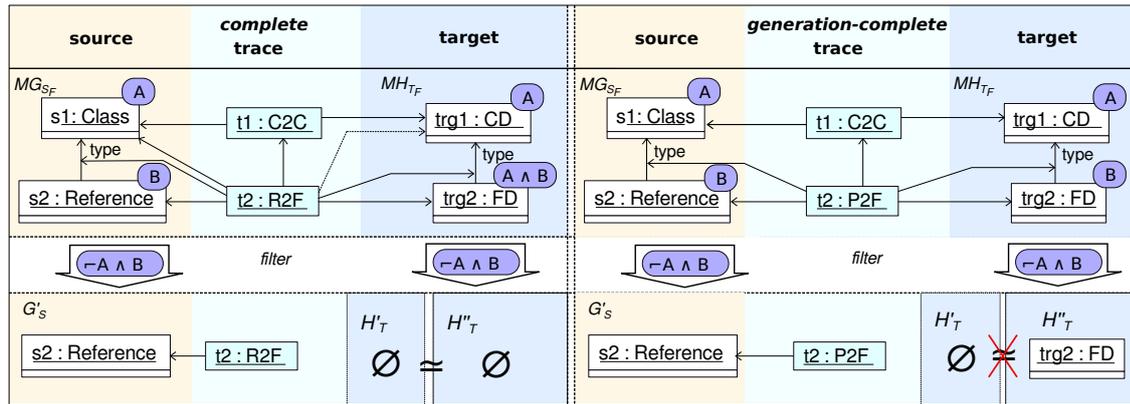


Figure 7.1.1: Commutativity in propagation based on *generation-complete* vs. *complete* trace. The *type*-link is annotated with the same annotation as the *s2* and *trg2*, respectively.

element (and the source element corresponding with the context element). This may be the case either because the presence of the source element which is responsible for creating the context element or the presence of the context element itself is necessary to *match* the rule and to generate the target element or because the missing source elements yield a malformed derived source variant and prevents the single-variant transformation to be executed at all. Ex. 7.1.1 explains such situation in a class model to Java model transformation.

Example 7.1.1: Violation of Commutativity due to Generation-Complete Trace

Fig. 7.1.1 illustrates an example of violated commutativity caused by generation-complete (fine-grained) trace information. The example demonstrates a rule transforming a reference into a field declaration. The multi-variant source model MG_{S_F} consists of a class *s1* and a reference *s2* which are annotated with the two optional features *A* and *B* (which are not related by further constraints).

The transformation specification requires the presence of the *type* of the reference to execute the rule *R2F*. If the source class which is linked as *type* is not present (i.e., *null*), the transformation will not execute the rule because of a respective application condition^a. Accordingly, the complete trace records the class *s1* besides *s2* and the *type*-link as source element for the rule application *R2F* and the class declaration *trg1* as context element whereas the generation-complete trace, depicted on the right side, only records the source and target element, *s2* and *trg2* and their respective *type*-links, respectively.

The propagation based on the *complete* trace, depicted in the left part of the figure, maps the annotation $A \wedge B$ onto *trg2* and its *type*-link, thereby combining the annotations of the second source element *s1* and the context element *trg1* with the one of the actual source element *s2* and its *type*-link (and removes each of the redundant features *A* and *B*).

The derived single-variant source model on the left hand-side comprises the reference *s2* only. The filter removes the *type*-link because it would be a dangling link. Consequently, the single-variant transformation will not execute rule *R2F* due to the missing required source type. Thus, H_T' remains empty.

Deriving the target model H_T'' from the multi-variant target model MH_{T_F} , depicted on the left hand-side, which was annotated based on the *complete* trace, yields a commuting transformation: The derived target model H_T'' is empty, too.

The right part of the figure depicts the result of annotating the target model based on the *generation-complete* trace. As opposed to the left side, the field declaration *trg2* remains in the derived variant H_T'' after deriving H_T'' from the MH_{T_F} whereas the filter removes the *type*-link again due to the missing target element. Nevertheless, the annotation propagated based on the *generation-complete* information causes a violation of commutativity.

^a The rule is (still) local because the type of the reference is part of the left side which needs to be matched in the input model.

Please note: A similar situation as in Ex. 7.1.1 may occur if the transformation engine transforms *valid* models only. If, instead of guarding the rule with an application condition, a well-formedness rule is present on the source metamodel, the single-variant transformation may not be executed at all. In this example a well-formedness rule may foster that the type of a reference must not be `null`. If the transformation engine checks whether the input model is valid before initiating the transformation, it will not execute the transformation specification. As a consequence, H'_T is empty and commutativity may be violated, too. This may also be the case, if a hierarchical filter is applied to the target model because it would only regard well-formedness rules associated with the target element for which the annotation of the context element is missing.

On the whole, the example demonstrates that a missing context element may provoke the situation where a rule is not executed due to a missing element and, consequently, that the target elements are not created. However the annotation mapped onto the target element in the multi-variant target model, which misses the annotation of the context element, provokes the inclusion of the target element in the derived variant. Thus, while in some scenarios generation-complete traces suffice to satisfy commutativity, as the evaluation will show (c.f., Sec. 10.3), a generation-complete trace alone cannot guarantee commutativity.

7.1.4 Consequences

All in all, a propagation based on a generation-complete trace ensures a completely annotated target model in the first place. Furthermore, the propagation maps the annotation of source elements onto the corresponding target elements which are created by the rule application. Accordingly, whenever the source elements are present in a derived product the corresponding target elements will be present as well. However, as a consequence of missing context elements, the single-variant transformation performed on the derived source model may not create the target elements, which are present in the derived target model. Thus, it may violate commutativity, in general.

7.2 Coarse-Grained Traces

The computational model postulates that traces are as *fine-grained* (c.f., Prop. 5.3.8) as the model filter and the mapping annotation function as one criterion for a correct propagation of annotations. If the trace persists more coarse-grained information only, necessary annotation information may be missing and commutativity will not be guaranteed anymore. Solving this problem requires to extract information about corresponding elements which is as fine-grained as the model filter and the mapping annotation function.

The following descriptions investigate the situation in which traces store correspondences between all *objects* but not between their *structural features* (i.e., their attributes and references)². Structural features of objects appear, for example, in each instance of the Ecore metamodel (c.f. Sec. 9.1.1). If the structural features of target objects possess customized values apart from a default one, the transformation specification may assign and transform specific values to these structural features. For instance a UML class to Java model transformation may assign the name of a UML class to the corresponding Java class declaration. As a consequence, at least the *transformation specification* encapsulates the knowledge which source elements and structural features are used to create a structural feature of a target object. In a similar way, *low-level bytecode* contains the similar information but it is represented in a way that is more agnostic of the corresponding transformation language, particularly of its syntax. Therefore, the transformation specification represented in form of bytecode instructions may be exploited complementary to the present trace information about corresponding objects.

This section sheds light on *white-box analysis* strategies to complement the propagation of annotations in multi-variant model transformations and discusses the benefits and shortcomings of analyzing a bytecode model. Thus, the following descriptions assume that the target model was

² The constellation of fine-grained mappings combined with a coarse-grained trace occurs, for instance, when propagating annotations of multi-variant models created with the tool Famile based on an ATL/EMFTVM trace.

already created and that a trace-based propagation of annotations at the object level has already been performed.

Road Map Sec. 7.2.1 illustrates the problem based on an example which maps an annotation, which is more specific than the one of the source object, onto one of the source object’s attributes which violates commutativity. Based on this example, it draws conclusions on the correctness of the transformation and sketches solution ideas.

The subsequent sections present how to analyze the bytecode of model transformations. At first, this task requires to be familiar with bytecode instructions and their relations with transformation rule assignments. Therefore, Sec. 7.2.2 establishes a general understanding of the bytecode instruction analysis. When analyzing the instructions patterns which assign structural features of source objects to structural features of target objects, assignment patterns have to be recognized which are explained in Sec. 7.2.3. The following sections, Sec. 7.2.4 and Sec. 7.2.5, derive the propagation algorithm based on identifying assignment patterns in bytecode instructions informally and formally. To conclude this section, Sec. 7.2.6 discusses the pros and cons of additionally analyzing bytecode instructions for propagating fine-grained annotations as well as related work on model transformation analysis.

7.2.1 Problem Statement

This sections illustrates the problem which occurs if the mapping granularity of the source and target model resides at a more fine-grained level than the correspondences between source and target elements stored in the trace.

Assumptions The following descriptions assume that (at least) a coarse-grained generation-complete trace is available. Consequently, a transformation with MG_S as input has created an output model MH_T . Moreover, correspondences between all source and created target objects are available in form of a trace and annotations have been mapped onto all target objects corresponding with source objects. However, the coarse-grained trace granularity does not always suffice to correctly annotate the target model, as the following example demonstrates.

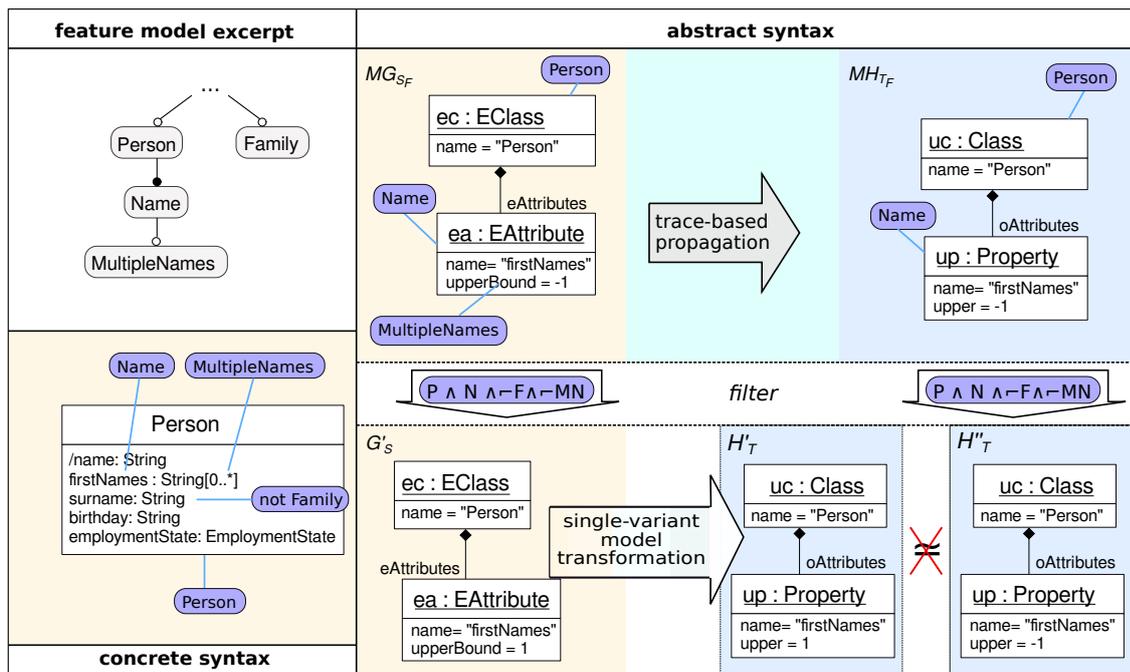


Figure 7.2.1: Commutativity violation due to fine-grained annotation mapping.

Example 7.2.1: Annotation Violation Provoked by Coarse-Grained Trace

Fig. 7.2.1 sketches how a coarse-grained annotation for a structural feature of an object violates commutativity. The figure illustrates an Ecore to UML class model transformation which converts an excerpt of the database product line model into the corresponding UML class model.

On the left, the top of the figure captures relevant excerpts of the adapted feature model whereas the bottom depicts the model fragment and its annotations in concrete syntax. The feature model is extended by an optional feature `MultipleNames` which serves as child of the mandatory feature `Name`. The latter is selected whenever the feature `Person` is selected in a feature configuration.

A product line developer has mapped the feature `MultipleNames` as annotation onto the value of the EAttribute `upperBound` which defines the number of allowed first names of a person. Consequently, in this example, it is possible to annotate the structural features of objects, which, for instance, the MDPLE tool `Famile`, supports. Furthermore, the presence of a structural feature depends on the presence of its objects. Accordingly, no annotations are mapped onto the remaining structural features because they carry the annotation of the object which they refine implicitly. Thus, they can only exist, if the object they refine exists, too.

The right side of the figure notes the transformation scenario in *abstract syntax* which depicts how annotations are mapped onto the structural features of objects. On the top, the figure shows the relevant excerpt of the multi-variant source and the multi-variant target model. The latter results from a trace-based propagation based on a coarse-grained trace, which only records source and target *objects* but no correspondences between their structural features. While the property which is created for the source attribute receives the annotation `Name`, no annotation is mapped onto the UML structural feature `upper` which corresponds with the structural feature `upperBound`.

On the bottom of the right side, the figure depicts the filtered models for the feature configuration which selects the features `Person` and `Name` and deselects the features `Family` and `MultipleNames`. As the annotation of the attribute `upper` is not satisfied by the feature configuration, the model filter removes the value and the default value 1 replaces the multiplicity of -1. Transforming this model with the single-variant model transformation creates the transformed target model H'_T where the attribute `upper` is assigned the same value 1. However, the derived target model H''_T still comprises the value -1 for the same attribute. Consequently, H'_T and H''_T mismatch and violate commutativity.

Consequences This example allows for drawing the conclusion that trace information which is more coarse-grained than the mapping mechanism cannot guarantee commutativity. If the mapping mechanism maps annotations at the level of structural features while the used transformation trace records source and target elements at the object level, the values of the structural features may diverge between the transformed single-variant source model and the derived target model. Thus, coarse-grained traces may threaten commutativity.

Solution Ideas Two solutions may mitigate this problem:

Firstly, the trace information could be extended which requires to inject functionality into the execution to record and map also structural features of objects. In practice, only few examples of traces, which persist such fine-grained information [Bec+07], already exist. Changing the execution engine instead, requires the rights to modify the engine and represents an engine-specific solution. Thus, it contradicts the research objectives to reuse existing transformation capabilities without modifications (item **RO1**) and to provide a generic solution (item **RO3**).

Secondly, the transformation specification which encodes the information as well could be analyzed. Particularly, some execution engines, such as eMoflon [LAS15] and the ATL/EMFTVM [Wag+12], persist execution models, resulting from applying the transformation. These execution models can be analyzed and used to extract the information about corresponding elements in a more general way than parsing and analyzing a specification written in a specific transformation language.

7.2.2 Bytecode Instruction Analysis

This section introduces the background knowledge for analyzing bytecode instructions to propagate fine-grained annotations. First, the section provides an overview of the dependencies between rules, traces and bytecode instructions. For extracting corresponding elements, the essential activity is to recognize assignment patterns in the bytecode instructions. Therefore, the second part of this section gives an introduction to bytecode instructions, in general, and continues to give a concrete example of an ATL transformation rule and the corresponding bytecode instructions, in particular.

Overview of Analyzing Transformation Rules and the Generated Artifacts Concluding from the sketch to solve the problem (c.f., Sec. 7.2.1), one way to gain a more fine-grained mapping may extend the trace information which requires to change the execution behavior to persist more fine-grained annotations during the execution. Alternatively, the *transformation specification* or its *execution model* can be analyzed. Thus, this section offers an overview on the dependencies between a transformation specification, traces, and bytecode instructions. These dependencies and corresponding technical considerations lay the ground to derive and examine the bytecode analysis in subsequent sections.

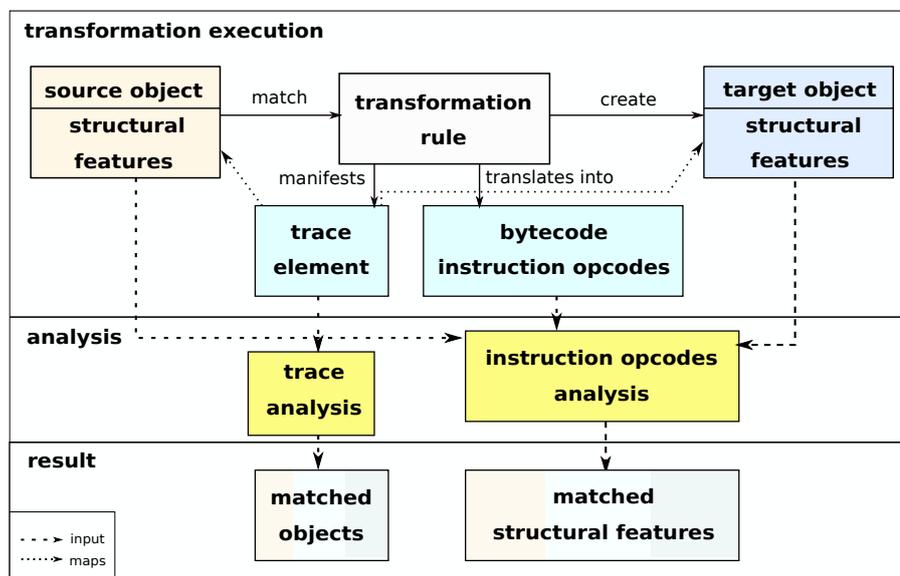


Figure 7.2.2: Schematic overview of bytecode instruction analysis-based propagation.

Fig. 7.2.2 presents a schematic overview of the dependencies between a *transformation rule*, a corresponding *trace element*, the transformed objects and the *bytecode instruction opcodes* into which the rule is translated. The figure exemplifies these dependencies based on applying one rule, which transforms one source object into one target object. This single rule application serves as representative for the entire transformation consisting of several of such and potentially more complex rule applications. A more complex rule may comprise several source and target elements. The *transformation execution* part, depicted on top of the figure, comprises the transformation rule, a trace element which is generated by applying the rule, and corresponding bytecode instructions into which the transformation rule is translated. A source object matches the rule and the rule creates a target object. Even though not depicted in the figure, other kinds of rules may also match several source objects and create or employ several target objects. A (generation-)complete trace records all matched source elements and all created target elements in the trace element whereas the *bytecode instruction opcodes* field summarizes all instructions being executed in order to generate the target elements.

While the middle part of the figure shows the two possibilities of either *analyzing* the trace or the instruction opcodes resulting from the execution, the part at the bottom presents the *results* of

Table 7.1: Overview of basic ATL/EMFTVM bytecode instructions.

Mnemonic	Description
LOAD <localVariable>	Load the reference to the given local variable onto the operand stack
GET <fieldname>	Fetch the value stored in the given field of the loaded object
PUSH <value>	Push the given value onto the operand stack
SET <fieldname>	Assign a value computed before to the given field
INVOKE <opname>	Perform the given operation that is specified (e.g., a concatenation operation or a trace lookup)
IF <offset>,<trg>	Branch the execution based on a condition
ITERATE <offset>,<trg>	Iterate a collection which provokes a branch the execution
GOTO <offset>,<trg>	Go to the instruction specified as target

these analyses: Analyzing the information from the trace element only, results in *matched objects* which will not suffice if annotations are mapped onto their structural features. If the *bytecode*, into which the transformation specification is translated, is analyzed, it will be possible to retrieve more fine-grained mappings. The bytecode consists of instruction opcodes which can be iterated in the analysis step to detect assignments of source structural features to target structural features. Particularly, this step can also include the analysis of supportive methods that compute a complex value, such as OCL expressions stated in queries or helper definitions. In a similar way, the rules of the transformation specification itself could be analyzed which requires a language-specific parser which is aware of the entire syntax and semantics of the language. For instance, the execution of a transformation rule in ATL depends on its nature and can be initiated by a global matching (top rules) or by invoking it from another rule (lazy and called rules).

In summary, either the transformation specification or the bytecode can be analyzed to retrieve fine-grained correspondences between the source and created target model. In the figure, this would mean either analyzing the transformation rule or the bytecode instruction opcodes. Analyzing the transformation specification requires access to the specification and a parser, specific to the language. Since the syntax and semantics of a language undergoes evolution more often than the bytecode representation, the parser needs to be updated to new syntax and corresponding semantics provided by the transformation specification and the execution engine. In general, bytecode instruction opcodes, however, remain stable while the language changes: For example, the opcodes used in the Java virtual machine have not changed in between the Java SE 8 and Java SE 17 versions [Lin+15; Lin+21]. Therefore and due to the fact that the execution bytecode for a transformation specification may be available as a model or can be abstracted and translated into a model, we discuss how to analyze the bytecode model in the remainder of this section.

Overview of Relevant Bytecode Instructions Before diving into the analysis of transformation rules, this paragraph introduces important bytecode instruction opcodes, which represent assignment statements. For exemplifying the instructions we employ the names of the ATL/EMFTVM engine, which are closely related but not as details as the Java instruction opcodes.

To begin with, a *stack of instructions* represents a transformation rule. A summary of these relevant instructions is provided in Table 7.1. Each instruction possesses an opcode *mnemonic*, representing the opcode kind with a short term in natural language, and an *operand stack* of variables on which the opcode is executed. The first column of Table 7.1 states the mnemonic and the operands pushed on the stack whereas the second column offers a short description.

The upper part of rows in Table 7.1 holds instructions that interact with variables. Load-instructions load a local variable by retrieving it from the given reference and push it onto the operand stack. Similarly, get-instructions retrieve specific fields from the loaded reference. For instance, if a class is loaded, the field name can be retrieved. While PUSH instructions do not necessarily access an object to push the given value, which can also be static, onto the operand stack, the SET operation retrieves the value and assigns it to the given *fieldname*. Accord-

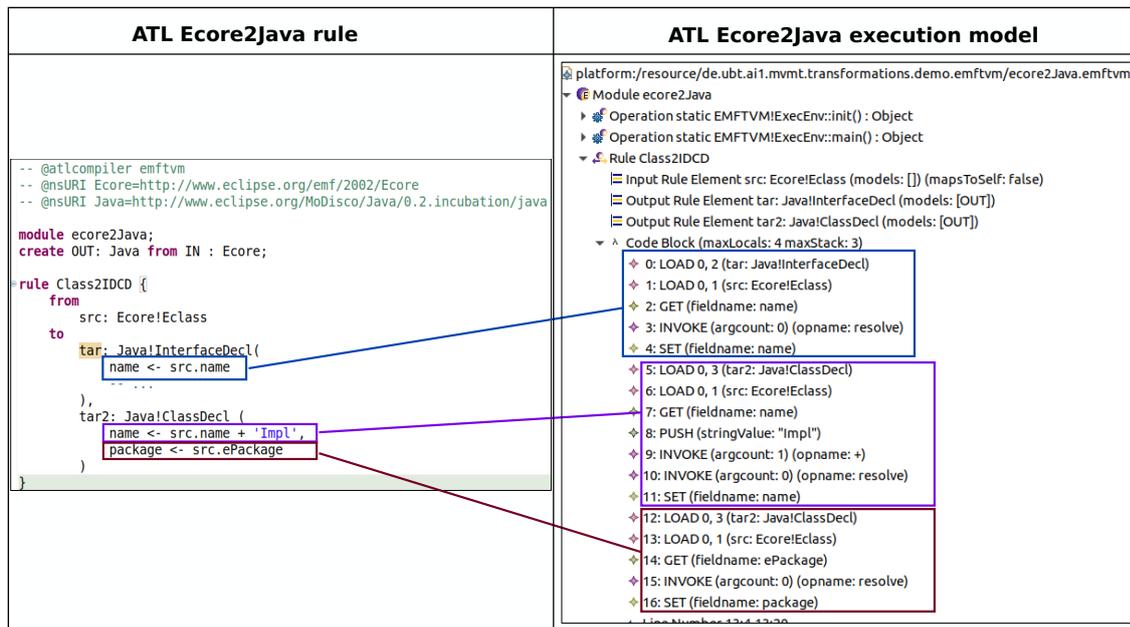


Figure 7.2.3: Mapping of ATL/EMFTVM rule onto bytecode instructions.

ingly, the EMFTVM execution model distinguishes `LocalVariableInstruction` (i.e., `LOAD`) from `FieldInstructions` (i.e., `GET` and `SET`) whereas the `PUSH` instruction is considered a regular instruction.

The lower part of the instruction table comprises *branch* instructions as well as the *invoke operation* instruction. While the *invoke operation* specifies the name of the operation that should be performed, *branch* instructions store an offset and a target instruction.

For more detail we refer the interested reader to the entire list of EMFTVM bytecode instructions depicted in Fig. A.2.1 which is retrieved from the corresponding ATL/EMFTVM Eclipse plugin³

Example of Representing Assignments as Bytecode Instructions A model transformation creates target elements according to the transformation specification. Regardless of the transformation paradigm, such as declarative or a procedural language, information is retrieved from the source model and used to create a target element. Based on that fact, we can identify common patterns which are used to assign a value to the structural features, i.e., the attributes and references, of an object.

Therefore, Ex. 7.2.2 presents a concrete example of an ATL rule and the corresponding bytecode instructions. The ATL/EMFTVM compiler records these instructions in a model at each *save* operation in the editing (Eclipse) workspace. The example and further code fragments which assign values in a transformation serve to classify and scrutinize types of patterns in the following sections.

Example 7.2.2: ATL Transformation Rule and Bytecode Instructions

Fig. 7.2.3 depicts an ATL rule, which creates a Java interface and a class declaration for each `EClass` of a given `Ecore` model, on its left side. On its right side, it shows the corresponding execution model created from the ATL/EMFTVM plugin.

While the first part of the ATL rule states the *from-pattern*, i.e., the characteristics required to match a source object, the *to-pattern* in the second part of the rule describes, which and how target elements are created. The rule creates a Java interface and class declaration as target elements. The bytecode model records the corresponding elements as *input* and

³ The ATL/EMFTVM execution metamodel is contained in the Eclipse plugin project: `org.eclipse.m2m.atl.emftvm`.

output rule elements, respectively.

The first assignment, `name <- src.name` occurs in the creation of the interface, which is highlighted with a surrounding blue rectangle. The statement assigns the name of the matching source object `src` as name to the interface. For this task, the assignment retrieves the value of the attribute `name` of the source object and assigns it to the corresponding attribute of the interface object which is created. The topmost blue rectangle on the right side highlights the resulting instruction opcodes. First, the opcodes `LOAD` the target and source element. Then, the name of the source object is `RESOLVED` and then `SET` to the one of the target object.

The second and third assignment occur in the part creating the class declaration. The second assignment `name <- src.name + 'Impl'` composes the name for the class declaration from the source name concatenated with the static String value `'Impl'`. Since the first part of the assignment is exactly the same as for the first assignment, the first part of opcodes is the same, too. In addition to the opcodes representing the first assignment, this assignment involves a `PUSH` opcode, which lays the static String value `Impl` on the stack, together with another `INVOKE` instruction, which stores the concatenation operator `+`. Based on executing these opcodes, in the end the concatenated name is assigned to the class declaration.

The third statement, `package <- src.ePackage` assigns the (first) element which is created for the referenced `ePackage` of the source object as the package of the class declaration. Even though the rule states that the `ePackage` of the source element is assigned to the target object, the ATL execution engine retrieves the corresponding target element from a runtime trace which it maintains to resolve objects. Interestingly, despite loading different source element types (attribute vs. referenced object, i.e., static vs. dynamic element), the opcodes in the execution model are the same for the first and third assignment.

Consequences This example allows for inferring that different assignment patterns occur which (may) rely on the values of structural feature of the matching source object. Either a source value is assigned directly, or it is composed of other static or dynamic values and assigned in this way. Similarly, referenced objects or their values may be used to retrieve the assigned value. More complex expressions can be stated on the right side of such assignment which the following paragraphs illuminate.

7.2.3 Classification of Patterns in Model Transformation Languages

As Ex. 7.2.2 shows, different forms of statements assign values to structural features of target objects. Depending on the paradigm, syntax and semantics of the transformation language, these assignments may be expressed at different stages of complexity. However, at an abstract level, the assignments share the goal of assigning a value to the structural feature of a target object, mostly by exploiting information of the source model.

Road Map Therefore, the first part of the following paragraphs introduce a taxonomy based on which assignment patterns can be classified. The second part illuminates concrete assignment patterns and their corresponding bytecode instruction opcodes whereas the concluding part collects observations which are important to propagate annotations.

I Assignment Pattern Taxonomy

Fig. 7.2.4 presents a taxonomy which classifies assignments and helps to gain an overview before exploring the details of specific assignments.

Occurrence Firstly, a transformation rule can state an assignment directly or indirectly. The ATL example of Fig. 7.2.3 demonstrates *direct* assignments only, which declare how to compute the assigned values from potentially different sources of information. In contrast, *indirect* assignments invoke another method to compute the assigned value. Such helper method may be implemented

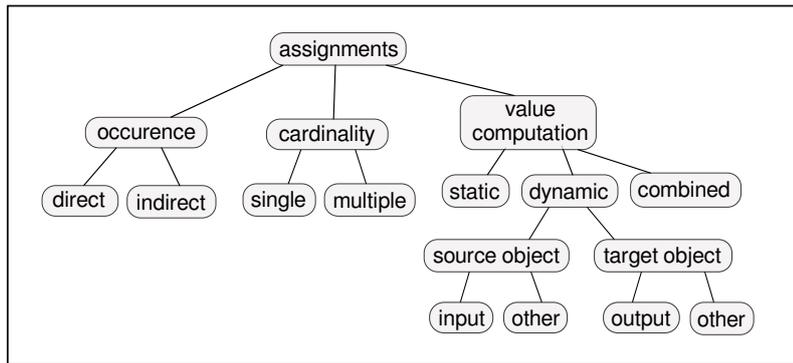


Figure 7.2.4: Classification of assignment patterns in model transformation languages.

either in the transformation specification or in an associated GPL file. The main difference between a direct and indirect assignment is that invoking a method creates a branch in the execution flow. As a consequence, different bytecode instructions represent the method call. Particularly, indirect assignments may invoke another method which may be defined in another file and potentially in another language.

Cardinality Furthermore, a transformation rule can use a *single* value (e.g., of the source object) or *multiple* ones to compute the assigned target value. The actual complexity of the assignment depends on the operations executed on these multiple values and the source from which the values are retrieved.

Value Computation The *computation* of values involves several stages of complexity. Assigning a *static* value resides at the lowest level of complexity and does not require any computation. Assigning a *dynamic* value instead, means to compute the information from model elements. At first, we can distinguish the type of object which is stated in the assignment. Either an object from the source or from the target model can be used to compute the assigned values. In the example presented in Fig. 7.2.3, only the input object serves as source from which the assigned values are extracted. While the first two assignments of this example retrieve values of attributes of the *input* source object, the third assignment retrieves the value from an object *referenced* as `ePackage` from the source object. Thus, the assigned value is computed either from the input source object or from another source element, which may be referenced by the source object or may be stored in a variable.

As an alternative, if a rule creates multiple target elements, it is possible to assign one of them as referenced by another of the created target elements. For instance, in Fig. 7.2.3 it is possible that the rule assigns the created interface declaration `tar` as one of the implemented interfaces by adding them into a corresponding reference of the created class declaration `tar2`. Thus, the rule uses the value of another *target* element of the same rule.

Furthermore, the values of a *context* element which is the target element of another rule might be used explicitly by employing the trace and providing some source object, e.g., referenced by the input source object of the rule. Such operation is stated by invoking an operation which accesses the trace during the execution and retrieves the corresponding created target element. As an example, ATL allows to access the trace by invoking the operations `resolve()` or `resolveTemp()`. It must be noted that stating the reference to a source object requires an implicit resolve operation which looks up the corresponding target elements in the trace, too. Even though the effect of stating a source object on the right side is the same as stating the target element which was created for that source object, the opcodes diverge.

II Assignment Patterns

Based on this classification, the following paragraphs introduce common patterns in more detail. Straightforward patterns are *single-value direct assignments* of exactly *one* value (e.g., of a source

element) to one structural feature of the target object. In contrast, multi-valued assignments compute the target value from multiple elements. The following paragraphs illuminate both types and further patterns subsequently.

Overview of Single-Value Assignments Table 7.2 enumerates types of assigning a single value to a single-valued structural feature of the created target object. It must be noted that the table incorporates assignment operations which reduces several distinct values to one. However, in each of these reductions only up to one *single* value from a structural feature of the input source object is retrieved which may be combined with several static values. The leftmost column classifies the assignment pattern whereas the second column gives an example in ATL. The rightmost column enumerates the corresponding bytecode instructions as represented in the ATL/EMFTVM execution model.

Table 7.2: Bytecode instruction opcodes of direct assignment patterns of *single* values.

Assignment Type	Example	Instructions
1-valued direct assignment, static	<code>name ← 'SomeName'</code>	LOAD(tar), PUSH, INVOKE(resolve), SET(name)
1-valued direct assignment, dynamic	<code>name ← src.name</code>	LOAD(tar), LOAD(src), GET(name), INVOKE(resolve), SET(name)
1-value direct assignment, combined	<code>name ← src.name + '.java'</code>	LOAD, LOAD, GET(name), PUSH(' .java'), INVOKE(+), INVOKE(resolve), SET
1-valued direct assignment, chained	<code>name ← src.ePackage.name</code>	LOAD, LOAD, GET(ePackage), GET(name), INVOKE(resolve), SET(name)
1-valued direct assignment, chained-combined	<code>name ← src.ePackage.name + 'impl'</code>	LOAD, LOAD, GET(ePackage), GET(name), PUSH('impl'), INVOKE(+), INVOKE(resolve), SET(name)

First of all, a *static* value can be assigned to a target attribute. The assigned value may not necessarily be of type `String`, as in the examples of the table, but the expression may assign another primitive type, such as an `Integer` (e.g., `upper ← -1`) or an enumeration literal, to an attribute with a corresponding type. The first line of the table gives an example where the attribute `name` of the target element receives the value `'SomeName'`.

Single-Value Static Direct Assignment The corresponding ATL/EMFTVM opcodes LOAD the target object which the *out*-pattern of the rule declares, in the first place. The PUSH opcode receives the static value which is in this example `SomeName` and the opcode INVOKE *resolves* the reference to the loaded target object by looking it up in the (internally maintained) ATL trace. Finally, the SET opcode assigns the pushed value to the `name` attribute of the target element, as declared in the assignment statement.

Single-Value Dynamic Direct Assignment Conversely, an assignment may retrieve the value of the target structural feature from the value of the structural feature of the given source object. The second row presents an example of such direct *dynamic* assignment of a single value. The expression `name ← src.name` assigns the value of a `name` attribute of the source object (`src.name`) to the target `name` whereas the third row exemplifies the *combination* of a static with a dynamic value.

The corresponding opcodes load not only the target object (first LOAD) but also the source object (second LOAD) which needs to be accessed to retrieve the value to be assigned. Then, the source name is GET and resolved. Finally, the source name is set as value of the `name` of the loaded target object.

Similarly, both assignment statements, the static and the dynamic one, can be *combined*. A statement is recognized as conforming to the pattern regardless of which one is stated first, the statically or the dynamically determined value. Thus, the pattern combines the operations of assignment patterns for a single static and dynamic assignment. However, it must be stated,

even though for propagating annotations it does not make a difference how many static values are considered, the complexity of the instruction opcodes increases significantly if several computations based on static and dynamic values are performed before being finally assigned.

Single-Value Dynamic Direct Assignment, Chained Moreover, several transformation languages allow for assigning the value of another source object which is referenced by the matched input object. A pattern which is rather easy to identify, retrieves the dynamic value from a directly linked source object (not of objects accessed by following multiple links) and assigns it without modifications. Row four of Table 7.2 exemplifies such assignment pattern which retrieves the name of the `ePackage` referenced by the source object.

The corresponding instruction opcodes mention a third get: In contrast to the patterns presented before, not the name of the source object is retrieved but the `ePackage` referenced by the source object is retrieved by a `GET` instruction first and its name by a second `GET` instruction. This name is set as the name of the target object.

Single-Value Dynamic Direct Assignment, Chained-Combined Similar to the combined direct assignment, the dynamic value taken from referenced objects can be combined with one or multiple static values. The bottom row of the same table provides an example which computes the assigned value by combining a referenced value with a static one.

Table 7.3: Bytecode instruction opcodes of direct assignment patterns of *multiple* values.

Assignment Type	Example	Instructions
<i>multi-valued direct assignment, dynamic</i>	<code>name ← src.name + src.upperBound</code>	L(tar), L(src), G(name), L(src), G(upperBound), I(+), I(resolve), S(name)
<i>multi-valued direct assignment, combined</i>	<code>name ← src.name + src.upperBound + 'impl'</code>	L(tar), L(src), G(name), L(src), G(upperBound), I(+), PUSH(' .impl'), I(+), I(resolve), S(name)
<i>multi-valued direct assignment, chained</i>	<code>name ← src.ePackage.name + src.name</code>	L(tar), L(src), G(ePackage), G(name), L(src), G(name), I(+), I(resolve), S(name)
<i>multi-valued direct assignment, chained, combined</i>	<code>name ← src.ePackage.name + src.name + 'impl'</code>	L(tar), L(src), G(ePackage), G(name), L(src), G(name), I(+), PUSH(' .impl'), I(+), I(resolve), SET(name)

Overview of Multi-Value Assignments In contrast to single-value assignments, multiple values of the source object can be used to compute the value which should be assigned to a structural feature of the target object. Table 7.3 enumerates patterns which use two values of structural features of the source object to compute the value of the target attribute `name`.

Table 7.3 assumes a similar structure as Table 7.2. Thus, in the left column it mentions a classification of the assignment type, in the middle a concrete example and in the right column the instruction opcodes of the corresponding ATL/EMFTVM model. As opposed to Table 7.2, Table 7.3 abbreviates the opcodes, `LOAD`, `GET`, `INVOKE` and `SET` to their first letter. This table serves as example of how such statements can be constructed.

Multi-Value Dynamic Direct Assignment The number of values used for computing the assigned single value is *not restricted by two* but it may be several ones of the source object or ones that are referenced by the object. However, since these instructions also serve to determine a single value for a single-valued structural feature of the target object, the assignment needs to perform one or several operations to reduce the multiple values to one.

In concrete, as before, two or multiple values of the source object only may be used to compute a single value that is assigned as exemplified in the first row of Table 7.3. The example concatenates the name of the source object with the value stored in the attribute `upperBound` of the source

object and assigns the result as `name` to the target object. The table highlights, the differences with the opcodes used for assigning a single-value which is combined with a static value in blue color. Particularly, a second pair of a `LOAD` and `GET` instruction has to be executed, which loads the source object another time and retrieves the value of the attribute `upperBound`.

Multi-Value Dynamic Direct Assignment, Combined The second row *combines* the two loaded dynamic values with an additional static value. Again, the order of combining the source attributes with the static value is irrelevant. Whenever a pair of loading the source object and a single get instruction appear before the set instruction, the pattern can be classified as multi-value direct assignment which combines several source values.

The difference with the opcodes in the previous row is highlighted in blue color. It requires additional *invocations* of reduction operators which combine the multiple values to one.

Multi-Value Dynamic Direct Assignment, Chained Finally, the third row shows the assignment where the value of one of the structural features is not retrieved from the source object but from an object referenced by the source object (i.e., a chain of references is expanded). As a result, two `GET` opcodes are put on the instruction stack consecutively followed by another `LOAD` instruction to load the source object again to `GET` its name. The first pair of load and get instructions, loads the source object, retrieves the value of the `ePackage` with the first get instruction and the `name` of the referenced package with the second get instruction. Similar actions are performed in the fourth row, where the chained get is combined not only with the direct retrieval but also with a static value.

The examples of the table only show how to compute *String* values of *two* structural features. In general, a multi-value direct assignment for a single-valued structural feature could employ more than two dynamic sources of information as long as these values are reduced to a single one. Similarly, for static values no other restrictions exist neither. The same holds for other primitive types. For example, several Boolean values could be combined in an expression and the resulting value be assigned to a respective attribute.

Collection Assignment In contrast to the patterns explained so far, several transformations allow to assign a list of values to a multi-valued structural feature of a target element. Consequently, a collection of values is assigned to a collection of values.

Example 7.2.3: Assignment of Collections

Listing 7.2.1 presents an example of a multi-valued assignment. Line 16 assigns the union of the elements created for the attributes and the methods of an `EClass` as the `bodyDeclarations` of the corresponding created Java class. The corresponding opcodes, which are placed in the line underneath, are exactly the same as for the assignment which concatenates two string values of two source values. Only the opcode which invokes the `+` operation in case of the string concatenation invokes the `union` operation instead but is still represented by the same opcode `INVOKE`.

Despite the similarity of the opcodes, the collection, consisting of either attributes of some source object or links to other objects, may comprise a *multitude of different annotations*. If it is a direct assignment of only a single multi-valued structural features, the annotation of each element in the collection needs to be looked up and combined in a disjunctive expression. However, if two or multiple collections are flattened to one, the annotation depends on the reduction operator which also may remove elements from one source set such that those elements are not required and their annotations do not have to be satisfied. Due to these ambiguities, the following analysis focuses on recognizing patterns which can be recognized uniquely and which offer clear semantics of how to compute an annotation for target elements based on the source elements.

Further Patterns Finally, the execution control flow can branch due to if-conditions and the possibility to imperatively influence the order of creating target objects by calling rules explicitly. Target objects which are created in a called rule are not present in the trace. Thus, the trace-based propagation cannot annotate them. Furthermore, besides the direct assignment patterns, *indirect* assignments may occur, too. Listing 7.2.1 exemplifies some of these patterns specified in ATL and presents the corresponding instruction opcodes as comments behind or underneath the transformation source code lines. Opcodes which are used to *load* or *get* an object or to *set* a value are abbreviated to their first letter.

```

1 rule Class2ClassDeclaration {
2   from
3     src: Ecore!EClass
4   to
5     tar: Java!ClassDeclaration (
6       name <- if (src.abstract) then -- L, L, G, IFN
7         'Abstract' + src.name.firstToUpper()
8         -- PUSH(Abstract), L, G, I(firstToUpper), I(+), GOTO
9       else
10        src.name -- L, G, I
11      endif, -- S
12
13      superClass <- thisModule.getSupCl(src.eSuperTypes)
14      -- L, GETENVTYPE, L, G(eSuperTypes), INVOKE_STATIC(getSupCl), INVOKE(resolve), S
15
16      bodyDeclarations <- src.eAttributes.union(src.eOperations)
17      -- L, L, G(eAttributes), L, G(eOperations), I(union), I(resolve), S
18    )
19  do {
20    if (not src.eSuperTypes.oclIsUndefined() and src.eSuperTypes.size() > 1)
21      -- left side of 'and': L(src), G, ISNULL, NOT, IFN
22      -- right side of 'and': L, G, INVOKE(size), PUSH, INVOKE(>), GOTO, PUSHF, IFN
23      {
24        for (c in src.eSuperTypes) { -- L(SRC), G, ITERATE, STORE, GETENVTYPE
25          thisModule.assignSuperInterf(src, c); -- L(src), L(c), INVOKE_STATIC, POP
26        } -- ENDITERATE
27      }
28  }
29 }

```

Listing 7.2.1: Example of ATL/EMFTVM transformation rule with complex assignments. The comments represent the corresponding bytecode instruction opcodes.

Firstly, by calling a helper or query method, such as in Line 13, a computation based on given source objects or of their structural features may be performed in another rule or method and assigned to the target element. In the example, a method is invoked which determines the class from which the created class inherits based on the given reference `eSuperTypes`. Even though the statement assigns a single value to the single-valued structural feature `superClass` based on a multi-valued reference, due to this *indirect assignment*, the opcodes which perform the computation are not accessible in the instruction stack of the rule.

Furthermore, it is not guaranteed that such computations are executed linearly. Conditional executions and iterating collections may provoke execution branches, as exemplified for retrieving a name for a class declaration in Lines 6-10. Even though in this case the name of the source class is the decisive attribute used in each branch, this cannot be assumed in general. More complex instructions and different structural features may be used in branches, requiring to decide which structural feature of a branch to prefer for determining an annotation. Particularly, in helper methods it is possible to express conditions and branch the execution based on certain values of the source object. Similarly, an iteration can be performed and used to invoke operations as shown in Line 24 of Listing 7.2.1. Due to these ambiguities, recognizing and determining an annotation based on these complex statements is not regarded in this thesis.

III Observations

From classifying the patterns and mapping a pattern onto bytecode instruction opcodes we draw the following conclusions which allow to identify the patterns.

Start and End of Instruction Firstly, each list of instructions representing *one assignment terminates* with a **SET** opcode and starts with a **LOAD** opcode to retrieve the created target object based on the name defined in the rule. The opcode stores the name of the target object. If a rule invokes another rule by a method call, a **POP** instruction will mark the end of a block instead.

Source and Target Structural Feature Secondly, the knowledge which structural features of which source objects are used to compute the assigned target value represents the key information for propagating their annotations. The **GET** opcodes store the names of the *source* structural features and the **SET** opcode hold the name of the feature to which the value is assigned.

Unique Assignment Patterns Thirdly, it is possible to identify a pattern unambiguously, if it does *not*

- branch
- invoke rules with source values as input
- iterate a collection of values to retrieve a single value
- flatten several lists of values to a single collection
- combine any of these statements.

As explained before, conditional execution, iterating collections and performing operations on them can become arbitrarily complex. Therefore, it may require additional knowledge about the semantics of functions (standard library as well as manually added) to analyze whether and if multiple source values exist, which are relevant to compute the assigned target value in the end. For that reason, the following propagation process does not regard the assignment of collections, or of branches, indirection based on method invocations and combinations thereof. In contrast, our rule analysis, described in Sec. 9.3.2, recognizes *direct* assignments of

- single or multiple static values (first row of Table 7.2)
- single or multiple values of one source object (second row of Table 7.2 and first row of Table 7.3)
- single dynamic values *combined* with static values (third row of Table 7.2 and second row of Table 7.3)
- a single or multiple *chained* value accesses (fourth row of Table 7.2 and third row of Table 7.3).

The following section explains how this information can be used for propagating annotations.

7.2.4 Propagation Process

After having identified assignment patterns and the corresponding bytecode instructions, this section demonstrates how to employ this information for propagation annotations to structural features. As Fig. 7.2.5 illustrates, the multi-variant fine-grained propagation process consists of two elementary steps: the propagation of annotations of objects, based on the trace, and of their structural features, based on analyzing assignment patterns. Chp. 6 explains in detail how the *trace-based propagation* behaves. Therefore, the first part of this section summarizes the main properties of this propagation which serves as starting point for the *bytecode instruction-based propagation* explained in the second part.

I Trace-Based Propagation

To propagate annotations from the source model to the target model, first, the annotations of corresponding objects are propagated by a trace-based propagation as described in Alg. 1.

Result of Propagation: Annotation Mapping for Data Nodes The reused single-variant transformation creates the superimposed target model, the graph H_T and at least a generation-complete trace H_{TR} . Exploiting the trace for propagating annotations results in an annotation

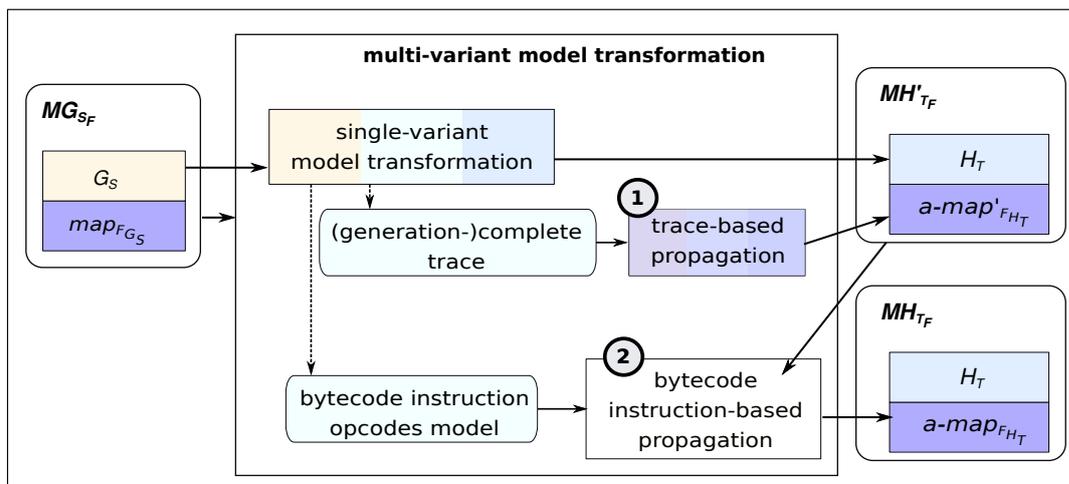


Figure 7.2.5: Overview of bytecode instruction-based analysis.

mapping function $a\text{-map}'_{F_{H_T}}$. In contrast to the previous explanations, the annotation mapping function does not only map annotations onto *graph* nodes and edges, which represent objects and their links but also onto their structural features, which are stored in *data* nodes attached to the graph nodes and edges. Def. 7.2.3 formally defines the function $a\text{-map}_F$ in Sec. 7.2.5.

Annotated Graph Elements In the following, we assume that the trace-based propagation generates a complete mapping function for the graph nodes and edges, representing objects and links, due to processing a generation-complete or complete trace. The data nodes and the edges that link the data node to the created target nodes and edges, receive a default annotation, which is the one of the graph element for which they store the value of one of their structural features. Accordingly, the presence of a data node always depends on the graph element for which it holds data values.

Refining Data Nodes Since the structural features of the source elements may comprise a more restrictive visibility in filtered products, as demonstrated in Sec. 7.2.1, the mapping function for the target graph needs to reflect these annotations, too. Therefore, the multi-variant source and target model, MG_{S_F} and MH'_{T_F} , respectively, are input to the second propagation. This propagation refines the mapping function by analyzing the bytecode instruction opcodes (e.g., persisted in form of a model) and assigning annotations of source data nodes to the corresponding target data nodes.

II Bytecode Instruction-Based Propagation

After the coarse-grained trace-based propagation based on the (generation-)complete trace has been performed, the multi-variant model transformation can execute the fine-grained propagation of annotations based on analyzing the bytecode instructions. Fig. 7.2.6 illustrates the steps performed in the bytecode instruction-based propagation schematically. First, the sections gives a schematic overview of the procedure before diving into more technical details:

Overview The given *bytecode instruction opcode model* is analyzed rule-wise: It starts with *extracting instruction blocks* for each rule. Secondly, it analyzes the blocks and tries to identify patterns. This results in a set of *classified assignment patterns* which store the name of the target structural feature as well the structural features of the source objects that are used. The assignment of a static value only is irrelevant for the further process but is recognized similarly. The following step determines the source and target objects that match the load instruction of a rule. Accordingly, accessing the source and target model is essential to determine the matching

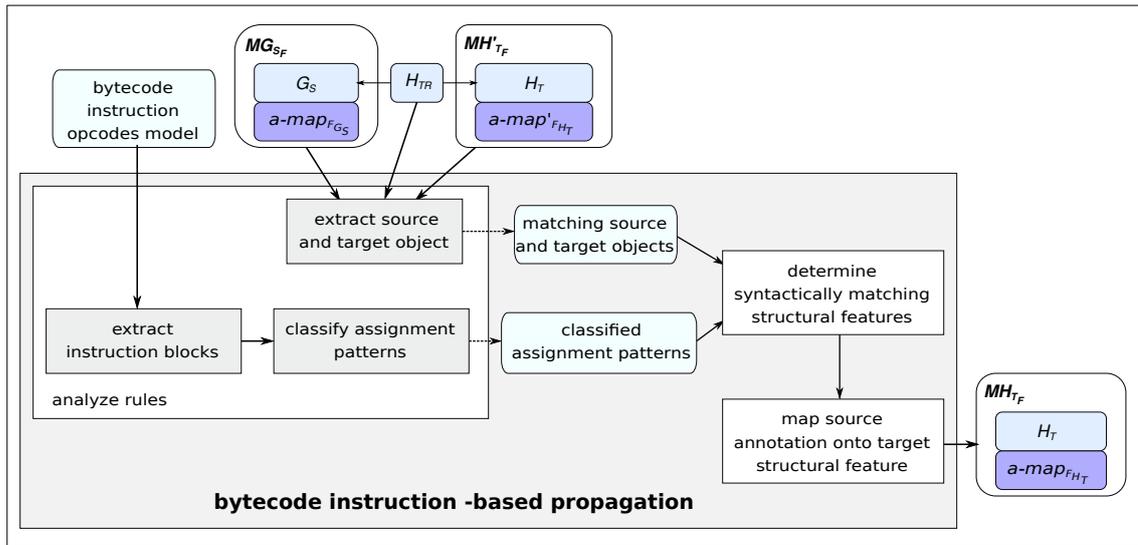


Figure 7.2.6: Steps of bytecode instruction-based propagation.

objects and the values of their structural features. This step can occur already before the pattern recognition or be input from the previous trace-based propagation.

Details In more detail, the bytecode model is input to the propagation mechanism and iterated rule-wise. To illustrate and concretize the procedure, we employ the ATL/EMFTVM execution model as example.

Extraction of Analyzed Rules For each ATL rule recorded in the bytecode model, which matches a source element, an *analyzed rule* is created. The analyzed rule is a data structure which composes the list of instructions that is stored inside the ATL rule into different recognizable assignments. Thus, the analyzed rule stores multiple lists of different assignment patterns which in turn store the key information for propagating annotations: the name of the set-field, which is the structural feature of the target object, and one or multiple names of get-fields which store the field names of the source object. Fig. 9.3.4 summarizes the implementation specifics of the `AnalyzedRule` in more detail. It is important to note is that it is beneficial to build a rule-to-analyzedRule map before (or during) iterating the source model to analyze a rule which matches a source element only once. The following paragraphs explain how different kinds of assignments are extracted from the instruction block associated with a rule.

Instruction Block Extraction To extract the information stored in the `AnalyzedRule`, the entire list of instructions of the rule is iterated. In case of the ATL/EMFTVM this requires to iterate the *applier* codeblock (to-block) and the *postApplier* (do-block) codeblock.

As stated as observation from comparing the instruction opcodes of common assignment patterns in Sec. 7.2.3, instruction blocks (in the applier) which represent an assignment of a value terminate with a `SET` opcode. Accordingly, an instruction *block* terminates with a set instruction and a new block initiates with the opcode stated thereafter. All instructions between the initiation and the set are collected and stored inside the instruction block.

Since the *do-block* in ATL influences the execution order, this part of a rule may invoke other rules without assigning any value. Such instruction block does not end with a `SET` opcode but with a `POP` opcode which needs to be considered when iterating the entire list of instructions to extract assignments. Thus, whenever a `POP` opcode appears, the instructions before are neglected and a new assignment instruction block is tried to be extracted from the opcodes following the `POP` opcode. The do-block is considered because it is possible to place assignments in this section of a rule, too.

Assignment Pattern Recognition Next, the instruction blocks need to be analyzed. Based on the common structure of instruction sequences observed by mapping different assignment statements onto the instruction opcode in Sec. 7.2.3, this step tries to recognize patterns in the instruction blocks.

A direct static and dynamic assignment of a single value can be recognized in a straightforward way. They consist of a fixed number of instructions (4 and 5, respectively) and of a fixed sequence of opcodes, as exemplified in row one and two of Table 7.2. While the corresponding recognition algorithm adds a direct dynamic assignment to a corresponding list in the analyzed rule, a static assignment is neglected. The analyzed rule records such assignment pattern as **SingleValueAssignment**, as shown in the model of Fig. 9.3.4. A static assignment of a single value cannot contribute to propagate annotations of a source structural feature because it does not state any source feature. Similarly, a **ChainedAssignment** is recognized whenever a sequence of two or more GET instructions occurs without interruptions of another opcode in the block of instructions. If the static assignment is *combined* with a dynamic assignment of a single source value or a chain thereof, it will be inserted as a **CombinedAssignment**. Such assignment holds a list of single GET instructions but can also consider chains of GET fields which are represented as lists. The identification algorithm recognizes such assignment whenever a single get instruction opcode (no other directly thereafter) follows two load opcodes. Opcodes in between which do not provoke a branch or iteration of the execution are ignored.

Whenever a branch instruction, such as an **ITERATE** or **IF/IFN** instruction, occurs in the instruction block, the algorithm categorizes the assignment as **OtherStatement** because it may involve diverging execution branches which are provoked, for example, by diverging values of a structural feature. Since both can be combined almost arbitrarily, their analysis requires more specific techniques and is left out of the scope of this thesis.

Propagation Algorithm After the patterns have been identified per rule, the propagation algorithm iterates the source nodes and executes the following steps: Due to its complexity, we state the steps of the propagation process in natural language for easier comprehension:

1. for each rule: create an analyzed rule and store it in a mapping function: $\rho\text{To}\alpha : \rho \rightarrow \alpha$
2. for each source object or link el
 - 2.1. retrieve transformation rule ρ , which records el as source element, from the execution model
 - 2.2. retrieve analyzed rule α by looking up ρ in the map $\rho\text{To}\alpha$
 - 2.3. retrieve the trace node n_{TR} , which records el as source element, from H_{TR}
 - 2.4. retrieve created target objects from trace node n_{TR} and store them in TRG
 - 2.5. for each created target object $trg \in TRG$
 - 2.5.1 retrieve its structural elements and store them in $DNODES_t$
 - 2.5.2 for each structural feature $f_t \in DNODES_t$
 - 2.5.2a search a pattern in α which records f_t in the set-field and loads trg
 - 2.5.2b if **no** pattern matches, continue with next target feature, i.e., **goto** 2.5.2; otherwise:
 - 2.5.2c for each get-field in the pattern search an equally named structural feature in the source object and store them in $DNODES_s$
 - 2.5.2d assign mapping to f_t : combine the annotation of the object trg with the annotation of the elements in $DNODES_s$ in a conjunction

Accordingly, the source model is iterated and we assume that a rule matches only one source element and creates a set of target elements (nodes and edges). Furthermore, no target element can be targeted by two distinct rules, i.e., the property for STT-graphs that each target element possesses exactly one incoming edge (c.f., Equation 6.13) still holds. As such, all target elements can

be retrieved from the trace model given the source object. Alternatively, based on the information in the execution model, corresponding target objects might be retrieved similarly.

Next, for each created target element in *TRG*, the structural features are determined in step 2.5.1. Then, a pattern in the analyzed rule α has to be found which assigns a value to the currently processed structural feature. For solving this task, the algorithm iterates the list *DNODES_t* and searches a pattern in the corresponding analyzed rule α , the set-field of which matches the name of the currently processed structural features. If the algorithm finds a match, it depends on the concrete pattern which exact steps to perform in step 2.5.2c:

SingleValueAssignments contain only one get-field which is used for searching an equally named structural feature in the source object. The algorithm will add the corresponding single data node to *DNODES_s*, to map its annotation onto the structural feature f_t .

ChainedGets contain a list of get-fields. Thus, the reference to the target object needs to be followed as long until the final **GET** instruction. The annotation mapped onto each object on this way needs to be combined in a conjunction with the annotation of the last attribute or reference to ensure the visibility of the elements along the path when filtering the multi-variant model.

CombinedAssignment may contain multiple structural features depending on the number of get-fields. Thus, the algorithm stores the matching structural feature in the set of source data nodes *DNODES_s*, the annotations of which are combined in a conjunction. Furthermore, if a chain occurs, the algorithm computes the annotation while following the references and combines them in a conjunction with the remaining annotations.

To this end, the computed annotations consist of the annotation of the source object el , which is also present on the target objects due to the trace-based propagation, and a conjunction of the annotations mapped onto the elements matching the get-fields. As a consequence, after having iterated the entire source model, each structural feature of the target model possesses a mode specific annotation, if its counterpart(s) in the source model possess(es) a more specific annotation than the graph element, too.

III Properties

According to the algorithm for analyzing bytecode instruction, informally described in the preceding parts of this section, the transformation must conform the following properties.

Property 7.2.1: Single-Value Assignment

The rules have to assign only a single value.

So far, the algorithm does not handle the assignment of collections of values to collections of values. In contrast, the algorithm can compute the single value based on multiple values which are reduced to one. If an assignment computes values for multiple data nodes, the pattern will be ignored.

Property 7.2.2: Linear Execution Control Flow

The execution control flow must not branch, particularly, not on collections which should be iterated.

Due to ambiguities and potentially conditional execution, branches are not covered and assignments are hard to determine unambiguously if branches and iterations are nested arbitrarily. Therefore, this behavior is not extracted as assignment pattern. Note: This property also prohibits the invocation of helper methods which is made explicit in Prop. 7.2.3.

Property 7.2.3: Self-Contained Transformation

The transformation must not delegate computations to other engines.

Prop. 7.2.3 requires that any computation necessary to construct the target model must be part of the transformation specification or accessible from it in a representation that resides at the same level of abstraction as the analyzed bytecode. On the one hand, this property forbids to employ computations which are implemented in another transformation language or a general purpose language because the bytecode model may deviate from the one used here. On the other hand, the property does not forbid helper-functions or modularity of transformations in general, even though they are not supported yet.

Please note: We do not explicitly forbid the usage of called and lazy rules in ATL/EMFTVM. If target objects remain without an annotation after propagating the annotations based on the generation-complete trace, we employ the container completion strategy, as described in Alg. 4, to assign annotations to the target elements missing one. Thus, before analyzing the rules for fine-grained mappings, it is guaranteed that the source and target mapping representations are completely annotated.

If these properties hold, it is possible to analyze the transformation and to retrieve annotations for structural features of the target model. Due to the finite sets that are iterated and missing recursion, it is guaranteed that the propagation algorithm terminates.

7.2.5 Foundations

So far, the formalism for trace-based propagation of annotations employs a *typed graph (transformations)* which allows to annotate nodes and edges, representing objects and links between them. This chapter, however, assumes that the structural features of graph nodes can be annotated, too. To express this in the formalism, we extend the set of elements in the graph such that different types of nodes represent objects and their features. Furthermore, the mapping annotation function must be able to map annotations onto the nodes and edges which represent a structural feature. The extension of the ordered graphs used so far requires to add nodes and edges which represent the structural features of an object. We add elements which extend the graph typed over T_N and T_E to an attributed typed graph, also denoted as E-graphs [EPT04].

E-Graphs In the first place, we extend the set of *graph nodes* N with a set of *data nodes* D_N . A data node d_n is typed and uniquely associated with a graph node to store the concrete value of an attribute. Similarly, we add a set of *node attribute edges* E_{NA} which associate the graph node with a value stored in the data node. Such edge origins from a graph node in N and targets a data node in D_N , such that $s_{NA} : E_{NA} \rightarrow N$ and $t_{NA} : E_{NA} \rightarrow D_N$ are the corresponding source and target functions.

Finally, a set of *edge attribute edges* E_{EA} incorporates edges which origin from a graph edge and assign a value by the means of a data node to the edge.

Definition 7.2.1: E-Graph

An E-graph EG with $EG = (N, E, D_N, E_{EA}, E_{NA}, (s_i, t_i)_{i \in \{G, EA, NA\}}, l_j)_{j \in \{N, E, D_N, NA, EA\}}$ is typed over type sets $T_N, T_E, T_{D_N}, T_{NA}, T_{EA}$ and comprises the following elements

- finite sets of graph nodes and graph edges, N and E
- finite sets of data nodes D_N
- finite sets of edges E_{NA} and E_{EA} which connect a graph node and a graph edge with a data node, respectively.
- source and target functions

between graph nodes: $s : E \rightarrow N$ and $t : E \rightarrow N$

between a graph node and data node: $s_{NA} : E_{NA} \rightarrow N$ and $t_{NA} : E_{NA} \rightarrow D_N$
 between a graph edge and a data node $s_{EA} : E_{EA} \rightarrow E$ and $t_{EA} : E_{EA} \rightarrow D_N$

- type labeling functions for graph elements:

$$l_N : N \rightarrow T_N, l_E : E \rightarrow T_E$$

The set of graph elements is defined as $EL = N \dot{\cup} E$. The entirety of all elements in the attributed graph is denoted as $AEL = EL \dot{\cup} D_N \dot{\cup} E_{NA} \dot{\cup} E_{EA}$.

For propagating annotations of source data nodes to target data nodes, the function *data* allows to determine the data nodes for a given graph element:

Definition 7.2.2: Data Node Function

Let EL be the graph elements of an E -Graph EG . The function $data : EL \rightarrow \mathcal{P}(D_N)$ takes a graph element and returns all data nodes with an incoming node attribute or edge attribute edge originating from the given element:

$$data(el) = \begin{cases} \{d_n \in D_n, e \in E_{NA} \mid s_{NA}(e) = el \wedge t_{NA}(e) = d_n\} & el \in N \\ \{d_n \in D_n, e \in E_{EA} \mid s_{EA}(e) = el \wedge t_{EA}(e) = d_n\} & el \in E \end{cases} \quad (7.1)$$

Moreover, the third adaptation modifies the mapping annotation function map_F (c.f., Def. 6.2.4) such that it can map an annotation onto the data nodes in the set D_N :

Definition 7.2.3: Attribute Mapping Annotation Function

Let F and A_F be the set of features and annotations over features, respectively, and let $ELD = EL \dot{\cup} D_N$ be the set of elements in the E -graph EG which can be annotated. The attribute mapping annotation function $a - map_p_F : ELD \rightarrow A_F$ is a total function assigning either an annotation or the empty element ϵ to each element in G .

Propagation Algorithm Based on the three extensions to the graph formalism for the trace-based propagation, Alg. 2 notes the steps of propagating the annotations of data nodes formally according to the informal descriptions of Sec. 7.2.4.

The propagation iterates the set of source graph nodes in Line 15 and determines a rule which records the graph node as source element of its left hand-side source graph in the following line. The algorithm further invokes the function `ANALYZEORLOOKUP`, which extracts the corresponding *analyzed rule* from a map or computes the analysis and stores it in the map. Furthermore, the algorithm retrieves the trace node which records the processed source element in its source elements. Since we assume generation-complete traces, the source element must be present.

Then, the algorithm retrieves the created target elements from the trace node and iterates them. For each traced target node, the algorithm determines the set of its structural features $DNODES_t$ by employing the function *data* and iterates this set. The function `MATCHINALPHA` in Line 24 tries to find an assignment which states the name of the currently processed data node d_t in its set-field. If it finds a matching pattern, it will store the pattern in the variable *assign-block* and use it to retrieve get-fields which match the data nodes of the source node $DNODES_s$.

Depending on the assignment type, the function `GetAnnotationFromMatch` combines the source annotations, as described in Sec. 7.2.5, II. If a single direct dynamic assignment is detected, the annotation of the source data node will be mapped onto the target data node without modifications. If multiple dynamic values are recognized in a `CombinedAssignment`, all corresponding source annotations will be combined in a conjunction. Similarly, the function combines the annotations on the way to the source attribute in a conjunction in case a `ChainedAssignment` is detected. Finally, the resulting annotation a_F is combined with the annotation of the target element in a

Algorithm 2 Propagation of annotations of structural features.

```

1: procedure PROPAGATE( $EH, a - map_{F_S}, P, a - map_{F_T}$ )
2:   in  $EH = S \leftarrow TR \rightarrow T$        $\triangleright$  attributed target STT E-graph, derived from  $S \leftarrow \emptyset \rightarrow \emptyset$ 
3:   in  $a - map_{F_S} : ELD_S \rightarrow A_F$        $\triangleright$  Annotation function for the source graph  $S$ 
4:   in  $P$                                            $\triangleright$  The set of transformation rules
5:   inout  $a - map_{F_T} : ELD_T \rightarrow A_F$    $\triangleright$  Annotation function for the target graph  $T$  which is
      already complete
6:
7:   var  $n_S \in N_S$                                  $\triangleright$  The current source node to be processed
8:   var  $n_{TR} \in N_{TR}$                              $\triangleright$  The trace node recording  $n_S$  as source node
9:   var  $d_T \in D_{N_T}$                                  $\triangleright$  The target data node to be annotated
10:  var  $a_F \in A_F$                                      $\triangleright$  Annotation retrieved from matching source elements
11:  var  $\rho \in P$                                          $\triangleright$  The currently matching transformation rule
12:  var  $\alpha_\rho$                                            $\triangleright$  The analyzed rule for  $\rho$ 
13:  var  $assign\_block$      $\triangleright$  Block of opcodes in  $\alpha_\rho$ , which stores the set-field that matches  $d_T$ 
14:
15:  for all  $n_S \in N_S$  do
16:     $\rho := \{(L, R) \in P \mid n_S \in L_S\}$        $\triangleright$  Retrieve rule which records  $n_S$  as source element
17:     $\alpha_\rho := \text{ANALYZEORLOOKUP}(\rho)$ 
18:     $\triangleright$  Retrieve trace node and target elements recorded in trace node
19:     $n_{TR} := \{tr \in N_{TR} \mid tr \xrightarrow{src} n_S\}$ 
20:     $TRG := \{te \in EL_T \mid n_{TR} \xrightarrow{trg} te\}$ 
21:    for all  $t \in TRG$  do                                 $\triangleright$  Process all target elements
22:       $DNODES_t := \text{data}(t)$                              $\triangleright$  get structural features of target element
23:      for all  $d_T \in DNODES_t$  do                         $\triangleright$  process structural features
24:         $assign\_block := \text{MATCHINALPHA}(\alpha_\rho, d_T)$      $\triangleright$  block with matching set-field
25:        if  $assign\_block \neq \epsilon$  then
26:           $a_F := \text{GETANNOTATIONFROMMATCH}(assign\_block, DNODES_s, map_{F_S})$   $\triangleright$ 
          computed annotation based on assignment type and matching get-fields
27:           $map_{F_T}(d_T) := a_F \wedge map_{F_T}(t)$            $\triangleright$  Annotate the target data node
28:        end if
29:      end for
30:    end for
31:  end for
32: end procedure

```

conjunction which ensures that the data nodes can only be present when the corresponding graph element is present in a configured variant.

7.2.6 Discussion

To sum it up, this section discusses the benefits and challenges of analyzing a bytecode model in order to propagate annotations from structural features of source elements to target elements.

At first, the section illuminates some critical and beneficial aspects as well as points to consider when applying a bytecode analysis whereas the second part of the section illuminates related work and discusses future potentials.

Computational Cost First of all, a developer may consider the computational cost when deliberating if the additional annotation refinement pays off. Alg. 2, which propagates the fine-grained annotations, performs several iterations over relevant data nodes to compute and assign the annotations.

Firstly, the bytecode needs to be analyzed rule-wise. This is not explicitly shown in the algorithm but part of the functionality of the function `CANALYZEORLOOKUP` and informally explained in Sec. 7.2.4. Secondly, the algorithm iterates the finite set of source nodes and for each source node

looks up a matching trace element and the corresponding analyzed transformation rule. Furthermore, the algorithm iterates the recorded target elements and for each of them the data nodes, as well as the data nodes of the source element to find matching set- and get-fields, respectively. Due to the complexity of the computation steps, which require several iterations of the graph elements and their data nodes together with mapping look-ups on top of the trace-based propagation, we argue that the solution pays off in two cases: On the one hand, if the annotations of a significant number⁴ of structural features are more specific than those mapped onto their respective objects, an automated approach to propagate their annotations is beneficial to decrease manual efforts and human error. On the other hand, if no trace is available, the potential to analyze the bytecode remains as one solution to reconstruct the entire mapping of source elements including their structural features.

Pattern Derivation Sec. 7.2.3 offers an overview of common assignment patterns in model transformations. The following remarks need to be considered:

Firstly, the relevant information for the purposes of this work is which source elements are used to create which target elements. *How* target elements are created, i.e., which actions are performed in between is not necessarily relevant. Accordingly, we focused to describe patterns which we can identify clearly and which provide the key information namely the source elements and the target element. As shortly discussed in the section, due to ambiguities and a necessary deeper analysis particularly when collections are assigned, we refrain from exploiting them in our proof of concept presented in Sec. 9.3.2.

Secondly, we have ATL transformation rules to demonstrate, exemplify and classify different kinds of assignments. For presenting the opcodes we employed the EMFTVM bytecode model. Although we are aware of similar patterns in other declarative languages, such as QVT (Relational and operational mappings), it may require further investigation to which extent these patterns occur in different transformation languages, particularly, if the transformation does not realize the declarative paradigm. Accordingly, we categorize this approach as language-specific even though the ATL transformation specification itself is not analyzed.

Benefits On its upside, similar to the discussion of a transformation based on a generation-complete trace in Sec. 7.1, the analysis-based propagation ensures that the target mapping is *completely* annotated, including its structural features at every time after the initial trace-based propagation. If patterns are identified which assign target values (i.e., the transformation adheres to the properties summarized in the third part of Sec. 7.2.4), the mappings of structural features will be refined if necessary. This is particularly relevant if the structural features can assume different values in different configurations due to different annotations.

Furthermore, mapping annotations to the structural features of objects in product line models is a highly fine-grained way of annotating elements but not supported by many MDPLE approach. It aims to allow for alternative values depending on the presence or absence of features. Particularly, when unconstrained variability is supported, for instance as implemented in single-variant editing tools, such as SuperMod [SW16] or the VASG in the projectional editing tool [Reu+20], allowing for different values of a structural feature may be necessary. However, in this thesis we assume constrained variability (Prop. 4.1.2). As a consequence, if a more specific annotation a_r is assigned to the value of a structural feature of an object than to the object (annotation: a_o , with $a_r \Rightarrow a_o$), the structural feature still is present in a filtered product even if a_r is not satisfied by the feature configuration. Due to constrained variability, the structural feature will assume its default value if the annotated data node does not pass the filter.

Finally, the potential of analyzing bytecode may pay off if no information about corresponding objects in form of a trace is available. A rule analysis can determine source and target objects which match the declared source and target types. If execution branches are discovered as well in a static analysis (e.g., by building control and data flow graphs) the mappings of assignments occurring in branches may also be determined. However, it must be noted that this would be

⁴ For instance, if each graph element possesses a data node with a more fine-grained mapping, an automated propagation may pay off to reduce the manual refinement efforts.

an expensive solution which is still specific to a transformation language and requires a complete disassembly of the source code and domain models which support unconstrained variability.

Related Work Static analysis of (ATL) transformations has been implemented and performed by Cuadrado et al. [CGL17; CGL18]. This work, however, aims at a different goal: The static analysis serves to enhance the correctness of an ATL transformation specification by analyzing it each time it is saved. Then, the analyzer searches, for example, for typing errors. In addition, the analysis is exploited to improve the performance of the transformation execution engine, e.g., through parallelization. Consequently, the analyzer does not provide the pinpointed information about corresponding source and target features but would help to determine whether all contributing objects are bound. Furthermore, the analyzer does not operate on the execution model persisted by the ATL/EMFTVM but on the operand stack created by the ATL default virtual machine. In contrast, we employ the EMFTVM because it stores an accessible generation-complete trace as well as the corresponding execution model, which we analyze instead. Finally, our analysis serves to propagate annotations from source elements to target elements and is, therefore, trimmed for this purpose.

In Sec. 4.3.3, we classify the *lifting* approach [Sal+14] as white-box solution. However, lifting does *not analyze* the transformation nor any of its artifacts but changes the semantics of the execution engine. Accordingly, the approach works independently of a specific transformation specification but also requires access and adaptation of a single-variant model transformation engine to become capable to transfer variability information. In contrast, analyzing bytecode instructions neither regards the transformation specification itself (e.g., by parsing it) nor requires to change the execution semantics. Similar to gray-box propagation, the approach analyses an artifact created for specifying the transformation which is, in contrast to a trace, already present before the transformation execution. Since our analysis focuses on the bytecode model of the ATL/EMFTVM, it is trimmed to the analysis of exactly that model and, thus, language-specific. Nonetheless, the assignment patterns itself are *independent* of a transformation language.

Outlook Similar to abstracting trace-based propagation, a bytecode instruction-based abstraction could be used to analyze several different kinds of bytecode, such as also the Java bytecode model [Rhe+18]. Accordingly, the bytecode instructions of several compilers could be compared and used to derive a common model from which also common assignment patterns could be combined. To further respect branches and iterations instead of extracting only assignment patterns, control and data flow graphs could be extracted by employing methods from static source code analysis. If this white-box analysis was used to create the target model, the boundaries of constrained variability might have to be extended and a multi-variant model where elements (e.g., structural features) may instantiate different values could be created, given an according representation is available (for instance, similar to the hidden superimposed model in SuperMod or the VASG in the projectional editor).

7.3 Incomplete Trace Information

Besides model transformation engines which persist complete and generation-complete traces, transformation approaches exist which pertain only records of the main pivotal source and target element per rule applications (i.e., 1:1 mappings) in *incomplete traces*. Triple graph grammars and similar approaches [Sch94; LAS14; Buc18], represent examples of approaches which record incomplete traces. This section describes how to maintain the situation when *incomplete trace* information is available. In this situation, we propose to perform the annotation propagation based on the information present in the trace in the first step. A second step completes the mapping information of the target model based on heuristics which exploit the hierarchical structure of the target models.

For introducing the situation provoked by incomplete traces, Sec. 7.3.1 illustrates the problems which may occur, due to the incomplete trace information while Sec. 7.3.2 explains how the foundation of transformation introduced in Chp. 6 have to be adapted in order to explain the alternative computation of a completely annotated target model in Sec. 7.3.3. To close this section, Sec. 7.3.4 discusses the impact of computing missing annotations with respect to the accuracy and the time efforts at the conceptual level.

7.3.1 Problem Statement

This section describes the situation which will occur if annotations are propagated based on *incomplete* trace information. It starts with presenting an example and draws conclusions from the example afterwards.

Example 7.3.1: Effects of Annotation Propagation Based on *Incomplete* Trace

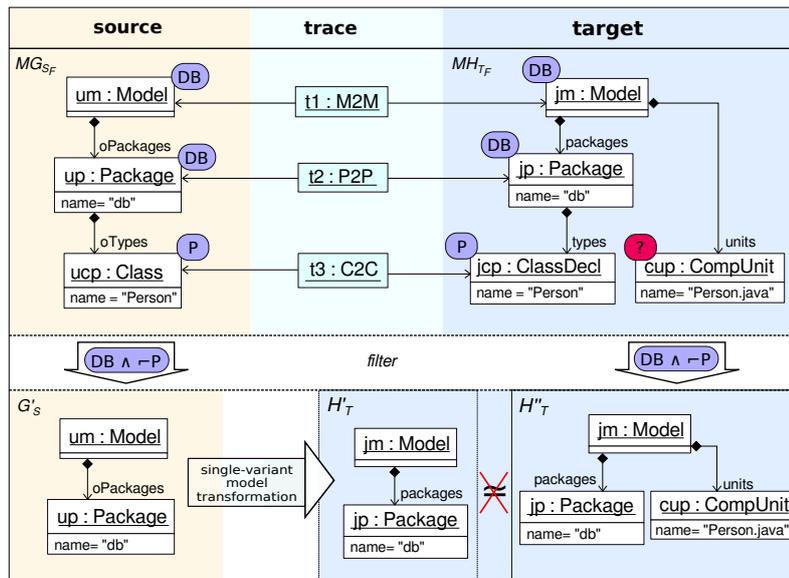
Fig. 7.3.1 illustrates an example which employs an *incomplete* trace to propagate annotations. Similar to the previous examples in this thesis, the source UML class model encompasses database contents which are transformed into a corresponding Java model.

The source package `db` contains the single class `Person` which is annotated with the optional feature `P`. The source edges miss an annotation. Therefore, the filter needs to employ a strategy to handle missing annotations. In this example, the model filter integrates elements missing an annotation in each product. Furthermore, this filter is capable to guarantee *referential integrity*: if a link passes the filter, both of its ends have to be present, otherwise the filter will remove the link.

Similar as in the previous examples, the transformation specification turns a UML class into a Java class declaration and a compilation unit which are stored in the corresponding Java package and model, respectively. In contrast to the previous examples, which assumed the availability of a (generation-)complete trace, here only an *incomplete* trace is available. Accordingly, the trace elements record the single pivot elements of the transformation rules. For instance, in case of the `C2C` rule, the trace element relates the UML class with the Java class declaration. Consequently, the propagation algorithm can only map the annotation of the single source element onto the single target element causing the compilation unit to remain without annotation.

The bottom of Fig. 7.3.1 shows one step of evaluating the commutativity criterion which filters the annotated multi-variant models by the feature configuration selecting the feature `DB` and deselecting the optional feature `P`. The left hand-side depicts the filtered source model G'_S , which comprises the UML model and package. Transforming G'_S with the single-variant model transformation results in the single-variant model H'_T which contains the corresponding Java package and model.

In contrast, filtering the multi-variant target model MH_{T_F} by the same feature configuration derives the single-variant target model H''_T which consists of the package but also of the compilation unit for the class `Person`. Due to the missing annotation, the filter assumes the respective element to be visible in each variant. Since H'_T and H''_T are not equivalent (up to isomorphism), the multi-variant model transformation violates commutativity.


 Figure 7.3.1: Commutativity violation due to *incomplete* trace information.

This example demonstrates that the annotations propagated to the multi-variant target model based on incomplete trace information do not suffice to guarantee commutativity and that the missing information in the incomplete trace causes this effect.

Influence of Model Filter As the example needs to assume the behavior of the filter in sight of missing annotations, we discuss the effect of the filter before drawing conclusions from the example:

It must be noted that not only the kind of trace may make a difference for satisfying the commutativity criterion but also the definition of the filter function: As explained in Sec. 7.3.1, III, a flat filter does not make assumptions about relationships between model elements but considers the annotation mapped onto an element to determine whether the element is integrated in the product derived for a feature configuration. Def. 6.2.5 defines this kind of filter as *flat filter function* while assuming that an annotations is mapped onto all elements.

However, filters have to react to situations in which elements miss annotations, too. Flat filters can either integrate each element missing an annotation (assuming an implicit `true` as annotation) or remove each element missing an annotation (assuming an implicit `false` as annotation) in the derivation. However, assuming the annotation `false` would mean the element is not present in any of the configured products which would make the element superfluous in the multi-variant model. Thus, the unconditional removal of an element can hardly represent a valuable alternative. For that reason, we consider flat model filters which integrate each element without annotation in the derived product in the example and in the sequel.

In contrast, a hierarchical filter regards relationships between elements of the multi-variant model. Based on the hierarchical structure formed in several model types, particularly in models conforming to the Ecore metamodel, the presence of existence dependencies can be assumed. As a consequence, hierarchical filters may override or suppress the integration of a model element in the derived variant based on the given annotations and the hierarchical structure of the models, which are assumed to form a spanning containment tree. In this example, however, the hierarchical filter would not make a difference because the compilation unit as a child element is filtered and the well-formedness is not affected by removing it. If the situation was vice versa, the compilation unit was kept in the derived variant and the model was removed, the hierarchical filter would prevent the situation by integrating the Java Model in the derived variant, too.

In general, filters for arbitrary models cannot assume dependencies inside the models generically and, thus, do not allow for propagating selection states as hierarchical filters do. Therefore, in the following discussions we assume a flat filter and only refer to the capabilities of a hierarchical

filter explicitly, if it is relevant. Furthermore, if annotations can be assigned only to objects of the models, referential integrity depends on the capabilities of the flat filter. In *general*, there are two viable solutions to handle edges without annotations: The filter may either integrate such edges in each variant unconditionally, assuming an implicit `true` as annotation, or it may integrate the edges only if both ends are integrated in the variant. Removing such edges (i.e., assuming `false` as annotation) would violate referential integrity because then only objects would be present in the derived product without links connecting them.

Consequences: Shortcomings Caused by Missing Annotations The example reveals that an incomplete trace as input to the trace-based propagation algorithm results in a partially annotated model. However, in a partially annotated model elements missing an annotation may be retained in or removed from too many configured products by mistake due to the missing mapping information and due to different model filter capabilities.

Accordingly, we identify at least the following shortcomings and problems caused by utilizing traces with insufficient information about corresponding elements in source and target model:

- **uncertainty:** in general, it is unclear whether an element missing an annotation is *included in or excluded from* a derived model. The presence depends on the capabilities and the configuration of the employed model filter.
- **increased resource consumption:** including too many elements in the derived *target* model which are not present in the corresponding derived source model may consume an increased and unnecessary amount of resources.
- **security threat:** integrating (sensitive) data, which is not intended to the customer according to its configuration, in the product may leak confidential or security-relevant information or functionality (although it may be hidden or dead but is delivered, nonetheless).
- **misbehavior and inconsistency:** excluding too many or the wrong elements may cause dangling references, misbehavior and inconsistent models which, in the end, may provoke non-functional source code to be derived.

For that reason, it is essential to assign annotations to all target elements which can be annotated. Only then, no (or the least) assumptions have to be made and misbehavior resulting from missing annotations can be avoided. Consequently, the subsequent sections discuss strategies to determine missing annotations in partially annotated models.

7.3.2 Foundations

In contrast to the total mapping function produced by trace-based propagation based on a complete trace, the propagation based on an incomplete trace creates a partial mapping function map_{pF} . The following definitions adjust the foundations provided in Chp. 6 to respect the conditions introduced by the missing correspondence information.

Partially Annotated Multi-Variant Graph Firstly, multi-variant graphs as defined so far comprise a *total* mapping annotation function (Def. 6.2.4). Therefore, Def. 7.3.1 declares a *partial* mapping annotation function:

Definition 7.3.1: Partial Mapping Annotation Function

Let F and A_F be the set of features and annotations over features, respectively, and let $EL = N \dot{\cup} E$ be the set of elements in a graph G . The *partial mapping annotation function* $map_{pF} : EL \rightarrow A_F \cup \{\epsilon\}$ is a *total function* assigning either an annotation or the empty element ϵ to each element in G .

We note the set of annotations including the empty element as A_F^0 , such that $A_F^0 = A_F \cup \{\epsilon\}$.

The definition uses the empty element ϵ to describe a missing annotation for an element in the graph. Furthermore, in contrast to the total mapping function map_F , referential integrity cannot be ensured by this function.

On the whole, a *partially annotated multi-variant model* is a graph with a partial mapping annotation function:

Definition 7.3.2: Partially Annotated Multi-Variant Graph

Given a set of features F and annotations A_F^0 , a partially annotated multi-variant graph is a pair $MG_{pF} = (G, map_{pF})$, where $G = (N, E, l_N, l_E, s, t)$ is a graph comprising the disjoint set of elements $EL = N \dot{\cup} E$, and $map_{pF} : EL \rightarrow A_F^0$ is a partial mapping annotation function assigning a potentially empty annotation to each element of G .

Filter Depending on the used tool, the visibility evaluation function (c.f., Def. 7.3.3) used by the filter function (Def. 6.2.5) may interpret an empty annotation differently.

If the filter includes all elements missing an annotation in the configured product, the evaluation function can be refined in the following way:

Definition 7.3.3: Positive Visibility Evaluation Function

Let F be a set of features, FC_F the set of feature configurations and A_F be the set of annotations over F . A positive visibility evaluation function over F is a function $v_F : A_F^0 \times FC_F \rightarrow \mathbb{B}$, where $\mathbb{B} = \{true, false\}$ denotes the set of Boolean values. v_F guarantees the following property:

$$v_F(\epsilon, fc_F) = true \quad (7.2)$$

$$v_F(a_F, fc_F) = true \Leftrightarrow fc_F \Rightarrow a_F \quad (7.3)$$

Consequently, regardless of the feature configuration, a missing annotation means that the element is visible and, thus, integrated in the resulting product.

Hierarchical Graph Moreover, the following algorithms assume that the graphs form a strict containment hierarchy. A node can only exist if its parent node exists in the graph or the node is the root of the containment hierarchy. Nevertheless, also cross-references inside the tree may occur which are not decisive for the containment hierarchy and, thus, irrelevant for the existence relationship. For extending the graph formalism to express a hierarchy, we employ the function $c : E \rightarrow \mathbb{B}$, which states whether a given edge is a containment relationship (*true*) or not (*false*). The function s applied to a containment edge determines the *container* and the function t determines the *contained node*.

Definition 7.3.4: Hierarchical Graph

Let T_N and T_E be the finite sets of node types and edge types, respectively. A hierarchical graph over T_N and T_E is a tuple $G = (N, E, l_N, l_E, s, t, c, root)$, where

- N is a finite set of nodes,
- E is a finite set of edges, where $(N \cap E = \emptyset)$,
- $l_N : N \rightarrow T_N$, $l_E : E \rightarrow T_E$ are a node and edge labeling function, respectively,
- $s : E \rightarrow EL$, $t : E \rightarrow EL$ are a source and target function, respectively,
- $c : E \rightarrow \mathbb{B}$ is a Boolean container function and
- $root \in N$ is the root node of the graph.

The container n_p of a node $n \in (N \setminus \{root\})$ must be unique, must not reference itself, and the root must not possess a container which the following three properties ensure:

$$\begin{aligned} \forall e_1, e_2 \in E: \quad & c(e_1) = c(e_2) = true \wedge t(e_1) = t(e_2) = n \\ & \Rightarrow e_1 = e_2 \end{aligned} \quad (7.4)$$

$$\forall n \in N \setminus \{root\}: \quad \nexists e \in E: s(e) = n \wedge t(e) = n \quad (7.5)$$

$$\forall e \in E: \quad c(e) = true \Rightarrow t(e) \neq root \quad (7.6)$$

Accordingly, the definition of hierarchical graphs models a containment hierarchy where the container of a node is unique.

Tree Operations For determining missing annotations, the respective strategies exploit the hierarchy of nodes. Functions delivering parent in children nodes in the graph can be defined in the following ways. Def. 7.3.5 initiates with describing how the *parent* of one node is determined.

Definition 7.3.5: Parent Node Function

Let G be a graph typed over T_N and T_E and $EL = N \dot{\cup} E$ be the set of its elements. The partial function $parent_E: N \rightarrow N \cup \{\epsilon\}$ returns the unique parent node n_p for a given node n or nothing if it is the root, such that

$$parent_E(n) = \begin{cases} n_p & \Leftrightarrow \exists e \in E: c(e) = true \wedge s(e) = n_p \wedge t(e) = n \\ \epsilon & n = root \end{cases} \quad (7.7)$$

The transitive closure over all parents of a node is denoted as $parent_E^+ \subseteq N \times N$.

Ex. 7.3.2 provides an example of applying the parent function.

Example 7.3.2: Parent (*Container*) Annotations

In the example, depicted in Fig. 7.3.1, the container of the UML package up is the UML model. Consequently, $parent_E(up) = um$. Similarly, the parent of the UML class ucp is the UML package and thus, $parent_E(ucp) = up$. In contrast, since um is the root node, the parent function will not return a node for this node as input: $parent_E(um) = \epsilon$.

Accordingly, in a spanning containment tree the parent node n_p of a node n is unique and the only one that is referenced as source node with an outgoing edge labeled as container from n .

In addition, we define the set of children nodes of a node n as all nodes that are referenced as target nodes contained in node n in Def. 7.3.6.

Definition 7.3.6: Children Nodes Function

Let G be a graph typed over T_N and T_E and $EL = N \dot{\cup} E$ be the set of its elements. The function $children_E: N \rightarrow \mathcal{N}$ returns the set of children nodes $N_C \subseteq N$ for a given node n , such that

$$children_E(n) = \{n_c \mid e \in E \wedge c(e) = true \wedge s(e) = n \wedge t(e) = n_c\} = N_C \quad (7.8)$$

Ex. 7.3.3 continues to demonstrate the children computation function based on the example graph presented in Fig. 7.3.1.

Example 7.3.3: Children (*Contained*) Annotations

In the example depicted in Fig. 7.3.1, the children nodes of the Java model jm are the Java package jp and the compilation unit cup . Consequently, $children_E(jm) = \{jp, cup\}$.

7.3.3 Computation of Missing Annotations

Based on the modified formalization which represents incompletely annotated multi-variant graphs, this section describes the process of computing missing annotations. The first part describes how to determine the missing annotations schematically to gain an overview of the three proposed strategies and their embedding in the propagation mechanism so far. Based on that information, the second part explains how to compute the elements which miss annotations and the third part offer details on the completion strategies for nodes followed by the strategy for edges.

I Schematic Propagation Procedure

Overview Due to the information missing in incomplete traces, the trace-based propagation does not behave as in the base version depicted in Fig. 5.3.1 but consists of two steps as depicted in Fig. 7.3.2. Since the transformation based on an incomplete trace may only produce a partial mapping annotation function $map_{p_{F_{H_T}}}$ resulting in a partially annotated multi-variant target graph $MH_{p_{H_T}}$ (1), the second step creates the completely annotated graph MH_{H_T} . Accordingly, the second box depicted in the middle of Fig. 7.3.2 represents a black-box algorithm which should determine missing annotations based on the partially annotated model. The input to this step encompasses the already produced partial mapping annotation function as well as the model, i.e., the partially annotated multi-variant graph $MH_{p_{H_T}}$. Both are indispensable for determining dependencies between model elements based on the heuristic algorithms we propose next.

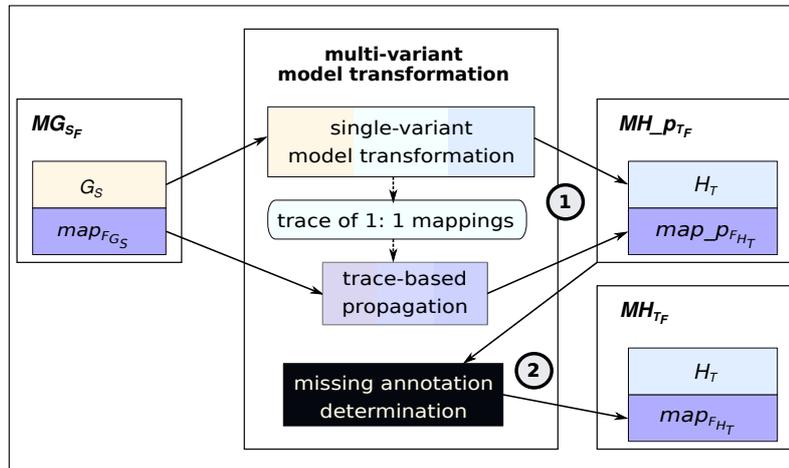


Figure 7.3.2: Schematic overview of employing completion strategies.

Please note: the graph H_T is not modified anymore but required as input to the second step to consult its structure and the dependencies between its nodes.

Assignment Strategies For determining missing annotations we introduce three algorithms which utilize the containment structure of EMF-based models. Since the Ecore metamodel fosters a strict existence relationship, we assume the same existence relationships in the multi-variant graphs. Accordingly, the multi-variant graphs form a spanning containment tree. Consequently, if the algorithms should be applied to other kinds of models conforming to a less restrictive

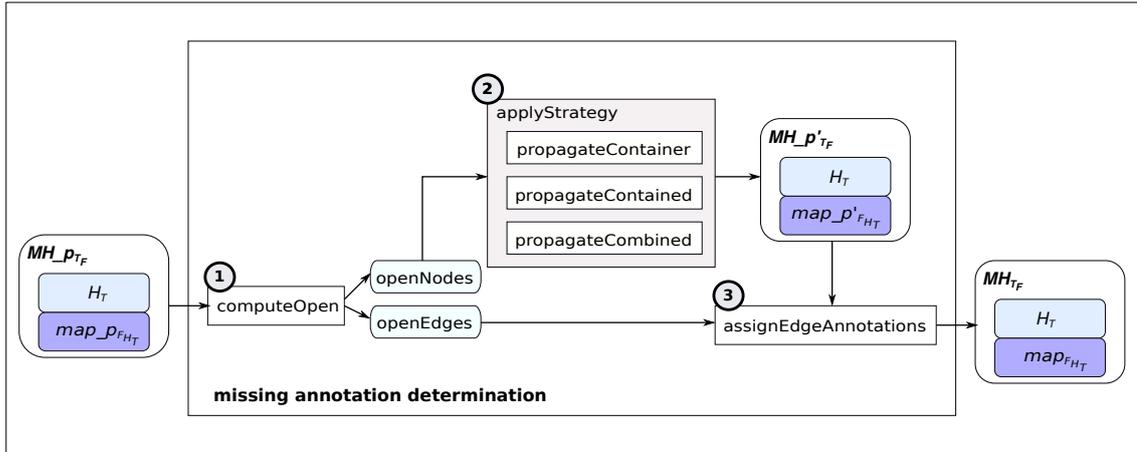


Figure 7.3.3: Overview of steps to complete annotations.

containment structure, first, the spanning containment tree will have to be determined. Otherwise, the algorithms cannot guarantee a proper annotation result.

The baseline of determining a missing annotation is the extended visibility function (Def. 7.3.3). In case of a missing annotation, its result is comparable to either mapping `true` or `false` onto each element missing an annotation.

Overview of Strategies In contrast, the first proposed strategy, *container*, assigns the annotation of the parent node of the node missing the annotation. Similarly, the second strategy, *contained*, combines the annotations of the contained nodes in a disjunction and assigns it to the current element. Finally, the first and second strategy can be *combined* to determine the annotations of the container nodes firstly and computing those of the children afterwards. Accordingly, the following three possibilities compute an annotation for each element el still missing an annotation and are enumerated here in short. The subsequent paragraphs explain them in detail.

1. *container*: use the annotation of el 's container for mapping assignment
2. *contained*: build the disjunction of the annotations of el 's contained elements for mapping assignment
3. *combined*: combine the el 's container (strategy (1)) annotation and the disjunction of the annotations of el 's contained elements (strategy (2)) in a conjunction for mapping assignment

Overview of Computing Missing Annotations Before describing the details of the three completion algorithms, Fig. 7.3.3 illustrates the dependencies and key parameters of the processed steps to determine a missing annotation.

According to Fig. 7.3.3, the computation of missing annotations consists of three basic steps. In the figure, rectangles represent operations and half-rounded rectangles the in- or output of an operation. At first, the `computeOpen` algorithm (Alg. 3) determines the elements still missing an annotation. It distinguishes nodes from edges and collects them in two sets ordered by a pre-order traversal of the spanning containment tree formed by the input model. The set of nodes is input to the second step, which employs *one* of the three strategies, prescribed in Alg. 4 - 6 to determine the annotations for each node in the set based on the partially annotated model. This step results in a partial mapping function $map_p'_{r_{HT}}$ where each node carries an annotation but not necessarily every edge. Consequently, the third step provides the new partially annotated model as well as the set of edges missing an annotation as input to the algorithm `assignEdgeAnnotations` (Alg. 7). Based on the annotations of the source and target node connected by an edge, the algorithm determines the annotations of the edges. To this end, the resulting mapping annotation function is total yielding a completely annotated graph.

Before starting the ‘missing annotation determination’ process with the partial model as input, as sketched in Fig. 7.3.3, the algorithm checks whether an annotation is already mapped onto the root of the model. If the root misses an annotation, the root element will receive the annotation of the root of the feature model because otherwise the entire target model would not exist. Additionally, this check and potential assignment of an annotation to the root of the model guarantees that container expressions exist when traversing the containment hierarchy of the considered model from top to bottom.

II Computation of Open Elements

After having determined the root annotation, the first step of Fig. 7.3.3 collects all graph elements without an annotation as described in Alg. 3. The algorithm computes two distinctive sets of *open* elements in the graph. It distinguishes nodes (*OpenNodes*) from edges (*OpenEdges*) to compute their missing annotations separately and those of the nodes in the first place.

Algorithm 3 Compute ordered sets of nodes and edges missing an annotation.

```

1: procedure COMPUTEOPEN( $H_T, map_{p_{F_{H_T}}}$ )
2:   in  $H_T$  ▷ target subgraph
3:   in  $map_{p_{F_{H_T}}} : EL_{H_T} \rightarrow A_F^0$  ▷ Partial annotation function for the target subgraph  $H_T$ 
4:
5:   var  $OpenNodes := \{\}$  ▷ Initialize ordered set of open nodes
6:   var  $OpenEdges := \{\}$  ▷ Initialize ordered set of open edges
7:   var  $el \in EL_{H_T}$  ▷ Processed target element
8:
9:   for  $el \in N_{H_T}$  do ▷ Topologically process all nodes of target graph
10:    if  $map_{p_{F_{H_T}}}(el) = \epsilon$  then
11:       $OpenNodes := OpenNodes \cup \{el\}$  ▷ Append element to set of open nodes
12:    end if
13:  end for
14:  for  $el \in E_{H_T}$  do ▷ Process all edges of target graph
15:    if  $map_{p_{F_{H_T}}}(el) = \epsilon$  then
16:       $OpenEdges := OpenEdges \cup \{el\}$  ▷ Append element to set of open edges
17:    end if
18:  end for
19: end procedure

```

Subsequently, the algorithm iterates the nodes of the target input graph H_T (i.e., the set N_{H_T}) in a pre-order traversal. Line 9 starts with collecting the set of open nodes. If no annotation is mapped onto the processed node (c.f., Line 10), the algorithm will append the node to the back of the set of open nodes. Similarly, the second loop iterates the set of edges (c.f., Line 14) and appends each edge which misses an annotation to the set of open nodes (c.f., Line 15). As opposed to the set of open nodes, the order of edges is irrelevant because the annotation which is computed for the edges, will be a conjunction of the their source and target node annotations.

Altogether, after having processed all elements of N_{H_T} , the sets of open nodes is ordered according to a pre-order traversal of the spanning containment tree formed by the model while the set of open edges contains them in arbitrary order.

Example 7.3.4: Computation of Open Elements

This example extends Ex. 7.3.1 in the way presented in Fig. 7.3.4. For easier readability, the (source) edges are not annotated and not recorded by the 1:1 links in this scenario. The example assumes a modified C2C transformation rule: Instead of creating a Java class declaration and corresponding compilation unit `CUnit` only, the transformation also adds an interface declaration (`InterfDecl`) to the Java package `jp` as well as a corresponding

compilation unit for the interface to the Java model jm . While the Java model stores the compilation units as before, the interface declaration is stored in the package jp linked with the UML package up comprising the transformed UML class ucp . Moreover, a package, called $impl$, should store each class declaration. Accordingly, the C2C rule will create the implementation package jpi stored inside the package containing the interface declaration additionally if this package does not exist in the containing package yet (i.e., for the first transformed class stored in this package).

As pivotal elements, the trace still records the UML classes and the Java class declarations and propagates the annotations depicted in the blue-colored rounded rectangles, accordingly. Consequently, the annotations of the two compilation units for each C2C rule application as well as of the implementation package and the interface declarations are missing after propagating annotations based on the incomplete trace.

The computation of the open elements iterates the target model MH_{TF} in pre-order traversal. The collection of nodes missing an annotation examines the left subtree of the root node, at first. The package jp is annotated but its children, both interface declarations of the *Person* and *Family* class as well as the implementation package, miss an annotation so that the algorithm adds them to the set of open nodes subsequently. The class declarations which are processed afterwards, carry annotations and thus, are not integrated into the set of open nodes. Similarly, iterating the remaining nodes of the model first appends the compilation unit for the *Person* interface and class declaration followed by the compilation units for the *Family* interface and class declaration to the ordered set of open nodes. As a result, the set of open nodes encompasses all nodes of the model missing an annotation in their hierarchical order: $OpenNodes = \{jif, jip, jpi, cup, cupi, cuf, cufi\}$

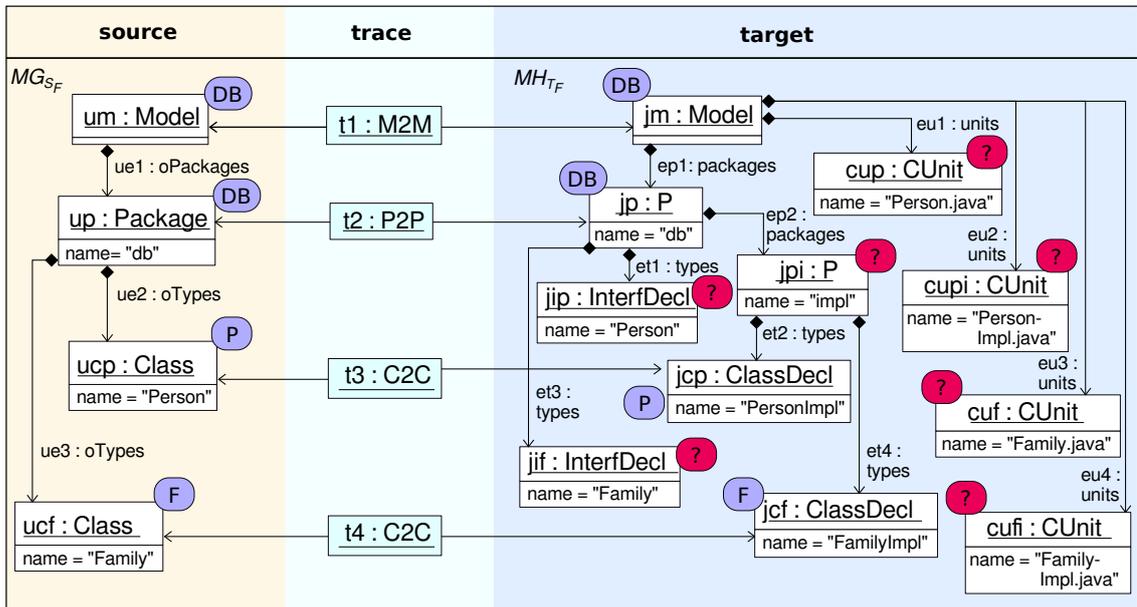


Figure 7.3.4: Commutativity violation due to incomplete traces in extended example. The C2C rule creates a class declaration, stored in an package $impl$, an interface declaration and two compilation units for storing the interface and class, respectively.

Please note: Even though the source edges are not annotated in the example, the pre-order traversal can also record the edges which miss an annotation. For reasons of better performance, it is suggested to iterate the partially annotated model only once and record both sets of open elements simultaneously. Recording the edges additionally results in a set of open edges, which comprises each edge of the target model. In Ex. 7.3.4 the set constitutes as follows: $OpenEdges = \{ep1, et3, et1, ep2, et2, et4, eu1, eu2, eu3, eu4\}$.

III Computation of Open Node Annotations

In the next step, one of the three completion strategies processes the ordered set of open nodes. The following paragraphs introduce the algorithms describing the processed steps in the container, the contained and the combined strategy, subsequently.

Container Strategy Alg. 4 describes the behavior of the *container* strategy. The algorithm iterates the set of open nodes from the beginning to the end (i.e., according to the pre-order traversal order) (l. 8). For each processed node n , in line 9 the algorithm determines the annotation of its container by employing the *parent* function defined in Def. 7.3.5. Then, the algorithm assigns the annotation stored in n_p to n by updating the partial mapping function in line 10. As a result, n may serve as annotated parent node if one of its children misses an annotation as well. Thus, the algorithm guarantees the availability of an annotation for the parent node for each processed node. Moreover, it ensures referential and existence integrity for the set of open nodes: whenever a node $n \in OpenNodes$ is integrated in a configured product, it is ensured that its container exists in the same product, too.

Algorithm 4 Assignment of the annotation of the container node.

```

1: procedure PROPAGATECONTAINER( $OpenNodes, MH\_p_{F_T}$ )
2:   in  $OpenNodes$                                 ▷ ordered set of nodes without annotation
3:   in  $H_T$                                          ▷ target subgraph
4:   in  $map\_p_{F_{H_T}} : EL_{H_T} \rightarrow A_F^0$     ▷ Partial annotation function for  $H_T$ 
5:   out  $map\_p'_{F_{H_T}} : EL_{H_T} \rightarrow A_F^0$     ▷ Partial annotation function for  $H_T$  with all nodes
      annotated
6:
7:    $map\_p'_{F_{H_T}} := map\_p_{F_{H_T}}$                 ▷ Initiate total annotation function with partial one
8:   for  $n \in OpenNodes$  do                          ▷ Iterate containment hierarchy top-down
9:     var  $n_p := parent_E(n)$                         ▷ get container node
10:     $map\_p'_{F_{H_T}}(n) := map\_p'_{F_{H_T}}(n_p)$     ▷ assign annotation of container node
11:  end for
12: end procedure

```

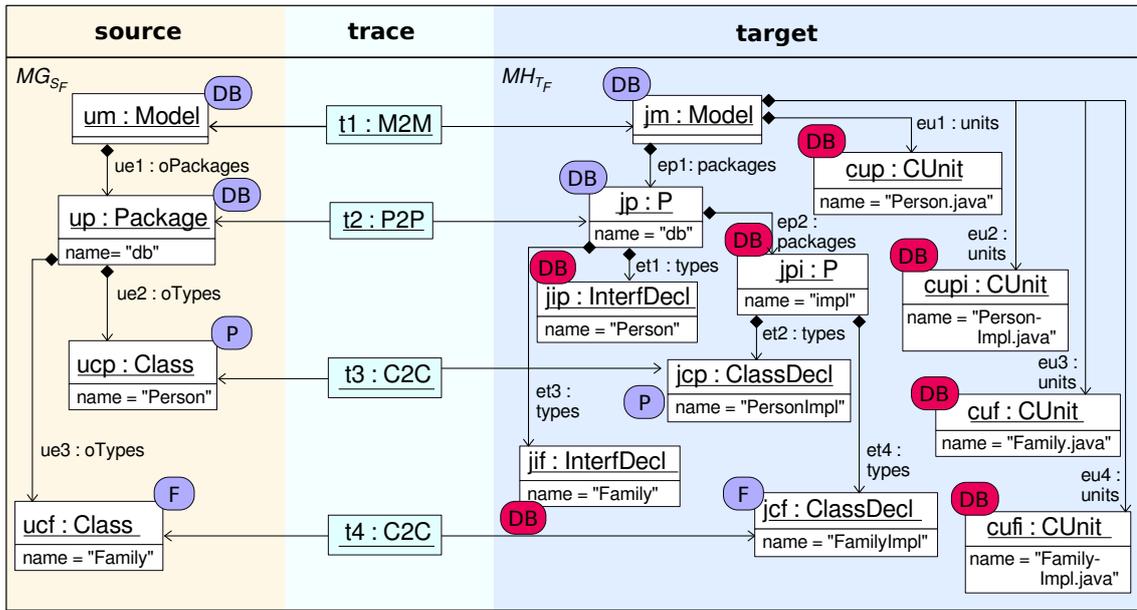
Example 7.3.5: Computation of Container Annotation

The extended example, depicted in Fig. 7.3.4, misses annotations for the set of open nodes, $OpenNodes = \{jif, jip, jpi, cup, cupi, cuf, cufi\}$, which are computed in Ex. 7.3.4. This lays the grounds for this example which describes how to compute the missing annotations for this set of open nodes based on the *container* completion strategy. Fig. 7.3.5 summarizes the result:

Firstly, the algorithm determines the annotation of the interface declaration *jif* corresponding with the UML class *Family*. The Java package *jp* represents its parent node onto which the annotation *DB* mapped. Accordingly, the annotation which is computed for the interface *jif* is *DB*.

Similarly, the next elements, the interface declaration for the class *Person* (*jip*) as well as the implementation package (*jpi*), receive the same annotation because the package *jp* encompasses these two elements, too. Thereafter, the compilation units receive an annotation, which is also *DB*.

As a result, in this example the container strategy assigns the annotation *DB* to each of the nodes stored in $OpenNodes$. Apparently, in this example the result of the algorithm not optimal because the root feature is assigned to all elements missing an annotation.

Figure 7.3.5: Computed annotations by *container* completion strategy (Alg. 4).

Contained Strategy The contained strategy – presented in Algorithm 5 – iterates the ordered set from the end to the beginning. For that reason, at first, a REVERSE operation inverts the set of open nodes (l. 8) which results from the pre-order traversal. Each element receives the annotations of its children nodes combined in a disjunction (l. 14). Consequently, the node (containing the children) has to exist as soon as at least one of the children nodes exists.

If no children elements exist (i.e., a leaf of the tree is processed), an empty disjunctive clause would be the result which is defined to evaluate to **false** in Boolean algebra. Since this would remove the element from each derived product (c.f., discussion of model filters in Sec. 7.3.1), as alternative, the algorithm retains each leaf node in a resulting product by assigning the root feature instead of the empty disjunctive clause in line 12. Nonetheless, due to the bottom-up propagation, all elements processed after the leaf nodes possess children where annotations are assigned.

Example 7.3.6: Computation of Contained Annotation

This example explains how to compute annotations for the set of open nodes of the example depicted in Fig. 7.3.4, based on the contained strategy.

First, the set of open nodes is reversed. Accordingly, the reversed set starts with the compilation units and comprises the interface declarations such that in the end:

Reversed = {cufi, cuf, cupi, cup, jpi, jip, jif}.

The result of the following computation deviates only in the annotation mapped onto the package impl (object jpi) from the annotated multi-variant target model created by the container strategy, shown in Fig. 7.3.5:

First, the algorithm processes the four compilation units which are leaf nodes of the spanning containment tree. The example assumes that this corner case is implemented as proposed in Alg. 5, which maps the annotation of the root feature, which is DB, onto those nodes. Consequently, the compilation units are integrated in every derived product.

Next, the annotation of the implementation package jpi is determined which is $P \vee F$, which combines the annotations of its children nodes in a disjunction. For the remaining open nodes, representing the interface declarations, the same situation as for the compilation unit occurs: the root feature DB is mapped onto them because both elements do not comprise children nodes.

Algorithm 5 Assignment of annotations based on contained elements.

```

1: procedure PROPAGATECONTAINED( $OpenNodes, MH\_p_{TF}$ )
2:   in  $OpenNode_s$                                 ▷ ordered set of nodes without annotation
3:   in  $H_T$                                           ▷ target subgraph
4:   in  $map\_p_{FH_T} : EL_{H_T} \rightarrow A_F^0$       ▷ Partial annotation function for  $H_T$ 
5:   out  $map\_p'_{FH_T} : EL_{H_T} \rightarrow A_F^0$      ▷ Partial annotation function for  $H_T$  with all nodes
      annotated
6:
7:    $map\_p'_{FH_T} := map\_p_{FH_T}$                     ▷ Initiate total annotation function with partial one
8:   var  $Reversed = REVERSE(OpenNodes)$              ▷ reverse the set
9:   for  $n \in Reversed$  do                            ▷ iterate bottom-up
10:    var  $Children = children_E(n)$                  ▷ set of children nodes
11:    if  $Children = \emptyset$  then
12:       $map\_p'_{FH_T}(n) := GETROOTFEAT()$ 
13:    else
14:       $map\_p'_{FH_T}(n) := \bigvee_{c \in Children} map\_p'_{FH_T}(c)$ 
15:    end if
16:  end for
17: end procedure

```

Combined Strategy Finally, Alg. 6 describes the realization of the *combined* strategy. The set of open nodes is iterated twice. Similarly as in Alg. 4, the first iteration of the set of nodes occurs from front to back according to the pre-ordered sequence (l. 9) and assigns the annotation of the parent node to each processed node (l. 11).

Secondly, the algorithm iterates the set of open nodes in reverse order (l. 14) in the same way as in Alg. 5. In contrast to the *contained* strategy, this algorithm does not have to determine an annotation if children elements are missing because the previous iteration ensures that at least the annotation of the container is already assigned. However, if children nodes are present, the algorithm will combine their annotations in a disjunction (l. 18). In the end, the resulting clause is combined with the annotation assigned in the first iteration in a conjunction (l. 19) and assigned this way to the respective node.

Example 7.3.7: Computation of Combined Annotation

Fig. 7.3.6 demonstrates the annotations resulting from applying the *combined* strategy to compute the annotations missing in Fig. 7.3.4.

First, the annotations of the parent nodes are assigned as in Ex. 7.3.5. Accordingly, each node in the set of open nodes receives the annotation DB . Afterwards, the algorithm iterates the reversed set of open nodes. If a node contains children nodes, their annotations will be combined in a disjunction. The resulting annotation is combined in a conjunction with the already present parent annotation. In this example, this is the case for the package `impl` (object: `jpi`). If the node does not contain children, the annotation computed in the first iteration will remain unchanged.

Thus, after executing the *combined* strategy, the mapping annotation function associates the following Boolean expressions with respective model elements:

$$\begin{aligned}
 map_p'(jif) = map_p'(jip) &= && DB \\
 map_p'(jpi) &= && DB \wedge (P \vee F) \\
 map_p'(cup) = map_p'(cupi) = map_p'(cuf) = map_p'(cufi) &= && DB
 \end{aligned}$$

All in all, the *combined* strategy may increase the execution time due to iterating the set of nodes twice but not its complexity, as the second iteration visits exactly the same number of elements. The complexity of each algorithm is discussed in Sec. 7.3.4. Moreover, it is expected that the

Algorithm 6 Assignment of combined (parent and children) annotations.

```

1: procedure PROPAGATECOMBINED(OpenNodes, MH_pTF)
2:   in OpenNodes                                ▷ ordered set of nodes without annotation
3:   in HT                                         ▷ target subgraph
4:   in map_pFHT : ELHT → AF0           ▷ Partial annotation function for the target graph HT
5:   out map_p'FHT : ELHT → AF0 ▷ Partial annotation function for target graph HT with all
      nodes annotated
6:
7:
8:   map_p'FHT := map_pFHT                       ▷ Initiate total annotation function with partial one
9:   for n ∈ OpenNodes do                         ▷ Iterate containment hierarchy top-down
10:    var np := parentE(n)                       ▷ get container node
11:    map_p'FHT(n) := map_p'FHT(np)           ▷ assign annotation of container node
12:  end for
13:
14:  var Reversed = REVERSE(OpenNodes)             ▷ reverse the set
15:  for n ∈ Reversed do                           ▷ iterate bottom-up
16:    var Children = childrenE(n)               ▷ set of children nodes
17:    if Children ≠ ∅ then
18:      var childClause =  $\bigvee_{c \in \text{Children}} \text{map\_p'F}_{HT}(c)$ 
19:      map_p'FHT(n) := map_p'FHT(n) ∧ childClause
20:    end if
21:  end for
22: end procedure

```

accuracy of the determined annotations with respect to satisfying commutativity when employing the combined strategy increases. Sec. 7.3.4 further elaborates on the accuracy of the heuristics to determine annotations in partially annotated models. Finally, Sec. 10.3 examines the accuracy of the completion algorithms in different transformation scenarios in practice.

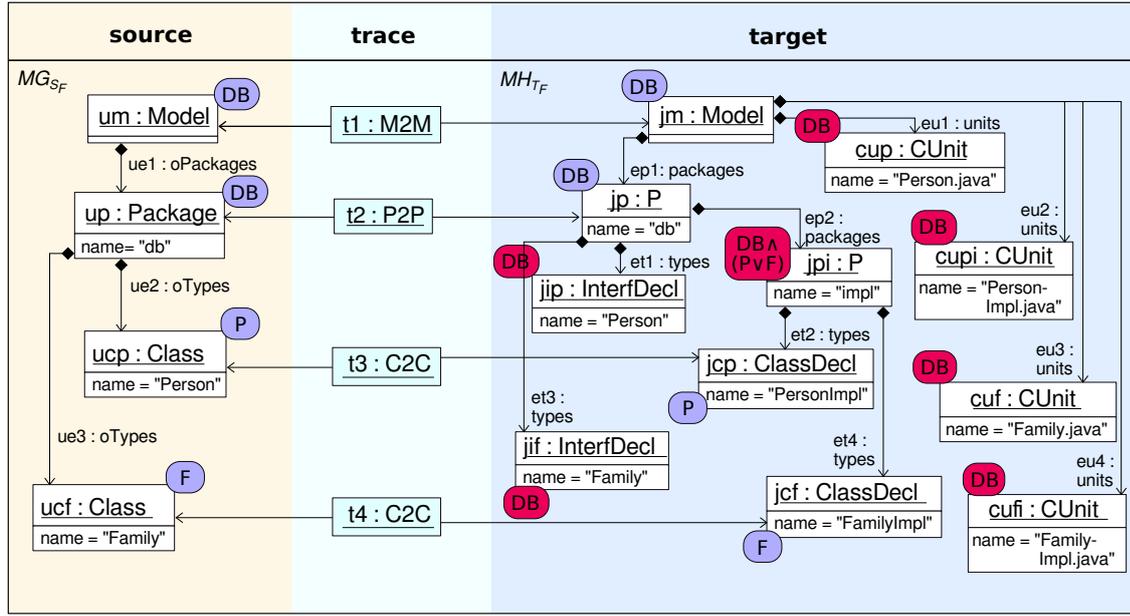
IV Computation of Missing Edge Annotations

Besides the set of open nodes, Alg. 3 computes the set of edges missing an annotation. This section explains how the set of open edges is processed.

After having determined annotations for each node, it is possible to consider the edges, regardless of whether they are forming the spanning containment tree (containment references) or establish a cross-reference between arbitrary nodes. Performing the algorithms in the order sketched in Fig. 7.3.3 guarantees that an annotation is present on each node.

To ensure referential integrity, both ends of an edge have to be incorporated in a derived product for avoiding dangling references. Otherwise, it depends on the behavior of the model filter whether the edge is included in each derived product or excluded. Therefore, the algorithm assigning annotations to edges retrieves the annotation of the source node (*s*) and of the target node (*t*) and combines them in a conjunction. This expression is assigned to the respective edge by employing the mapping annotation function.

Alg. 7 describes the assignment of annotations to edges missing one. The partially annotated target graph as well as the set of open edges are input to the algorithm. As sketched in Fig. 7.3.3, the partial mapping already records an annotation for each node in the graph. In contrast to processing the set of open nodes, the order of processing the edges is irrelevant because all nodes are annotated. Consequently, the set of open edges may be iterated in arbitrary order. For each edge in the set, the algorithm assigns the conjunction of the annotation of its source and target node (l. 9) to the edge. To this end, the entire completion process ensures that an annotation is not only mapped onto each node but also onto each edge in the graph.


 Figure 7.3.6: Computed annotations by *combined* completion strategy (Alg. 6).

Algorithm 7 Assignment of annotations to edges.

- 1: **procedure** ASSIGNEDGEANNOTATIONS($OpenEdges, MH_{p_{F_T}}$)
- 2: **in** $OpenEdges$ ▷ set of edges without annotation
- 3: **in** $H = H_S \leftarrow H_{TR} \rightarrow H_T$ ▷ target STT graph
- 4: **in** $map_{p'_{F_{H_T}}} : EL_{H_T} \rightarrow A_F$ ▷ Partial annotation function for the target graph H_T
 where all nodes are annotated
- 5: **out** $map_{F_{H_T}} : EL_{H_T} \rightarrow A_F$ ▷ Total annotation function for the target graph H_T
- 6:
- 7: $map_{F_{H_T}} := map_{p'_{F_{H_T}}}$ ▷ Initiate total annotation function with partial one
- 8: **for** $e \in OpenEdges$ **do** ▷ Iterate set of edges
- 9: $map_{F_{H_T}}(e) := map_{F_{H_T}}(s(e)) \wedge map_{F_{H_T}}(t(e))$ ▷ assign annotation of container node
- 10: **end for**
- 11: **end procedure**

Example 7.3.8: Assignment of Annotations to Edges

After having assigned annotations to each node by one of the three strategies introduced before, the set of open edges is iterated to assign annotations.

As a result of determining elements missing annotations in Ex. 7.3.4, the set of open edges encompasses each edge in the target model: $OpenEdges = \{ep1, et3, et1, ep2, et2, et4, eu1, eu2, eu3, eu4\}$

Since the algorithm computes annotations after having applied one of the three strategies, all nodes are already annotated. We assume that the container strategy (Alg. 4) provided the missing node annotations. Then, Alg. 7 assigns the annotations to the edges in the mapping function, which Fig. 7.3.7 highlights by blue rectangles with white letters, as follows.

Again we refrain from simplifying the annotations to show their composition based on the annotations of the source node (left side of the conjunction) and target node (i.e., the right

different scenarios.

I Computational Complexity

As illustrated in Fig. 7.3.3 the process of determining the total mapping function consists of three major steps: first, the collection of graph elements missing annotations, second, the computation of node annotations and third, the computation of edge annotations. For selecting one of the heuristics, its computational complexity may be regarded and weighed against its benefits for accuracy. Accordingly, we first analyze the complexity to offer a worst case estimation of the computational complexity regarding time consumption. For that reason, this part discusses the complexity of the algorithms which determine the open sets of elements in the target graph missing an annotation first. Next, the complexity of determining the missing annotations for the graph edges is discussed before regarding the three heuristics which compute the missing annotations for the graph nodes.

Preliminaries For analyzing the algorithms in an isolated way the following variables are of interest:

The input to the algorithms is the partially annotated target graph $MH_T = (H_T, map_p_{H_T})$. A number of $n = |N_T|$ nodes and of $e = |E_T|$ edges counts the elements of the target graph H_T , where $x = n + e$ represents the number of all graph elements.

Furthermore, let $n' < n$ and $e' \leq e$ be the number of nodes and edges missing an annotation, respectively. Even though no annotations were assigned before by an incomplete propagation, the preprocessing step would annotate the root node of the target graph with the root feature, ensuring that at least one node out of the n nodes of H_T is annotated. This in turn guarantees that the number of open nodes is smaller than the number of all nodes in H_T .

Moreover, the strategies to determine missing annotations based on the already assigned annotations have to access the mapping function. In reality the complexity of accessing and storing elements in maps depends on several factors, such as the implementation of the map, the capacity in case of a hash map and a balanced dispersal among buckets, etc., which in the best case is $O(1)$ and in the worst case $O(m)$ with m representing the number of elements stored in the map. For that reason, in the sequel we assume the worst case of a complexity of $O(x)$ for retrieving a graph element from and adding a graph element to the map.

Termination In the first place, it must be guaranteed that the annotation determination strategies terminate on an arbitrary partially annotated multi-variant graph as input. The main loop that iterates the set of open nodes or edges is decisive for the termination of the algorithms to find the missing annotations.

As stated in the preliminaries, the number of open nodes $n' = |OpenNodes|$ is limited by the finite number of nodes in the multi-variant input graph $n = |N|$, such that $n' < n$. Moreover, the sets of nodes and edges in H_T are finite implying that the sets of open nodes and edges are finite as well when beginning the iteration of the sets.

Secondly, the algorithms neither add nodes to the set of open nodes nor edges to the set of open edges. Moreover, in case of the contained and combined strategy another loop iterates the children of a node inside the loop which processes the open nodes. Since the number of all nodes in H_T limits the number of children sharply, i.e., $|children_E(n)| < |N|$, and no children nodes are added, the inner loop as well as the outer loop terminate when determining annotations for the nodes.

Similarly, only one loop iterates the finite set of open edges. Assuming one computation step is required to access the source and target node connected by an edge, the loop iterating the set of open edges is guaranteed to terminate, too, because the algorithm does not modify the set. Consequently, since the number of elements in the sets of open nodes (and children nodes) and edges remains finite, processing the sets is guaranteed to terminate.

Complexity of Computing Set of Open Elements The first step executes the algorithm to determine the sets of open nodes and edges (Alg. 3). This algorithm exposes *quadratic execution time* with respect to the number of graph elements as upper threshold as the following deduction

Algorithm 8 Asymptotic complexity of computing open nodes and edges.

```

1: ...
2: for  $el \in EL_{HT}$  do  $\triangleright O(x)$ 
3:   if  $map_{p_{FH_T}}(el) = \epsilon$  then  $\triangleright O(x)$ 
4:     if  $el \in N_{HT}$  then  $\triangleright O(n)$ 
5:        $OpenNodes := OpenNodes \cup \{el\}$   $\triangleright O(n')$ 
6:     else  $\triangleright O(1)$ 
7:        $OpenEdges := OpenEdges \cup \{el\}$   $\triangleright O(e')$ 
8:     end if
9:   end if
10: end for

```

shows. The upper thresholds for the time-relevant processing steps of Alg. 3 are put in comments behind the corresponding expressions in Alg. 8.

The algorithm executes the outer loop for each node and each edge implying the upper threshold for the execution time is $O(x)$ in Line 2. Inside the loop, the algorithm checks whether an annotation is already mapped onto the element, which requires to access the mapping function. Therefore, Line 3 spends at most an execution time of $O(x)$. Thereafter, it is checked whether the element is included in the set of nodes which requires to compare the element with n nodes at maximum, resulting in a time of $O(n)$. In case the processed element is not present in the set of open nodes, it is contained in the set of edges causing no additional runtime.

Next, the algorithm appends the element to the already determined set of open nodes. Depending on the implementation of appending elements to the set of elements, its complexity may be constant or linear: If the element is appended to the end of the corresponding set without a need to iterate the entire set, a complexity of $O(1)$ can be achieved. Nonetheless, in the worst case the complexity of Line 5 exposes linear time of $O(n')$ when not allowing for accessing the last element of a collection in constant time and similarly $O(e')$ in Line 7.

All in all, each graph element is processed once in the outer loop while the *inner loop* performs the following number of computation steps at maximum: $x + n + n'$ or $x + n + e'$, with respect to the element being an edge or a node. We can approximate the inner processing steps as $x + x + x = 3x$ because $n, n', e' < x$. As a result, the overall complexity sums up to $O(x * 3x) = O(x * x) = O(x^2) = O((n + e)^2)$. Summing it up, computing the sets of open nodes and edges performs in quadratic execution time with respect to the number of nodes and edges in the worst case.

Complexity of Computing Edge Annotations To determine the annotation of all edges missing one, the set of edges is iterated once. Accordingly, this process requires linear time of $O(e')$ in the worst case. In addition, accessing the source and target node of an edge should be managed in constant time ($O(1)$) by following a reference. Accessing the mapping of both nodes inside this loop exposes a time of $O(x)$ at maximum yielding an overall asymptotic execution time of quadratic amount: $O(e' * 2x)$. Thus, the algorithm performs in quadratic time with respect to the number of all elements in the graph in the worst case.

Complexity of Computing Node Annotations Regarding the execution time of the heuristics implemented in the three strategies, the algorithms expose an asymptotic quadratic and cubic complexity with respect to the number of nodes in the worst case.

Complexity of Container Strategy The container strategy iterates the set of open nodes once ($O(n')$) and accesses the container of each node. Alg. 9 summarizes the asymptotic estimations of the execution time as comments in each line.

We assume that the parent node can be accessed in *constant* time in Line 4. This assumption is justified because it requires to follow the unique incoming edge labeled as container. A pointer may realized accessing this edge since it is the single edge of this kind for each node. Thereafter, accessing the mapping function exposes linear complexity $O(x)$ in the worst case plus assigning the resulting annotation to the same map of annotations which spends the same amount of time

Algorithm 9 Complexity of container strategy (1).

```

1: ...
2:  $map\_p'_{FH_T} := map\_p_{FH_T}$  ▷ Initiate total annotation function with partial one
3: for  $n \in OpenNodes$  do ▷  $O(n')$ 
4:   var  $n_p := parent_E(n)$  ▷  $O(1)$ 
5:    $map\_p'_{FH_T}(n) := map\_p'_{FH_T}(n_p)$  ▷  $O(2 * (x))$ 
6: end for

```

in the worst case in Line 5. Consequently, the worst case asymptotic execution time is quadratic: $O(n' * (2 * x)) = O(x^2)$.

Complexity of Contained Strategy In contrast, the contained and combined strategy require to iterate the number of children nodes $n_c \leq n$ inside the outer loop which iterates the set of open nodes. As sketched in Alg. 10, this additional iteration of the children nodes, which accesses the mapping function for each child node, exposes an asymptotic execution time of $O(n * x)$ (c.f., Line 9) at maximum. In total, the computation steps *inside the loop* sum up to $n + x$ or $n + n * x + x$ in the first branch (no children) or second branch, respectively. As a result, the worst case execution time inside the loop is $O(x^2)$. Since iterating the (reversed) set of nodes requires linear time, in total the asymptotic execution time sums up to $O(x^3)$ in the worst case.

Algorithm 10 Complexity of contained strategy.

```

1: ...
2:  $map\_p'_{FH_T} := map\_p_{FH_T}$  ▷ Initiate total annotation function with partial one
3: var  $Reversed = REVERSE(OpenNodes)$  ▷  $O(n')$ 
4: for  $n \in Reversed$  do ▷  $O(n')$ 
5:   var  $Children = children_E(n)$  ▷  $O(n)$ 
6:   if  $Children = \emptyset$  then
7:      $map\_p'_{FH_T}(n) := GETROOTFEAT()$  ▷  $O(x)$ 
8:   else
9:     var  $ann = \bigvee_{c \in Children} map\_p'_{FH_T}(c)$  ▷  $O(n * x)$ 
10:     $map\_p'_{FH_T}(n) := ann$  ▷  $O(x)$ 
11:   end if
12: end for

```

Complexity of Combined Strategy The *combined* strategy, which combines the contained and container strategy, exposes the asymptotic execution time approximated by the time to execute the contained strategy, which is of cubic complexity. This results from the following observation. First, the algorithm executes the container strategy exposing quadratic execution time at maximum. Afterwards, it executes the contained strategy, exposing cubic execution time. In summary, we can approximate the execution time as $O(x^2 + x^3)$ which results in an asymptotic execution time of $O(x^3)$.

Reflection An optimized implementation of accessing and manipulating the mapping function may, however, reduce the execution time significantly. If accessing and adding elements to the mapping function is executed in constant time, each asymptotic runtime will be reduced by an entire factor, i.e., the resulting execution time of the contained and combined strategy will be of quadratic ($O(x^2)$) and the container strategy of linear ($O(x)$) complexity.

All in all, this estimation shows that the complexity of the contained and the combined strategy is of the same size. Even though both strategies expose an additional potency of complexity, it is caused by approximating the number of children of each node by the number of all nodes. In reality, however, the number of children of an open node can at most only once reach the

amount of $n - 2$ causing that no further nodes are present to be processed anymore. Instead, the probability that the number of children nodes remains far below the number of all nodes in the graph is relatively high, even a number of zero children nodes is possible. If it could be assumed that the mapping of all children nodes can be accessed in linear time, there would be no difference between the three strategies with respect to the asymptotic execution time.

II Accuracy

As mentioned before, the accuracy of the annotations determined by one of the heuristic strategies do not guarantee 100% correctness with respect to the commutativity criterion. Nonetheless, we argue that they result in a deterministic behavior, on the one hand, and improve the accuracy compared to having no annotations, on the other hand.

General Remarks In the first place, it must be noted that each of the strategies guarantees to determine annotations for all elements missing one. However, the accuracy is limited by the heuristics exploiting the structure of the model as well as the presence of annotations assigned beforehand, for example, based on the incomplete trace.

Container Strategy By assigning the annotation of the parent node, the container strategy (1) ensures referential integrity with respect to contained children. The strategy implies that if the parent node is present, its children nodes which were not annotated automatically will be present in a derived product, too. Moreover, since it should not be possible for a child node to be present without its parent, the child should not receive an annotation which is less restrictive. Consequently, assigning the annotation of the parent node guarantees that products incorporating a child without its parent will not be derivable if the child was missing an annotation. However, the annotation retrieved from the parent node may still be too broad, i.e., the child node may be included in too many products.

For instance, let \mathbf{r} denote the root element of the domain model. If children of \mathbf{r} are missing an annotation, the algorithm will assign the annotation of \mathbf{r} to its children as well. However, since \mathbf{r} should be present in each product, the root feature is mapped onto that element to ensure the existence of the derived variant. As a result, the children of \mathbf{r} , which miss an annotation, will be present in each derived product even if they realize an optional feature.

Moreover, if a hierarchy of elements misses an annotation, the container strategy propagates a potential coarse-grained annotation from the top along the hierarchy down to the bottom until the first element carrying an annotation is reached. Accordingly, although a more specific annotation would have to be assigned to any of the elements in this hierarchy, the annotations remain at the same broad level as at the top of the model hierarchy.

In the extended version of the introductory example depicted in Fig. 7.3.4, the compilation units of the Java model all miss an annotation. Since their parent is the root of the model, they receive the root feature as annotation. Consequently, when applying the parent strategy, all compilation units will be integrated in every derived product regardless of the fact whether they comprise only optional classifiers. For that reason, even the compilation unit holding the class `Family` is contained in a product realizing the feature configuration where `F` is deselected without containing the class. The filter will remove the class due to its more fine-grained annotation. In addition, if the compilation units contained further children which missed an annotation, they would receive the root feature as annotation rendering them present in every derived product, too. This limitation for leaf nodes manifests in each of the proposed algorithms in this example.

Contained Strategy In the contained strategy, the annotation of the children nodes is mapped onto the parent node missing one. Alg. 5 guarantees that each child is annotated before processing the parent. In this strategy a node without children nodes (i.e., a leaf node) would be completely excluded from every product if the annotation `false` was assigned. As our algorithm assigns the root feature as annotation instead, the leaf node is included in every product, meaning that the annotation is too broad.

Similar as in the container strategy, the coarse-grained annotation of a leaf node missing an annotation may propagate along the hierarchy of the spanning containment tree from bottom to top if multiple ancestors consecutively miss an annotation. Particularly, excluding these elements from every derived product by assigning the annotation `false` would result in derived products missing too many elements unconditionally, from which we refrain.

On its upside, if children nodes are present for a node missing an annotation, the determined annotation will guarantee *referential integrity* for the children of the node missing the annotation. If the algorithm combines all annotations of children nodes in a disjunction, it will guarantee that for each child of the node, which is pertained in a configured product, its parent is present, too. As a result, a well-formed structure with respect to the children dependency is guaranteed.

Moreover, if specific annotations are mapped onto all children nodes of the node missing the annotation, the resulting annotation will be *more specific* than assigning only the single annotation of the parent node in the contained strategy as well as more specific than assigning the root feature, which is the case if a leaf node misses an annotation. In concrete, the node will be present *only* if one of its children is present, i.e., the node will be only part of a product if its presence is required to ensure the existence of a child. However, if the parent of the node missing the annotation is annotated with an optional feature which does not imply the annotation of the children, it is possible to derive products, where the parent of the node is missing while the node and its children are present but without a valid container.

As highlighted in Ex. 7.3.6, the annotation of the compilation units serves as an example where children nodes are missing. Consequently, assigning the annotation `false` would remove all of them from every product event if they would contain a classifier realizing a mandatory feature. Then, the classifier might be integrated but without the corresponding compilation unit. In contrast, if the root feature is assigned to the compilation units, the same effect as of the container strategy occurs: too many compilation units may be present in derived products. However, the annotation computed for the implementation package (`jp1`) in the same example, is appropriate. The package will be included in derived products if the features `F` and `P` are selected solely or both. In case none of these two features is selected, the model filter removes the package due to its annotation. Since its parent, the package `jp`, is annotated with a mandatory feature, in this example the problem of removing a parent while its child, which was missing an annotation, remains in a derived product does not occur.

Combined Strategy The combined strategy is introduced to combine the benefits and to reduce the drawbacks of the contained and container strategy with respect to accuracy.

First of all, the annotations of parent nodes are mapped onto the nodes missing an annotation. This ensures that each of the open nodes carries an annotation but the risk remains that the propagated annotations are too broad which may reduce the accuracy. As a consequence, if a leaf node misses an annotation, the annotation of the parent will be present. Not assigning the root feature to such element is beneficial for the following reason: Assigning the root feature is comparable to assigning `true`. Instead, the annotation of the parent node may be more specific and therefore becomes decisive for the presence in a derived product.

If both, annotated children nodes and an annotated parent node, are present for a node missing an annotation before executing one of the strategies, the combined strategy is able to determine an annotation that preserves the containment structure in any feature configuration and which is more specific than assigning the parent or children annotations in isolation. If the parent node does not form part of the set of open nodes, the annotation which corresponds with the respective source node is mapped onto it. As a result, the annotation assigned in the first iteration (i.e., based on the container strategy) is at least as specific as the annotation of the parent node. However, it still may be too broad for the node missing the annotation. Due to the fact, however, that the annotations of the children are combined with this annotation, the node will only be present in a configured product if one of the children is present, too. Accordingly, if annotations are mapped onto the parent and children nodes of a node missing an annotation, the resulting annotation ensures that the node is only present if its parent and one of its children are present. Thus, the combination with the annotation of the children nodes refines the result and ensures the persistence of the containment hierarchy in configured products without dangling references.

Ex. 7.3.7 sheds lights on the benefits and shortcomings of the combined strategy. On the one hand, the resulting annotations may be too broad: Since the compilation units are stored in the root of the model and miss children nodes, the computed annotation is too broad: the compilation units form part of every derived product. For the same reason, the annotations of the interface declarations are too broad. Even if their container, the package `db`, was annotated with a more specific annotation than the root feature, the annotation of the package might not subsume the annotations of all classifiers contained in the package. On the other hand, the annotation of the implementation package (`jpi`) resides at the right level of accuracy. As annotations are mapped onto its parent and children nodes, the computed annotation guarantees the presence in the correct set of derived products: If the features `P` and `F` are deselected, the package will be removed. This is semantically correct and should be implemented in this way in the model transformation because the package is not needed in this case. If one of the features `P` and `F` or both are selected instead, the package will be present. Consequently, for this element the combined strategy (as well as the contained) strategy computes the correct annotation with respect to satisfying commutativity.

Conclusion The discussions on accuracy and complexity allow to suggest a strategy when propagating annotations based on incomplete trace information.

The discussion reveals that the *container* strategy assigns annotations which may be too broad and, as a result, include too many elements in the configured products. As mentioned in the introduction to this section (Sec. 7.3.1), including too many elements in configured products implies potentially leaking information and functionality which may not belong to the customer. Thus, the strategy may achieve higher accuracy than assigning the annotation true but may render leaf nodes visible in too many derived variants.

In contrast, the *contained* strategy uses the annotations of children elements to compute the missing annotation. This strategy ensures the presence of the parent node for children whose parent node is not annotated, on the one hand.

On the other hand, let n_p denote the parent node of the node n which misses an annotation and receives an annotation computed from the annotation of its children. If an annotation is mapped onto n_p based on the trace propagation (due to a 1:1 trace element), the contained strategy cannot ensure that n_p is present in the same set of configuration as n because it does not regard the annotation of n_p . Moreover, if n is a leaf node, the root feature is mapped onto it resulting in a too broad annotation. Consequently, the combined strategy assigns the most accurate annotation even though the corner cases of leaf nodes, particularly with the root node as parent, still receive the root feature as annotation which means the least specific one.

Weighing the accuracy against the computational complexity, the container strategy is ideal, if a fast executable solution is desired and broad annotations are sufficient. If the execution time is no critical factor and accuracy is of relevance, we recommend the combined strategy.

7.4 No Persistent Trace Information

If transformation execution engines do *not* persist trace information, either the information of corresponding elements of the source and target model for propagating annotations from source to target elements will have to be retrieved in different ways or the propagation mechanism must change.

This section sketches two possibilities to reconstruct the information of corresponding source and target elements despite missing execution traces. On the one hand, relationships between the source and the target model can be declared manually in a DSL with incorporated tooling to assign annotations based on the correspondences to the target model. On the other hand, the source and target model can be compared to detect similarities for inferring corresponding elements of source and target model automatically based on shared characteristics.

Road Map Therefore, this section starts with stating the problem by recapitulating the specifics of propagating annotations based on trace information. Thereafter, Sec. 7.4.2 and Sec. 7.4.3

introduce how a propagation DSL and a model matching approach, respectively, may propagate annotations despite missing traces.

7.4.1 Problem Statement

First of all, this section recapitulates properties classifying multi-variant model transformations. As declared in Sec. 4.3.2, multi-variant model transformations can be classified by different features. Trace-based propagation is a *post-processing, automated* approach which does not invade in the execution engine but – as a gray-box approach – relies on the trace as artifact directly resulting from the execution of the respective models. Since traces transformation engines persist trace information in varying ways, the trace-based propagation itself is generic with respect to the model transformation language as well as to the metamodels to which the source and target models, conform. However, some transformation execution engines do not maintain or persist trace information. For instance, the ATL default virtual machine records the source and target elements created by a matched rule but does not persist this information after the execution. Furthermore, the applications of lazy and called rules are not automatically recorded but could be maintained by the developer inside the specification. As a consequence, missing trace information requires to annotated the target model completely manually if no automation, as proposed in this section, is available.

The two approaches, presented in the following sections, do not require any access to the transformation specification or its execution engine artifacts. On the one hand, the propagation DSL, as one approach, requires to define corresponding source and target elements of two metamodels *manually* to automatically propagate annotations. On the other hand, by matching the source and target model the second approach may retrieve corresponding elements *automatically* which can be used for the automatic trace-based propagation. Both approaches are *post-processing* approaches that are not intertwined with a transformation execution if it is present at all.

7.4.2 Propagation DSL

To propagate annotations when no trace is available, this section presents the concepts of propagation DSLs which allow for declaring relationships between the metamodels. An interpreter receives the source and the target model and parses the specification. By iterating the source model, the interpreter assigns the annotation of a source element to its corresponding target elements as declared in the specification. For instance, the DSL **ModelSync** [BG18] realizes such mechanism.

Road Map In the sequel, the first part demonstrates how a DSL serves to propagate annotations. General design decisions for such approach are given in the second part whereas the third part demonstrates the DSL **ModelSync**, as one representative, in particular. The descriptions on propagating annotations based on a DSL close in the fourth part with a delimitation of other approaches and a discussion of the automation capabilities offered by this mechanism.

I Schematic Overview

First of all, the developer can use the DSL approach to declare corresponding elements between two metamodels (thereby replacing the need to retrieve it from a model transformation). Fig. 7.4.1 gives a schematic overview of the approach. While the first step (1) does not foster how the target model is created beforehand (e.g., by a model transformation or manually), which is sketched with the *create* box in the figure, the developer has to specify corresponding elements of the source and target metamodel in the DSL specification manually. In step two (2), the concrete DSL specification serves as input to the DSL propagation mechanism as well as the source model and source mapping (which may be combined in one artifact represented as MG_{S_F} in the figure). In summary, the manual task involves at least specifying corresponding source and target elements based on the metamodels. If the target model cannot be created automatically, it may origin from a manual creation process, too. However, how the target model is created is irrelevant to the

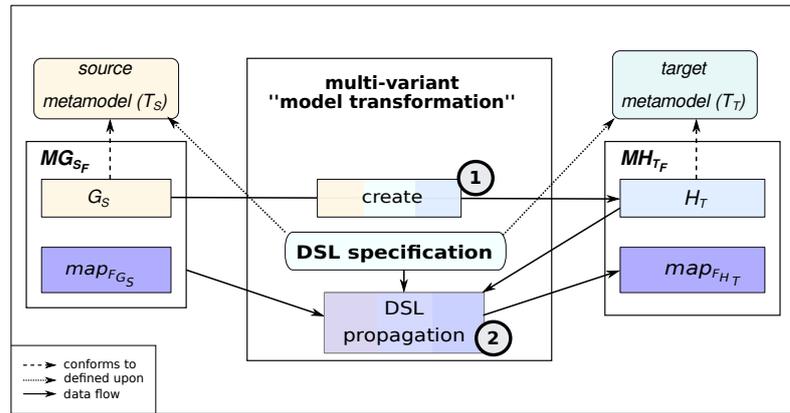


Figure 7.4.1: Schematic overview of propagation DSL-based propagation.

propagation mechanism and not the task of the propagation mechanism⁵. Moreover, defining correspondences based on the metamodels, offers the benefit that the specification can be reused for every instance conforming to both metamodels. Thus, the DSL approach pays off in situation where multiple instances of the source metamodel require a propagation and a model transformation engine is not accessible or other persistent information about deriving target elements from corresponding source elements is not available.

II DSL Design Decisions

As a consequence from the functionality, different design possibilities exist to define the syntax and semantics for the DSL. Since the DSL specifies corresponding elements of the source and target metamodel, the design decisions are similar to classifying properties of model transformations (Sec. 2.2.1). Fig. 7.4.2 summarizes important design criteria for a propagation DSL which the following paragraphs explain from left to right.

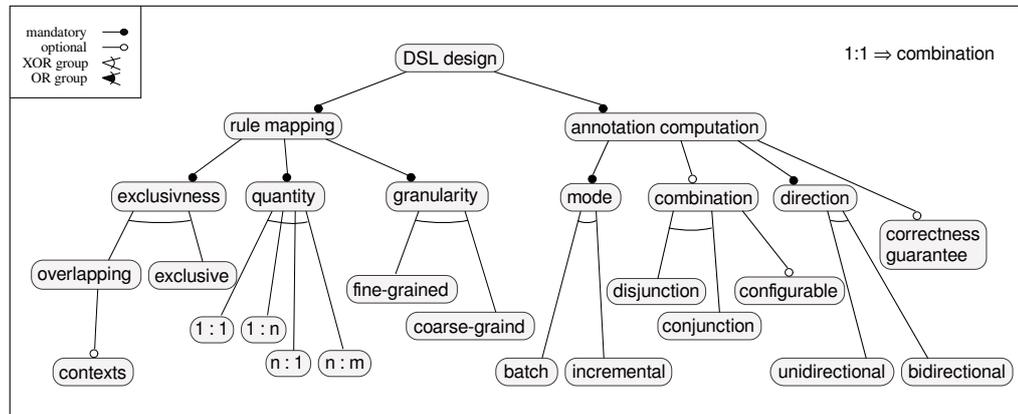


Figure 7.4.2: Feature-based classification of designing propagation DSLs.

Rule Mapping The first major design criterion of a propagation DSL, as depicted in Fig. 7.4.2, regards the type and structure of the *rule mapping* to define corresponding elements in the metamodels.

Similar to the completeness criterion for traces and the quantity definition for model transformations, the *quantity* of a rule mapping may vary. First, either one source element may be mapped onto one target element or onto multiple target elements, i.e., specifying 1:1 or 1:n mappings.

⁵ The DSL may reflect model transformation rules which could create the target model.

Similarly, multiple source element may be mapped onto one target or multiple target elements, yielding m:1 or m:n mappings.

Secondly, the mappings may be further distinguished by the *exclusiveness* criterion which allows for specifying the same target element in multiple mapping declarations (*overlapping target sets*) or exclusively in one rule. This property will be essential if only 1:1 mappings can be declared: if the mapping is not overlapping, all source and target elements might be specified only once. Mapping two source elements onto the same target type, or vice versa, however, may be necessary if no appropriate representative exists in the opposite model. For instance, the metaclass `UnlimitedNatural` of the UML metamodel has no straight-forward (1:1) representative in the Java model and may, thus, be mapped onto the primitive type `long`.

Moreover, if overlapping mappings are allowed, it might be beneficial to declare *context* elements explicitly, i.e., a target type that corresponds with another source type (being created for that source element) but which is necessary to create the actual target element.

Mapping Granularity Thirdly, the *granularity* of the mapping may vary. In the simplest form, in terms of the Graph formalism, only source nodes are mapped onto target nodes (*coarse-grained*). A more fine-grained mapping may map also edges and attributes of nodes onto each other (*fine-grained*).

Annotation Computation As a second major design decision, the *computation* of annotations may vary. Similar to a model transformation, the propagation *mode* may vary and may work only in a batch or in an incremental way. An incremental propagation would consider already existing annotations on the target model and detect conflicts whereas a batch propagation ignores already existing annotations and overwrites them with the one of the source element. Similarly, the direction property may support a bidirectional propagation or a unidirectional propagation only. As another design decision of the DSL, the way the annotation of source (and context) elements as well as annotations, assigned previously in case of an incremental propagation, should be combined may vary and should be configurable. If it is not configurable either a disjunction or conjunction of the annotations of source elements may be computed. A design decision for the category *combination* will be required only, if the DSL supports 1:n (n:1) or n:m mappings.

Generally, depending on the design and the underlying interpretation, it may be possible to give correctness guarantees for the computed annotations.

Correctness Criterion Without Model Transformations For defining the correctness of the annotation propagation, so far we use the *commutativity of model transformations* (c.f., Fig. 1.2.1). However, black-box approaches do not assume that a model transformation creates the target model or is accessible. For that reason, without an available model transformation it is necessary to define correctness criterion based on the available artifacts:

In the first place, even though traces are not available a model transformation may create the target model, nevertheless. In this case, the commutativity criterion, as it is defined by Salay et al. [Sal+14], can be used as guarantee, despite the missing trace: The transformation result of the single-variant transformation of a derived source product can be compared with the derived target product for each feature configuration.

In contrast, if a model transformation is unavailable, the first step for guaranteeing a reasonable target mapping annotation function can check the function for completeness. If annotations are missing, one of the completion algorithms presented in Sec. 7.3.3 can be employed.

Furthermore, equivalence of two models, which are instances of different metamodels, may be determined by a (generic) equivalence operator. Fig. 7.4.3 illustrates the commutativity criterion when the multi-variant target model is created manually and a DSL propagates the annotations. Then, the only information at hand is the propagation DSL and no automated mechanism generates the target model given a single-variant source model. Therefore, an equivalence operator between the single-variant source and target models is necessary. A model matching mechanism for instances of different metamodels may determine this equivalence by relying on the stable information of corresponding metamodel elements specified in the DSL. Sec. 7.4.3 elaborates on matching these instance.

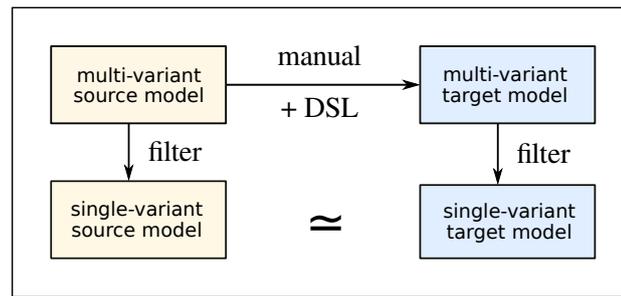


Figure 7.4.3: Commutativity without model transformations.

Summing it up, without an available model transformation the correctness criterion either fosters weak properties, such as the completeness of the mapping annotation function, or requires to match models corresponding to different metamodels.

III ModelSync

Overview **ModelSync** is a DSL developed with Xtext [Bet16] which allows declaring the corresponding elements of two different metamodels. An interpreter reads the specification to propagate annotations from a mapping model for the source domain model to a mapping model for the target domain model. The grammar of this textual propagation language resembles the syntax of the ATL model transformation language and maps one source onto multiple target elements in a declarative and unidirectional way. It allows for declaring guards, which are Boolean expressions based on the structural features of the declared target type, and further conditions over the structural features of objects to define matching source and target objects. In contrast to ATL, ModelSync does not employ procedural elements, such as `do`-blocks, but only declarative mappings. Due to declaring relationships at the level of two metamodels, the script can be employed to propagate annotations from any source to any target instance of these metamodels.

Grammar Listing 7.4.1 presents excerpts of the ModelSync grammar specified in Xtext: After having declared the source and target metamodels and provided a unique name in the specification (left out from the listing), a *propagation rule* (Lines 2-5) carries an ID and encompasses one mapping. A mapping (Lines 7-9) links one source element (which can be defined more specifically by a guard) with up to many target elements. *Conditions* (Lines 18-25) further constrain, which target elements match the propagation rule.

One specific feature of the DSL allows for matching alternative values for the value of a structural feature in conditions. This may be beneficial, for instance, if a name varies with the type of the source elements. For example, the access method for a class field either starts with the prefix `get...` or `is...` depending on the fact whether the field is of an arbitrary type or of a Boolean type, respectively. Moreover, in the current state, the DSL supports different operations on strings which can declare conditions more specifically.

Interpreter Given an annotated source mapping model and a target mapping model (without annotations), an interpreter computes matching source and target elements. The interpreter iterates the set of rule to initially determine all target objects matching the declared target types in the rule. Next, it iterates the source model to determine for each source element which of the target elements passes the guards and matches the conditions. The matching ones are collected in a final match. Note: A preprocessing step computes all target candidates of the input multi-variant model for each rule. Since the DSL does not validate the uniqueness of a target element in a rule, a target element might occur in several matches in general.

Based on the resulting sets of matching source and target elements, the propagation mechanism iterates the source mapping model and determines for each of the matching source element the set of target elements and their corresponding mapping elements in the target model. If the source

object is annotated, the propagation algorithm of the interpreter will map the annotation of the source object onto the mappings of all matching target objects.

```

1  ...
2  PropagationRule:
3    'rule' name=ID '{'
4    mapping=Mapping
5    '}'
6
7  Mapping:
8    'source' sourceElement=ElementDef guard=Guard?
9    (targets+=Target)+;
10
11 Target:
12   ('target' (targetElement=ElementDef)
13   '{' (conditions+=Condition)* '}');
14
15 Guard:
16   {Guard} '(' (guardOperation=GuardExpression)? ')';
17
18 Condition:
19   target=FeatureAccess '='
20   // Mapping
21   (sourceLeft=STRING '+')? source=FeatureAccess (':' operation=Operation)? ('+'
22     sourceRight=STRING)?
23
24   // Alternative mapping
25   ('|' (altSourceLeft=STRING '+')? altSource=FeatureAccess (':'
26     altOperation=Operation)? ('+' altSourceRight=STRING)?)
27   ';';
28
29 FeatureAccess:
30   ElementRef ({FeatureAccess.ref=current} "." right=[Feature])*;
31
32 ElementRef returns Reference:
33   {ElementRef} type=[Element];
34
35 Feature:
36   Attribute | Reference;
37
38 Operation:
39   StringOperation;
40
41 ...

```

Listing 7.4.1: Excerpt of the ModelSync grammar specified in Xtext.

Example The following example demonstrates excerpts of how a ModelSync specification maps types of the UML class metamodel onto corresponding ones in the Java MoDisco metamodel .

Example 7.4.1: ModelSync Specification

Listing 7.4.2 presents an excerpt of mapping the UML metamodel onto the Java metamodel with ModelSync. First, each specification imports the source and target metamodels (Line 1 and 2). Thereafter, rules declare a *source* and up to multiple *target* elements. The excerpt sketches two mapping rules: *Class2Class* lists corresponding target elements for a UML class and *Property2Field* corresponding elements for a UML property. The rule for mapping UML classes illustrates that multiple target types (*ClassDeclaration* and *CompilationUnit*) can be declared for one source type. Furthermore, its attributes (c.f., Line 8) and attributes of linked elements (c.f., Line 9) can be mapped onto each other

in form of further conditions. The interpreter searches an adequate element in the target model which matches these conditions and assigns the annotation from the source object to the matched target objects. For instance, the name of the compilation unit must be composed of the UML class name followed by the suffix `.java` to be accepted as match and to receive the annotation of the UML class name.

Secondly, the rule mapping a property onto a field declaration illustrates *guards* for source elements (c.f., Line 17). Only single-valued properties match this rule and, thus, only their annotations are propagated to the declared correspondences.

```

1 importMetaModel "http://www.eclipse.org/uml2/5.0.0/UML"
2 importMetaModel "http://www.eclipse.org/MoDisco/Java/0.2.incubation/java"
3
4 ...
5 rule Class2Class {
6   source umlC : Class
7   target javaCD : ClassDeclaration {
8     javaCD.name = umlC.name;
9     javaCD.package.name = umlC.package.name;
10  }
11  target javaCU : CompilationUnit {
12    javaCU.name = umlC.name + ".java";
13  }
14 }
15 ...
16 rule Property2FieldSingle {
17   source umlProp : Property (umlProp.upper == 1)
18   target javaFD : FieldDeclaration {
19     javaFD.type.type = umlProp.type;
20     javaFD.fragments.name = umlProp.name;
21  }
22   target javaSetter : MethodDeclaration {
23     javaSetter.name = "set" + umlProp.name.toUpperFirst();
24  }
25 }

```

Listing 7.4.2: Excerpt of a ModelSync specification defining correspondences between the UML and Java metamodel.

Design Decisions Altogether, based on this example together with the descriptions of ModelSync, we can draw the following conclusions on the design decisions of the DSL:

Regarding the *quantity* and *granularity*, the ModelSync DSL can specify 1:n mappings at a fine-grained level. One source element, which can be specified at the granularity of its attributes by exploiting guards, can be declared as corresponding to multiple target elements. Although the grammar and validation of ModelSync does not forbid the usage of context elements, they are not considered for the computation of annotations.

Regarding the *computation of annotations*, the ModelSync interpreter applies a unidirectional copy approach: If a source element carries an annotation, the interpreter assigns the annotation to all target elements matching the conditions. Thus, it is a batch mode execution without an option to combine or configure the computation of (more complex) annotations. Furthermore, the DSL does not guarantee any properties. Source elements can be mapped in multiple rules and target elements may match multiple rules. However, once a match has been determined for a target element, the element will be excluded from further matching and thus receives the annotation of the first source element of the corresponding matching rule.

Consequences As a consequence of the design of ModelSync, the product line developer has to define a specification for each pair of instances of distinct metamodels. However, due to the 1:n design, the DSL may more easily support refinement- relationships where the source model is extended than relationships between source and target model that decrease the number of

source elements (requiring n:1 mappings). Thus, if the source and target metamodel comprised n:1 relationships, the propagation of annotations could provoke inconsistent annotations. Moreover, the underlying DSL interpreter expects two F2DMM (Feature To Domain Mapping Model) of the MDPLE tool Famile (c.f., Sec. 9.1.2). It assigns annotations to the one designated as target model. Consequently, it is trimmed to solve exactly this task and is tightly intertwined with the development of a product line in in this tool. For different mapping mechanisms it cannot be used out-of-the-box.

IV Discussion

To conclude, we reflect the pros and cons of a propagation DSL, starting with a delimitation and the categorization of the approach with respect to propagating annotations and ending with a discussion of the automation capabilities. Moreover, since the ModelSync specification resembles declarative model transformations to a great extent, we compare ModelSync with model transformation approaches. At the end, this part discusses the automation degree of a propagation DSL.

Delimitation As delimiting factors, first of all, it must be mentioned that the target model already exists and information how it was created is not available due to the black-box nature. Consequently, in contrast to the variations of trace-based propagation, the propagation DSL does not rely on a model transformation meaning that the target model may also have been created manually. This implies that correctness in form of commutativity cannot be defined over model transformations but an equivalence operator or a weaker correctness condition is necessary. To this end, based on the presented properties, a propagation DSL represents a language-independent, specification-specific, black-box post-processing approach for propagating annotations.

Comparison With Model Transformation Specification In general, the ModelSync DSL reflects a declarative model transformation specification with many respects: It requires specifying corresponding elements of the source and target model and can map their attributes and references onto each other. In contrast to a transformation specification and the corresponding engine, the propagation mechanism does not create target elements but assumes that they already exist. Moreover, the DSL engine will not report errors, if a source element misses target elements but simply will not recognize a match. Finally, it creates a mapping model used in the MDPLE tool Famile and, thus, is a tool-dependent solution.

Discussion on Automation Employing a propagation DSL to declare corresponding elements of two metamodels offers two main advantages: On the one hand, similar to a model transformation specification, a DSL specification works for any pair of models conforming to the source and target metamodel it is defined for. On the other hand, the approach works completely independent of a transformation engine and, thus, is per se tool- and language-independent. As a consequence, it does not restrict the transformation language or the mechanism that creates the target model. One major disadvantage is the missing automation due to the *manual* specification of the mappings in a concrete specification. However, one automation possibility may compare the source and target model, as discussed in Sec. 7.4.3, to determine matching source and target elements.

7.4.3 Trace Generation by Model Matching

Despite the upside of defining a correspondence specification and the automated propagation of annotations to the target model, on the downside, the propagation DSL has to be specified for each new pair of metamodels *manually*. In contrast, matching the source and target model to generate a propagation DSL or to re-engineer trace information increases the automation. Grammel et al. [GKV12] propose an automatic trace generation approach relying on matching models when no out-place transformation is available which we examined as automation technique and present in this section.

Road Map To keep this thesis self-contained, the following parts introduces research on the topic of *model* and *metamodel matching* first. Afterwards, they present a matching framework proposed by Grammel et al. [GKV12] which offers a solution to extract trace information by comparing two models. The final two parts sketches how the matching concepts can be exploited to propagate annotations and discuss these solutions, respectively.

I State-Of-The-Art in Model Matching Approaches

When multiple versions of models exist due to editing a model over the course of time, for instance, a version control system may want to determine modifications of model elements in terms of differences for efficient versioning. Since falling back to *textual comparisons* of parsed model representations (e.g., in XML format) does not suffice due to a lack of semantic context information [FLW11; Wes14], it is necessary to apply *model-specific* matching mechanisms. The following paragraphs shed light on various existing matching strategies proposed in previous related work, the result of which this thesis may use to propagate annotations. Additionally, it introduces the model matching framework EMFCompare, provided as Eclipse plugin, which supports several of the presented matching categories.

Classification Categories Kolovos et al. [Kol+09] distinguish four types of model matching approaches: *static identity-based*, *signature-based* and *similarity-based* matching as well as *custom language-specific* matching. These four categories lay the foundations for further subsequent and contemporary classifications of model matching approaches [ASW09; Bro+12; Uhr11; BPV10; Gra14; SA20]. Besides these forms of state-based matching of models *operation-based matching* compares edit operations. Furthermore, “out-place matching” of models which are instances of different metamodels and matching metamodels themselves represent specific categories which require sophisticated techniques.

Static Identity-Based In the first category, a static identifier, such as a UUID (universal unique identifier), allows for identifying an evolving object across versions of a model. As a consequence, this approach may only be applicable if the same model is versioned and a mechanism assigning unique identifiers is available by the modeling tool. Moreover, if two independently created models are compared, this approach will not work because even if the objects of both models are structurally equivalent, their identifiers may be different.

Signature-Based matching tries to alleviate the shortcoming of static identification (requiring universal identifiers) by computing an identity value based on user-defined functions. The function may regard the structural features of an object and the algorithm compares the computed values. The downside of this approach is that the user needs to provide the functions that allow for computing the identity of an object which has to be unique for each object in one model.

Similarity-Based In contrast to static identity- and signature-based matching, which recognizes a match if and only if the identities of objects are the same, similarity-based matching computes the similarity based on (some of) the objects’ features. If the computed similarity for two objects exceeds a (configurable) threshold, two objects will be considered similar. This approach allows for weighing features differently and typically achieves better matching results than a solely identity-based mapping. The algorithms performed on *typed attributed graphs (TAG)* [EPT04] in these approaches achieve more accurate matching results than identity-based ones [Kol+09]. Algorithms and entire frameworks supporting this mechanism only are, for instance, the similarity flooding algorithm [MGR02], SiDiff [SG08], DSMDiff [LGJ07] as well as a trace-generating framework explained in Sec. 7.4.3, II.

Modeling Language-Specific Matching While the three aforementioned categories can be applied to any type of metamodel (general applicable), custom language-specific approaches are trimmed for a modeling language (e.g., UML). Accordingly, they can specify concrete matching properties and metrics to determine an accurate matching result taking the model specifics into

account for the matching. As an example, Uhrig defines editing costs between two graphs representing class models for determining a matching [Uhr11] and several solutions focus on comparing UML models [KWN05; MRR11a; MRR11b] or Ecore models [201; KH10].

EMFCompare [TI06; BP08] serves as one example of a configurable and customizable model matching framework to compare EMF-based models. It may apply a static identity-based comparison if UUIDs are available, or a similarity-based matching [Add+16] otherwise. For computing the similarity, EMFCompare considers the name, the type (i.e., the features of the meta-element), the remaining attribute values and relationships (regarding containment and non-containment references) in order to heuristically match the elements of two models. Furthermore, it can be customized to compare instances of a specific metamodel due to an open API and is developed with efficiency to compare large model, composed of thousands of elements. According to Grammel [Gra14], however, there is a lack of accuracy when EMFCompare is used to compare models conforming to different metamodels. Sec. 10.2.1 provides more detail on the technical specifics of this matching framework which is used as part of the evaluation of commutativity.

Operation-Based Besides these four classical categories which realize *state-based* comparisons, a monitoring system may record edit operations applied to the model and may use them to compute differences between two versions of a model, yielding an *operation-based* matching [Her10; KKT13]. The latter, however, requires an edit recording mechanism and, thus, is not as generically applicable as a state-based matching. Due to the capabilities of an operation-based matching system, a recent systematic literature survey of Somogyi and Asztalos [SA20] extends the basic four categories of matching algorithms with the categories of using *none* of the four classical categories, which is the case in operation-based techniques, or being *configurable* [BPV10; Alt+08], i.e., multiple possibilities can be exclusively chosen, in a staged way or in a combined way. Moreover, model matching may not only employ a textual or a Graph-based comparison but a combination thereof [RV08]. In addition, the matching algorithms vary with respect to the type of models that is compared, being either structural or behavioral models. Somogyi and Asztalos [SA20] state that most of the matching algorithms exhibit low accuracy when comparing behavioral models.

Out-Place Model Matching Typically most of the model matching algorithms, particularly those designed for versioning models, compare one type of (evolving) model. Although metamodels may evolve, too [Wac07; PMR16], the matching mechanisms compute differences between two versions of a model and, therefore, match two instances of the same metamodel.

For employing model matching algorithms to automatically determine relationships between a source and target model of an exogenous transformation, however, a matching between two models *conforming to distinct* metamodels is required. As opposed to matching an evolving model to detect differences, matching two instances of different metamodels is more complex: Technically,

- the types of objects do not match,
- the identifiers do not match and, more importantly,
- it may not necessarily be 1:1 matches but an element in one model may correspond with zero to up to multiple elements in the second model.

Due to this complexity, retrieving correspondences between two models conforming to different metamodels may require even more sophisticated matching algorithms than for comparing instances of the same metamodel. The next section illuminates the capabilities of matching framework dedicated to derive trace information.

Metamodel Matching Matching two metamodels can be considered a special case of model matching and exhibits a similar complexity. While both metamodels may be instances of the same meta-metamodel, they do not have to share the same concepts if the two compared models are not versions of an evolving metamodel. Metamodels, in particular, may exhibit *structural* (i.e., same concept represented with different constructs) and *syntactic heterogeneity* (i.e., same

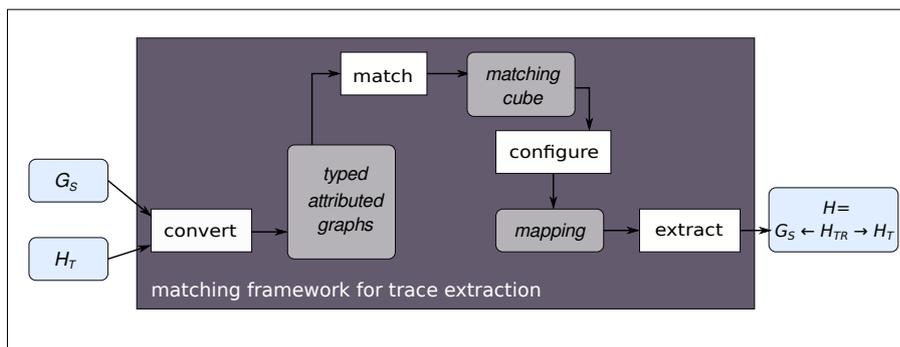


Figure 7.4.4: Model matching framework for generating trace links based on [GKV12].

construct named differently) as well as differences in the *formal language* (e.g., hierarchical vs. relational) [Wim+11a]. For the specific purpose of matching metamodels, which may be required, for instance, to infer a model transformation, additional approaches have been proposed besides the four categories of Kolovos et al. [Kol+09]. Besides other approaches, which have been implemented particularly for the purposes of deducing model transformations [FL19; LF20; VIR10] by applying different deterministic techniques, heuristic approaches are a different mechanism to narrow down the search space for computing matching source and target elements.

For example, one search-based mechanism [Kes+14] employs genetic algorithms based on simulated annealing [KGV83] which determine a solution and examine the local search space for an optimum. The function which the search mechanism optimizes regards structural and syntactic similarity. This approach achieves the highest accuracy for matching distinct metamodels which resulted from evolution. It is able to outperform each of the examined deterministic approaches, particularly, in determining n:n mappings and matching elements with different names because it explores the search space only partially.

Summary All in all, the previous paragraphs demonstrate that different solutions to detect similarities in between two models exist. Deterministic approaches may rely on syntactic and structural similarities and can specify metamodel-specific rules. Heuristic approaches, in contrast, allows for computing optimized solutions by exploring the search space efficiently. As a sole syntactic comparison of elements may not suffice to determine matching elements in two models, the next section illuminates an approach which was developed specifically to reconstruct trace information for models conforming to two distinct metamodels.

II Matching Framework for Trace-Generation

In the following we describe an approach that explicitly targets the comparison of models conforming to different metamodels. Grammel et al. [GKV12] propose a matching framework to retrieve trace links when a model transformation is not available. In contrast to the work in this thesis, the purpose of their work is not to propagate annotations in multi-variant models but to establish trace-links for multiple purposes in MDSE: comprehension of the system, analysis of change impacts and source model coverage as well as for debugging a transformation [RJ01; OO07].

The proposed matching and extraction process constitutes as depicted in Fig. 7.4.4. In the figure, we adapt the input (G_S , H_T) and output (H) of the matching framework to be represented as in the Graph formalism established in Chp. 6. A (multi-variant) source model as well as a (multi-variant) target model without mapping functions are input to the matching framework. The graph H_{TR} constitutes the mapping that is computed which offers links between the multi-variant source and target model to complete H in its entirety. Next, we describe the single steps performed in the matching process.

Convert In the first step of computing matches, the framework converts the source and target model into *typed attributed graphs* (**TAG**) to represent them in a normalized data structure which

the algorithm can compare. This formalism considers metamodels as *attributed typed graphs* (**ATG**), which are also input to the conversion step, and their instances as *attributed graphs* (**AG**). Due to an additional mapping to the metamodel, a TAG is a special kind of AG. The proposed approach assumes that the metamodels are already matched based on similarity of the attributes and nodes [Gra14] and the resulting matching serves as additional input to the matching algorithm.

AGs comprise two types of nodes: *data nodes* and *graph nodes*. While a graph node represents a concrete type of the model (i.e., an instance of a metamodel type) an attribute node carries an attribute value, e.g., the name of the type. Similarly, three types of edges exist: *graph edges*, *node attribute edges* and *edge attribute edges*, representing either a link between graph nodes, a connection to a data node originating from a graph node or from a graph edge, respectively.

Match Based on the representation as TAG, the similarity-based *match* step computes similarities for each pair of graph nodes. As similarity measures, the algorithm either employs the *attributes* solely, the attributes together with the *connections* which consider the outgoing graph edges to other graph nodes, or an *instance-of* measure exploiting the type information extracted from mapping the AG onto its metamodel graph (i.e., an ATG). The outcome of comparing nodes pairwise is a similarity cube which comprises a *matrix* of node pairs for each measured similarity type (i.e., attributes, connections and instance-of).

The matching based on the attribute-similarity of a pair of source and target nodes compares the values of all data nodes of the source node with the values of all data nodes of the target node. For measuring the similarity of two data nodes, Grammel [Gra14] consider String comparisons using a function which yields a result of 1 if the Strings in the data nodes are the same, of 0.5 if one is the Substring of the opposite node and of 0 if there is no match [GKV12]. According to the corresponding dissertation [Gra14], for determining the matching, the approach exploits the functionalities of the MatchBox framework [VIR10] additionally. Therefore, the framework compares Strings either on the basis of building *Trigramms* as a specific form of *n-gram* [MS99] or the Levensthein distance. This yields a more accurate result than relying on the pure Substring function: For example, the Strings `PersonDatabase` and `FamilyDatabase` are neither equal nor Substrings of one another but share a common subsequence. Accordingly, the basic three-valued comparison [GKV12] would compute a similarity of 0 whereas a Trigramm or *Longest Common Subsequence* [HS77] detects a higher similarity.

Non-String Datatypes In reality, however, not only Strings constitute the data of a graph node but different primitive types, such as Boolean values or numeric values (e.g., doubles or integers). Therefore, one possibility may represent these values as Strings as well by potentially losing semantic-related accuracy. For example, the double values 1.5 and 15.0 may yield a higher similarity than 1.5 and 1.6 when being matched as String whereas in absolute numbers 1.5 and 1.6 are related more closely. Alternatively, the comparison needs to employ customized functions for computing similarities of primitive types apart from Strings being stored in data nodes.

Connectivity Similarity For employing the connection measure, it is essential to compute the attribute values first. Otherwise the children nodes cannot be compared because the comparison is based on attribute values. Then, MatchBox [VIR10] matches parent, children, sibling and leaf nodes or employs a graph editing distance or the matching of predefined patterns.

Instance-Of Similarity For the instance-of measure, a prematch of the two accompanying metamodels is necessary. Both metamodels can be matched in a similar way, as the model but in contrast share a common metamodel, in our use cases the Ecore meta-metamodel. This similarity, however, is not used for matching the attributes or connections per se.

Data Matrix Reduction After a pair-wise comparison of the data nodes, resulting in a *data matrix* for each pair and similarity measure, this matrix is reduced to a single value which is associated as the respective similarity measure with the two graph nodes and put as entry into

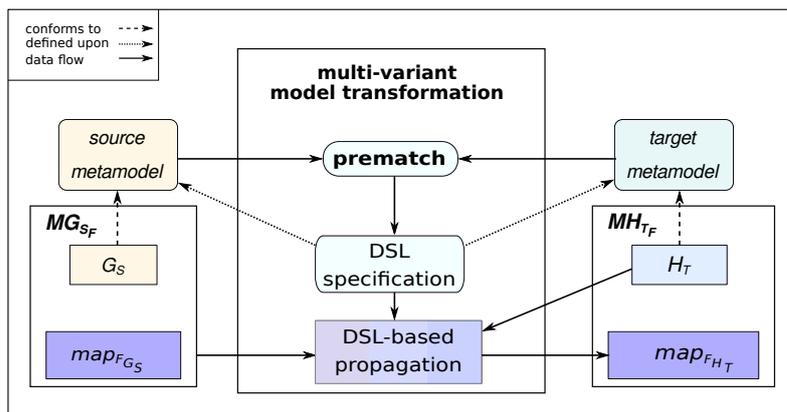


Figure 7.4.5: Schematic overview of *DSL-based* propagation by metamodel matching.

the similarity matrix for the graph nodes. Since multiple similarity measures (i.e., attributes, connections, instance-of) may be computed, a *similarity matrix* is kept for each measure and, thus, results in a *similarity cube*.

Configure The result of the matching step is a similarity cube which lists for each pair of graph nodes the reduced computed similarity measures. In the configuration step, one out of (at least) three strategies can be chosen. Either the similarity measure per node pair can be *aggregated*, *selected* or a ranking can be iterated in one *direction*.

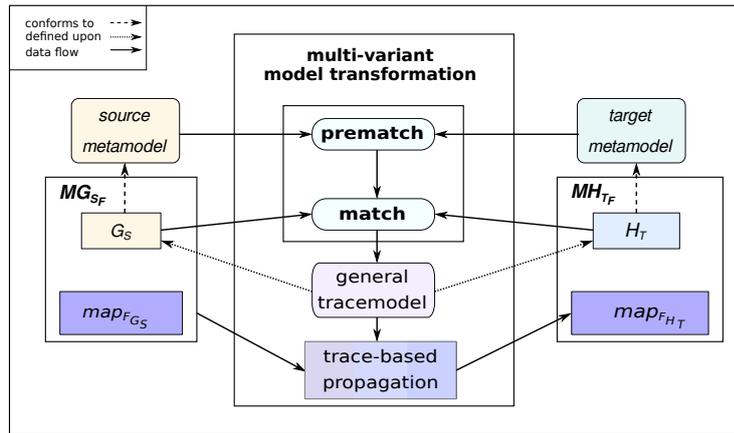
For the aggregation [Gra14], either the *maximal* or *minimal* similarity value can be chosen to declare a pair of nodes as similar, or the results can be *weighted* or the *average* can be computed. The selection either retrieves all matches above a threshold, takes the first N highest ranking matches or defines a delta after the values have been aggregated. Finally, the ranking of values can be iterated in forward or backward direction or a combination thereof. As a consequence, it is not only 1:1 mappings but *1:n mappings* can be retrieved from the similarity cube depending on the threshold configuration.

Extract Lastly, the trace-generating framework extracts links between the two input models. In our formalism the trace sub-graph H_{TR} needs to result as part of a source-to-target graph incorporating G_S as source and H_T as target sub-graphs and with established links between both. In contrast to the source-to-target graphs defined in Def. 6.3.5, this algorithm is unaware of *context* nodes and the resulting trace granularity and completeness depends on the preceding *configuration* step. However, depending on the similarity threshold, it is possible to map multiple elements onto one source node which is in contrast to matching algorithms which compare models conforming to the same metamodel.

III Schematic Overview of Matching-Based Propagation

After having demonstrated how to refactor trace information from model matching, this part sketches two possibilities how the result of the model matching can enrich the propagation of annotations when trace information is missing. On the one hand, a matching between the metamodels *only* may be computed or, on the other hand, the metamodels *and* the models can be matched.

Metamodel-Matching Based Firstly, Fig. 7.4.5 demonstrates how (meta)model matching can automate the propagation of annotations by extracting the DSL specification (Sec. 7.4.2). Since we assume that the source and target model are instances of metamodels which are in turn models conforming to the Ecore meta-metamodel, it is possible to apply the model matching algorithm presented in the previous part (or another metamodel matching approach) of this section to the metamodels in the *prematch* step. As a result from this matching, correspondences between

Figure 7.4.6: Schematic overview of *trace-based propagation* by model matching.

the source and the target metamodel are available. Since a propagation DSL as discussed in Sec. 7.4.2 defines correspondences at the meta-level, the result of matching metamodels may be transformed into such DSL specification. Consequently, we can construct a mapping rule in the DSL specification for each pair of matched source and target graph nodes and feed that script into the DSL parser together with the source model and its mapping as well as the target model. As a result, the mapping for the target model is created automatically in the same way as the DSL specification.

Model-Matching Based Secondly, Fig. 7.4.6 demonstrates the usage of a matching framework to propagate annotations to the target model with a trace-based propagation algorithm. Instead of comparing only the metamodels and employing the DSL propagation mechanism, the trace-based propagation can be used. Consequently, besides computing the prematch, which improves the matching result due to employing the *instance-of* similarity measure in addition, the concrete source and target model, G_S and H_T are matched. To use trace-based propagation as explained in Sec. 5.2.3, the resulting trace information of the *match* step needs to be transformed into the common trace model. Then, Alg. 1 can create the mapping for the target model, however, without guaranteeing commutativity due to missing guarantees on the completeness and granularity of the matching result.

IV Discussion

This section reflects, on the one hand, on the *automation* degree of employing a model matching framework for extracting trace information as well as on the expected *accuracy*, on the other hand.

Automation In contrast to defining a DSL specification manually, the benefit of metamodel matching is to automatically extract the DSL specification based on the matching result. However, it is essential that the matching is accurate (which is claimed by the designers [VIR10; Gra14; Kes+14]). If the matching is inaccurate, wrong annotations will be propagated to target elements. Similarly, if the source and target model are matched instead, and can be turned into the common trace model (c.f., Sec. 5.2.3), the automated trace-based annotation propagation may be used as well. Even if not all target elements were matched, the completion algorithms for partially annotated models (Sec. 7.3.3) could be used in a postprocessing step.

Consequently, matching the metamodels or their instances represents a *fully automated* annotation propagation.

Accuracy Due to the configuration possibilities and the dependency on model structures in the deterministic and heuristic (meta)mode matching processes, the correctness and completeness of the computed correspondences can not be guaranteed. Therefore, the resulting matching is an approximation. It is not possible to detect context elements with any of the examined matching

approaches (c.f., Sec. 7.4.3, I and Sec. 7.4.3, II). Particularly, the matching framework of Grammel et al. [GKV12] does not consider the explicit extraction of context elements despite the main objective to generate trace information. When allowing to match a target node multiple times, this node may be mapped onto multiple source nodes without clarification whether it has to be considered a context node or a target node. Without that information, the annotation for a target node, mapped onto multiple source nodes, may be overwritten when propagating annotations and the last assigned annotation remains. If the matching algorithm extracts only 1:1 mappings instead, the extracted trace will be incomplete requiring to execute the propagation for partially annotated models (c.f., Sec. 7.3.3) to complete the annotations automatically. The accuracy of completing annotations is discussed in Sec. 7.4.3, II.

Summary In summary, the discussion illustrates that matching models or their metamodels will both allow to completely automatically propagate annotations to the target model. However, due to the nature of the matching algorithms, correctness in form of commutativity can hardly be guaranteed and depends on the granularity and completeness of the matching.

7.5 Incremental Annotation Propagation

As stated in the introduction to this chapter, some solutions for propagating annotations cannot guarantee commutativity when (fine-grained) information about corresponding elements of the source and target model is missing in the trace. Particularly, the annotations which are determined by a completion strategy may be too broad such that the element remains in too many configurations. Accordingly, a product line developer may fix those annotations manually in a post-processing step. To maintain manually modified annotations, for instance in iterative development scenarios, this section discusses how to maintain already existing annotations of the target model in an incremental annotation propagation.

Road Map For that reason, Sec. 7.5.1 describes the problem which occurs when maintaining existing target annotations manually firstly. Secondly, Sec. 7.5.2 discusses evolution scenarios of model-driven product line in order to delimit the incremental propagation strategy. For propagating annotations incrementally, a mechanism to distinguish manually assigned from automatically computed annotations is necessary. Sec. 7.5.3 presents two ways to accomplish this task in its first part and describes the corresponding propagation algorithms in its second part. In the end, Sec. 7.5.4 discusses the proposed strategies.

7.5.1 Problem Statement

Sec. 7.3 and Sec. 7.4 show that an automatic approach to annotate the target model completely may not assign annotations that guarantee 100% correctness in case the trace information is incomplete or in case the trace information resides at a more coarse-grained level than the annotation mapping function.

For instance the examples in Fig. 7.3.5 and Fig. 7.3.6 illustrate that the annotations computed by the three completion strategies, are mostly too broad. Due to the hierarchical strategies, particularly, the annotations mapped onto leaf nodes hardly reside at the right level of specificity. Thus, the model filter pertains the corresponding model nodes in too many filtered variants. As a consequence, the product line developer may change these annotations and assign **semantically correct** ones **manually**.

Example Fig. 7.5.1 demonstrates the result of the annotation propagation based on an incomplete trace. The annotations situated in the rectangles of light-purple color are those which are mapped onto the target elements based on the trace information. The annotations which are crossed out are those assigned by the *combined* completion strategy (c.f., Sec. 7.3.3). The manually repaired annotations reside next to the crossed out annotations in blue rounded rectangles.

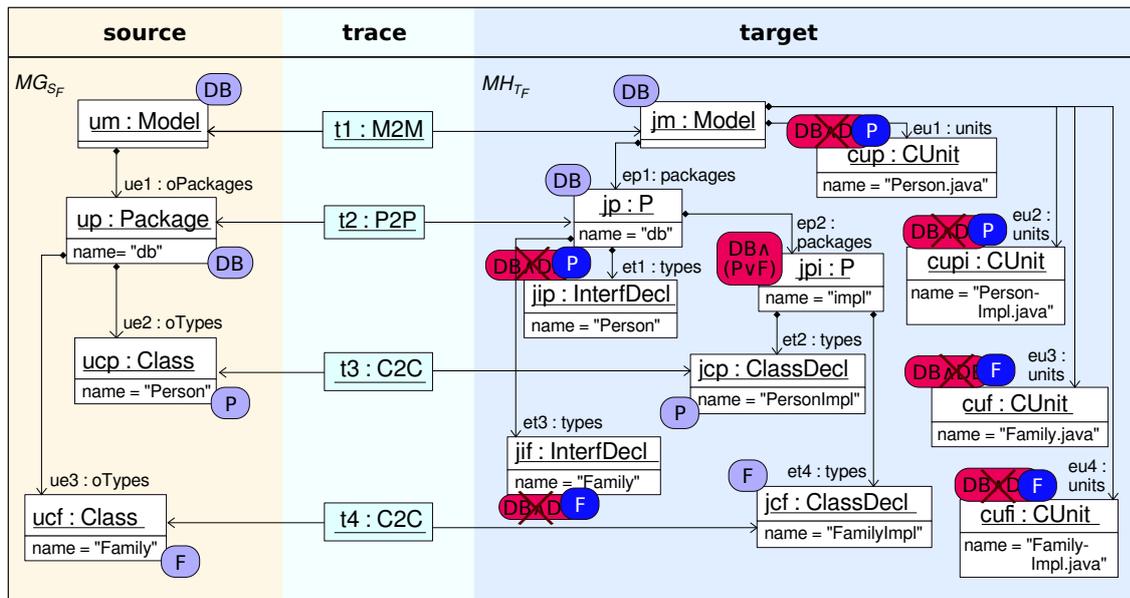


Figure 7.5.1: Manually repaired broad annotations.

Accordingly, except for the annotation of the implementation package, which carries the semantically correct annotation $DB \wedge (P \vee F)$, the developer exchanges the remaining completed annotations (e.g., those of the compilation units) by either assigning P or F for elements realizing the classes *Person* and *Family*, respectively.

Incremental Maintenance Consequently, the question arises: how are the existing target annotations affected when a product line developer executes the forward transformation another time, for instance, because a UML element was added? The execution of multi-variant model transformations, as defined so far, ignores the fact that a target model exists and, thus, considers neither existing target elements nor their annotations. Even though the reused single-variant model transformation is performed incrementally and recognizes the already existing target model, it will not be aware of annotations because the propagation algorithm is defined in batch mode. As a consequence, all annotations of the target model are *overwritten*. Furthermore, if an incomplete trace is the source of information for propagating annotations and missing annotations are computed by one of the completion strategies, the second execution of the trace-based propagation provokes the same situation: the annotations of the leaf nodes are not specific enough and violate commutativity. As a result, all manually assigned annotations (i.e., the entire invested work) are lost. In turn, this requires to repeat the manual process which increases the development cost and is laborious, unnecessarily redundant work.

Consequences For that reason, we draw the conclusion that strategies, which regard the presence of already existing annotations in incremental transformations, are indispensable to support iterative development. If the product line developer changes annotations of the target model after an automatic propagation manually, they should not be overwritten in a subsequent execution.

In addition, the example shows that it is not possible to propagate the modified annotations back to the source model since unique information about the corresponding source elements may not be available in the incomplete trace. Thus, the common trace model, which is used for the propagation, misses this information, too. Due to the fact that the trace does not record these elements, without further knowledge it is uncertain which source elements correspond with manually modified target elements. This makes it impossible to associate manually changed annotations with a corresponding source element. As a consequence, a mechanism to preserve and maintain these manual annotations in consecutive annotation propagation processes is indispensable.

7.5.2 Background

Iterative and incremental software development [LB03] is a natural process of building a software system in a step-wise way. Typically it deviates from the classical waterfall process [Roy87] (even if that one is performed in iterations) and incrementally evolves artifacts of certain development stages. In our context, we assume an incremental and iterative construction of domain models building a product line.

Road Map Accordingly, the product line may evolve and incremental transformations may propagate the changes made to source domain model to the target model. Therefore, the first part of this section illuminates background information on evolution dimensions in product line engineering and incremental model transformations. This knowledge serves to delimit the contribution of this thesis in the third part.

I Product Line Evolution Dimensions

Evolution of a product line can be parsed in different ways: Evolution can target the feature model only but changing the feature model may affect existing annotations and realization artifacts. Thus, a combination of evolving the feature model, a realization artifact and its annotation may, particularly, occur in iterative development processes. Please note: In the following we do not consider the (co-)evolution of single variants [Kir+21; Sch+16] as we focus on changes occurring at the level of domain engineering. More detail and discussions on evolving software product lines including the engineering method and product maintenance in terms of reusable assets is presented, for example, by Botterweck and Pleuss [BP14].

Feature Model Several works [TBK09; BTG12; Bür+16] focus on the evolution of the feature model which determines how the set of variants changes due to new requirements.

Borba et al. [BTG12] offer formal foundations for behavior preserving product line evolution, i.e., the functionality remains stable but the feature model evolves. A similar line of work compares two distinct feature models resulting from, for instance, *collaborative* (and *distributed*) *development* or *refactorings* [KAK08], in a state-based way. These works check whether two feature models are *semantically* equal [TBK09; Ach+12; FLW11]. If two feature model are semantically equal, they will share exactly the same set of valid feature configurations.

In contrast, feature models may also evolve as part of an *iterative development process*: While the set of products and functionality of the product line is a priori fixed, it may take several (mostly monotonic) development steps until the feature model is completely developed. For instance, variation control systems, such as SuperMod [SW16; SW19], incorporate and manage a feature model evolving in this way. Even though these systems edit single variants, this is the only means to edit the elements of the software platform.

Annotations Due to changing the feature model (without preserving behavior), the annotations mapped onto realization artifacts may (have to) change, too: If the feature model evolves by adding, changing or deleting features and their constraints, already assigned annotations may become obsolete and require to be changed as well. Modifying annotations may not only occur as a consequence of changing the feature model but may also occur in isolation. For instance, as illustrated in the motivation (Sec. 7.5.1), automatically computed annotations may be not specific enough due to missing propagation information. Therefore, they may have to be refined.

Realization Artifacts As third change type, the realization artifacts, in our case the domain models, may change. Either only one model or both models connected by a transformation can change, provoking either an update of the unmodified side or a synchronization. The latter is necessary if the latest changes should be preserved on each side and not be overwritten by a unidirectional transformation.

Refactorings, iterative development and new customer requirements may provoke the evolution of realization artifacts. Particularly, in iterative MDPLE a domain model is developed in several

development steps. The changes may be propagated to related models subsequently and may also involve synchronization tasks if multiple models are changed individually. Modifying two models requires the usage of bidirectional incremental transformation to keep both models in a synchronized state [Anj+20] whereas keeping multiple models synchronized requires even more sophisticated methods [Stü+20; Stü+21; Kla21].

Combination Typically, the three types of changes rarely occur in isolation. Changing the feature model without preserving its semantics may involve a change of annotations because they may have become obsolete, when a mentioned feature was removed, modified or added, or too broad due to changing the dependencies between features. Furthermore, changing the feature model may involve changing the realization artifacts. For instance, model elements may be added to a domain model to realize a new feature. Adding a new domain model element to the source model may also require to add a new annotation which is mapped onto that model element. Removing a domain model element may require to remove its annotation (e.g., a mapping element), too. Similarly, changing a domain model element may also provoke the change of a corresponding annotation which may have to be propagated to the target model.

Summary To sum it up, annotations, the realization artifacts, particularly the domain model in our context, and the feature model may change in isolation or jointly. Consequently, different stages of complexity have to be considered for preserving annotation consistency after an evolution step.

II Incremental Model Transformations

In the following we assume that the batch transformation which creates the multi-variant target model, can also be executed incrementally. Accordingly, it is capable to detect modifications to the input model and only propagates those to the target model. The following paragraphs present background information on incremental transformations.

Delta Types To distinguish an incremental transformation from other approaches, Anjorin et al. [Anj+20] offer a classification of synchronization scenarios: The authors classify a unidirectional incremental transformation as *directed synchronization* which prioritizes the direction of the transformation as opposed to a synchronization where changes on both sides can occur.

A *delta* describes the difference between the previous model and the model at the state of synchronization which is necessary to detect and propagate only changes in the incremental transformation. The authors distinguish *operational deltas* from *structural deltas* and from *edits*. While operational deltas comprise the sequence of modification steps yielding the new state of a model, a structural delta represents the difference by a mapping between the old and the current version. An edit encompasses a list of deltas in an ordered way which are applied to one model.

For the realization of an incremental transformation, the respective languages and the corresponding execution engines handle and employ different approaches:

Model-To-Model Transformations For performing an incremental transformation the engines need to determine the modifications in the input model. Even though model transformation may compare the new and previous input models to detect the differences between the old and new versions, the structural comparisons may not be cost-effective and not fine-grained enough. For example, if no unique identifiers are used for the comparison, changing a model element may be also recognized as a deletion followed by an addition.

Therefore, (incremental) M2M transformations frequently rely on *trace* information [Wag+12; Obj16; BBW21; Buc18] to build an incremental version such that a comparison of source model elements suffices. First of all, the source model edits are transformed and update the trace information. Adding a source element requires to add a trace element (and the corresponding target elements). In contrast, if a source element is deleted, a trace element will remain, which references target elements but no source elements. Thus, an incremental transformation execution

deletes trace elements which do not reference a source element anymore as well as the target elements referenced by the respective trace element.

Model-To-Text Transformations Incremental M2T transformations require to compare the source model with its previous version or to record edit operations which yielded the new state, similar as M2M transformations do. For instance, both, Acceleo and Xpand, compare the current and previous state of the source model in order to incrementally generate the target text.

Since in M2T transformations text is generated which may be changed by the developer, these languages employ *protected* regions to retain manually added or modified text fragments in the generated text. These regions have to be identified uniquely. For example, Xpand requires to mark them with an ID. Furthermore, they need to state an escape sequence, for instance the words **generated not**, which the text generator recognizes and does not modify anything inside the enclosed region. However, if no source element matching the block containing the enclosed region is detected, anything that was created for the source element before will be deleted. Sec. 8.2.1 offers more detail how the realization of the Mof2Text standard, Acceleo, and the template-based transformation language Xpand realize incremental transformations.

III Delimitation

Based on the background information of possible evolution scenarios in MDPLE, the following two paragraphs describe which change and transformation scenarios the following contribution solves: the incremental maintenance of annotations between a source and target model.

Annotation Preservation While different stages of complexities can be related with incremental transformations, the sequel focuses on the *evolution of annotations*. Manually repaired annotations which were too broad or too specific due to missing trace information, as sketched in the Fig. 7.5.1, should be retained in a subsequent transformation execution if the corresponding source elements have not changed. Thus, we explain the preservation of annotations which have been manually exchanged after an initial transformation and adapt the trace-based propagation algorithms accordingly. The following explanations consider unidirectional scenarios which (re-)generate the target model, which change neither the direction of the transformation nor of the annotation propagation.

Transformation and Feature Model For propagating changes of a source model to the target model, we employ a unidirectional incremental single-variant transformation which may add elements to the target model which correspond with elements added to the source model. If the target model changes (apart from its annotations), the modifications will be overwritten according to the transformation specification and the source elements. Moreover, if the reused single-variant model transformation is executed again, it must be in incremental mode so that the trace still records the same source and target elements. Otherwise all elements are new, and thus, require an annotation which would provoke an entirely new computation of the target mappings. Furthermore, if the feature model changes, the MDPLE tool needs to check whether each annotation can still be satisfied immediately after the change has occurred. Thus, the annotations which are mapped onto model elements are consistent with the feature model before propagating them.

7.5.3 Incremental Annotation Maintenance

As illustrated in the introductory example of this section, the annotations mapped onto target elements in Fig. 7.5.1 may be not specific enough as a result from missing reliable information about corresponding source and target elements. Therefore, these annotations may have to be changed manually after the automatic propagation.

Road Map For propagating annotations incrementally, the first part of this section provides the foundations for representing the different types of annotations, manually assigned and automati-

cally computed ones, and maintaining them. Based on these definitions, the second part describes two algorithms to incrementally propagate annotations.

I Preliminaries

For incrementally propagating annotations, the mapping annotation function may store different kinds of annotations. The following paragraphs provide the foundations for representing different kinds of annotations and for maintaining them in an incremental propagation.

Annotation Version When propagating annotations incrementally, up to three different *versions of annotations* may be relevant:

The *stored annotation*, a_s , has been propagated by a multi-variant model transformation at time t_n and, thus, it has been assigned automatically to a target element. At time t_{n+1} , before updating the target annotations, a_s is mapped onto the target elements. Therefore, a_s can be regarded as the *base version* of an annotation [GNS22]. This annotation may be changed in between the two executions at t_n and t_{n+1} , yielding an annotation a_m , which a developer maps onto the respective target element *manually*. Finally, a consecutive execution of the annotation propagation at time t_{n+1} *computes* an annotation a_c which should be assigned to the target element and may be the same as a_s or replace it.

Annotation Representation The storage of these annotations, if required, can be realized by extending a mapping element to distinguish the stored from the manually changed annotation. The computed annotation a_c is transient and is not stored itself but either overwrites the stored one or it is discarded. Without loss of generality, we define such two-valued mapping as follows:

Definition 7.5.1: Two-Valued Annotation Mapping Function

Let EL be the elements in graph G and A_F be the set of annotations constructed from a set of features F . The function $t\text{-map} : EL \rightarrow A_F \times (A_F \cup \{\epsilon\})$ maps up to two annotations onto a graph element, such that $t\text{-map}(el) = (a_s, a_m)$, a_s represents an automatically computed annotation and a_m an optionally added manual one.

Accordingly, for a graph element not only one annotation is stored in the mapping but a tuple consisting of the stored annotation and the optional manual one. For the presence of a model element in a filtered product, the manually assigned annotation is decisive. Whenever an annotation is mapped onto an element manually, the annotation overrides the automatically assigned one. This form of selective filter is defined as follows:

Definition 7.5.2: Selective Filter

Let F be a set of features and FC_F be the corresponding set of feature configurations. Let \mathcal{MG}_F and \mathcal{G} denote the sets of all multi-variant graphs and single-variant graphs, respectively. A selective filter is a function $s\text{-filter}_F : \mathcal{MG}_F \times FC_F \rightarrow \mathcal{G}$ defined as follows:

For a given multi-variant graph $MG_F = (G, t\text{-map}_F)$ and a feature configuration fc_F , the function $s\text{-filter}_F(MG_F, fc_F) = G'$, with $G' \subseteq G$, adheres to the following properties:

$$N' = \{n \in N \mid (map_F(n) = (a_s, \epsilon) \wedge v_F(a_s, fc_F) = true) \vee (map_F(n) = (a_s, a_m) \wedge v_F(a_m, fc_F) = true)\} \quad (7.9)$$

$$E' = \{e \in E \mid v_F(map_F(e), fc_F) = true\} \quad (7.10)$$

Mapping two annotations onto the same element can be realized when using external mappings, which distinguish these two kinds of annotations. The mechanisms to realize internal mappings

vary (c.f., Sec. 7.5.3, III), affecting the possibility to map annotations onto text fragments. If the internal mapping mechanism allows for mapping a specific annotation, for instance as preprocessor directive, onto a realization fragment, it may offer a possibility to add a second annotation (e.g., in form of a comment) to the respective element.

Equality of Annotations In the following, we will have to compare annotations which are Boolean expressions over features in the feature model and which can be represented as Strings. The straightforward way of comparing two Strings may recognize two annotations as *syntactically different* even though a programmer would recognize them as *semantically equal* [GNS22]. For instance, the annotations $\text{Person} \wedge \text{Family}$ and $\text{Family} \wedge \text{Person}$ are syntactically different but satisfy the same set of feature configurations. Thus, they are semantically equal. For that reason, for comparing two annotations, being Boolean expressions, they have to be in a normalized state, e.g., in CNF, to be *syntactically* comparable and to be able to uniquely determine their syntactic equality.

If the feature model is consulted as context information, even more annotations, which do not assume the same CNF, may have the same effect for keeping an element in a product upon derivation. For instance, if P and F are mandatory features without further constraints, the annotations $a_1 = P$ and $a_2 = F$ will include the elements onto which they are mapped in the same set of filtered variants. Consequently, the two annotations are *semantically equal*, which means they satisfy the same set of feature configurations.

For determining semantic equality, the context needs to be considered. Three simplification mechanisms which regard the feature model as content have been proposed by von Rhein et al. [Rhe+15]. These strategies employ mostly heuristics and do not discuss the comparison of two distinct annotations but check only if an annotations and its simplified form have the same effect. To use them for comparing two distinct annotations, both annotations must be simplified with a deterministic strategy and compared thereafter. However, this requires, for instance, that the strategies render the two annotations, a_1 and a_2 of above example equal.

Protected Annotations To preserve an annotation from overwriting in an incremental propagation, the annotation can be flagged similar to employing protected regions in M2T transformations. A *protected* annotation may not be changed by the automated propagation algorithm. This functionality can enrich each type of mapping mechanism.

In an incremental transformation, protecting an annotation will be indispensable if only one (type of) annotation is stored per model element. In contrast, when storing manually modified annotations, separately protecting the annotation is not necessary because the computation does not change the manually assigned one.

To check, whether an annotation is protected, we employ a function *protected* defined as follows:

Definition 7.5.3: Protected Evaluation Function

Let G be a graph and $EL = N \dot{\cup} E$ its elements. The protected evaluation function $protected : EL \times A_F \rightarrow \{true, false\}$ states whether the given annotation of the given element is protected, i.e., it cannot be changed.

Incremental Model Transformation Furthermore, we assume that the incremental transformation which converts the source model into a target model has already been performed. As a result, a trace model H_{TR} is available which records all pairs of corresponding source and target elements. The incremental annotations propagation iterates this trace model as in batch mode, and computes annotations for the target elements of each trace element as described next.

II Incremental Annotation Propagation

According to the previous descriptions, different types of mappings can occur. This section illuminates the two situations of only one and two annotations stored per mapping and proposes algorithms to prescribe how to assign target annotations incrementally.

In the first situation a single annotation is mapped onto a model element. This annotation can be *protected* from being overwritten. In the second situation two different annotations are mapped onto each element as defined in Def. 7.5.1. The mapping encompasses the stored annotation, which was automatically assigned in a previous execution, and a manually modified one. While an automatically assigned annotation must be present for each target element at each time after an initial propagation, the manually modified annotation can also be empty and is expected to be empty for the majority of model elements.

Single Annotation In the sequel, we propose an incremental propagation algorithm for the situations in which the MDPLE tool stores a single annotation per model element. In this case, the MDPLE tool must offer a mechanism to flag target annotations in order to prevent them from being overwritten if they are changed manually.

Accordingly, the propagation algorithm does not need to compare the computed annotation with the stored annotation but needs to check whether the already stored annotation is protected. Given the presence of a protection mechanism, the incremental annotation propagation algorithm may overwrite any annotation which is not protected.

Algorithm 11 Incremental propagation of annotations one annotation mapped onto elements.

```

1: procedure PROPAGATE( $H, map_{F_S}, map_{p_{F_T}}, map_{F_T}$ )
2:   in  $H = S \leftarrow TR \rightarrow T$             $\triangleright$  STT graph, derived from  $S \leftarrow TR' \rightarrow T'$  in an incremental
      transformation
3:   in  $map_{F_S} : EL_S \rightarrow A_F$             $\triangleright$  Annotation function for the source graph  $S$ 
4:   in  $map_{p_{F_T}} : EL_T \rightarrow A_F$         $\triangleright$  Partial annotation function for the target graph  $T$ 
5:   out  $map_{F_T} : EL_T \rightarrow A_F$         $\triangleright$  Annotation function for the target graph  $T$ 
6:   var  $TRG$                                   $\triangleright$  Set of target elements
7:   var  $SRC$                                   $\triangleright$  Set of source elements
8:   var  $a_c$                                   $\triangleright$  Computed annotation
9:   for all  $n_{tr} \in N_{TR}$  do                  $\triangleright$  Iterate trace nodes in topological order
10:     $SRC := \{s \in EL_S \mid n_{tr} \xrightarrow{src} s\}$ 
11:     $TRG := \{t \in EL_T \mid n_{tr} \xrightarrow{trg} t\}$ 
12:     $CTX := \{t \in EL_T \mid n_{tr} \xrightarrow{ctx} t\}$ 
13:     $a_c := \bigwedge_{s \in SRC} map_{F_S}(s) \wedge \bigwedge_{c \in CTX} map_{F_T}(c)$   $\triangleright$  Compute annotation from source
      elements
14:    for all  $t \in TRG$  do                      $\triangleright$  Iterate target elements
15:      if  $\neg protected(t, map_{p_{F_T}}(t))$  then
16:         $map_{F_T}(t) := a_c$                     $\triangleright$  Overwrite annotation
17:      else
18:         $map_{F_T}(t) := map_{p_{F_T}}$             $\triangleright$  Retain protected annotation
19:      end if
20:    end for
21:  end for
22: end procedure

```

Alg. 11 presents the steps which are performed in an incremental propagation where target annotations may be protected. The algorithm receives the STT-graph (i.e., the source, trace and target model) after having performed the incremental transformation, and the source and target mapping functions. In contrast to the batch transformation, the target mapping function is not only output but also input and stores annotations for each model element which was created in a previous transformation execution. Since for new elements no annotation may be present yet, the given target mapping function is partial whereas the output target mapping function is total.

Similar to the batch propagation algorithm (c.f., Alg. 1), Line 9 of Alg. 11 iterates the trace model in topological order, which ensures that an annotation assigned to a context element exists already. The algorithm retrieves the sets of recorded sets of source, context and target elements from each trace node n_{tr} . The source and context elements are used to compute the annotation as in the

batch approach. If the set of context elements remains empty (e.g., due to a generation-complete trace), the neutral element of a conjunction (i.e., `true`) will be placed instead.

Then, the computed annotation is assigned to each target element, the annotation of which is not protected. Since no further comparison with the previous annotation is performed, the new annotation replaces the old one without further constraints. If no annotation was present so far because the target element was created by the incremental transformation that occurred right before the propagation, the computed annotation will be the first one mapped onto the element. To this end, the annotation of each element is updated or preserved.

It must be noted, that at first glance, it would seem beneficial to iterate only trace model elements which have been added in comparison to the previous trace version. However, this would not suffice because annotations of existing source elements may also have been changed. Therefore, the entire trace model is iterated.

Two Annotations In contrast to overwriting each annotation which is not preserved, the following paragraphs describe how to propagate an annotation when two annotations are stored per element. If an MDPLE tool realizes such behavior, the $t - map$ function replaces the mapping function of the multi-variant target graph. In a backward transformation the source mapping function would also have to be replaced but in unidirectional scenarios it can be kept, assuming that there is no difference between a manual and a computed source annotation.

Alg. 12 describes the steps that are performed during the incremental propagation in the presence of up to two annotations per element. Similar to the algorithm before, the STT graph as well as the source and partial mapping annotation function are input to the algorithm. The complete target mapping annotation function forms the output of the algorithm. In contrast to Alg. 11, the partial and complete target mapping functions store the manually mappings, too.

Again, the algorithm iterates the trace model in topological order and retrieves the recorded sets of source, context and target elements of each trace node. Based on this information and the source and target mapping annotation function, the algorithm computes the annotation as in the batch version defined in Alg. 1. When iterating the target elements thereafter, the algorithm checks in the first place whether a *manual annotation* is mapped onto the target element.

If no manual annotation exists, the algorithm will deduce that no specific user added information needs to be preserved. Thus, the algorithm maps the computed annotation onto the target element in Line 20. If a manual annotation is present, the algorithm checks whether the stored and the computed annotations are the same. If they are equal, no further action is necessary because the annotation does not change the effect of the already existing annotation.

If the stored and the computed annotations diverge, the algorithm checks whether the computed annotations equals the manual one. For instance, this may be the case if the previously assigned annotation a_s was too broad before and the developer refined the target annotation before changing it on the corresponding source elements, too. In this case, the stored annotation is overwritten and the manual annotation remains as it is. Particularly, the manual annotation is not deleted because in another subsequent propagation the information might be necessary again.

Finally, if the three annotations, a_c , a_s and a_m diverge, a conflict exists. However, as the manual annotation outweighs the computed annotations in the selective-filter, a straightforward solution of resolving this conflict overwrites the computed annotation. Nonetheless, it might be beneficial to inform the product line developer that particularly the two computed annotations deviate, which might require an update of the fixed manual annotation, too.

All in all, in the end, each trace element and, thus, each information from the source model is propagated to the target model and a total mapping annotation function accompanies the target model.

7.5.4 Discussion

To sum it up, the two propagation algorithms as well as the ways to represent mappings aim to support the incremental maintenance of annotations in a multi-variant target model. The primary goal of this thesis is to protect target annotations, which were manually modified by the product line developer, in a consecutive execution of the multi-variant transformation.

Algorithm 12 Incremental propagation of annotations with *two* annotations mapped onto target elements.

```

1: procedure PROPAGATE( $H, \text{map}_{F_S}, t - \text{map}_{p_{F_T}}, t - \text{map}_{F_T}$ )
2:   in  $H = S \leftarrow TR \rightarrow T$   $\triangleright$  target STT graph, derived from  $S \leftarrow TR' \rightarrow T'$  in an incremental
   transformation
3:   in  $\text{map}_{F_S} : EL_S \rightarrow A_F$   $\triangleright$  Complete annotation mapping function of source graph  $S$ 
4:   in  $t - \text{map}_{p_{F_T}} : EL_T \rightarrow A_F$   $\triangleright$  Partial annotation function of the target graph  $T$ 
5:   out  $t - \text{map}_{F_T} : EL_T \rightarrow A_F$   $\triangleright$  Annotation function for the target graph  $T$ 
6:
7:   var  $TRG$   $\triangleright$  Target elements
8:   var  $SRC$   $\triangleright$  Source elements from which target annotation is computed
9:   var  $a_c$   $\triangleright$  Computed annotation
10:  var  $a_m$   $\triangleright$  Manual annotation
11:  for all  $n_{tr} \in N_{TR}$  do  $\triangleright$  Iterate trace nodes in topological order
12:     $SRC := \{s \in EL_S \mid n_{tr} \xrightarrow{src} s\}$ 
13:     $CTX := \{c \in EL_T \mid n_{tr} \xrightarrow{ctx} c\}$ 
14:     $TRG := \{t \in EL_T \mid n_{tr} \xrightarrow{trg} t\}$ 
15:     $a_c := \bigwedge_{s \in SRC} \text{map}_{F_S}(s) \wedge \bigwedge_{c \in CTX} t - \text{map}_{F_T}(c)$ 
16:
17:    for all  $t \in TRG$  do
18:       $(a_s, a_m) := \text{map}_{p_{F_T}}(t)$ 
19:      if  $a_m = \epsilon$  then  $\triangleright$  If the annotation was not manually
20:         $\text{map}_{F_T} := (a_c, a_m)$   $\triangleright$  The old annotation can be replaced
21:      else
22:        if  $a_s \neq a_c$  then
23:          if  $a_c = a_m$  then
24:             $t - \text{map}_{F_T}(t) := (a_c, a_m)$   $\triangleright$  Replace stored annotation
25:          else
26:             $\text{RESOLVECONFLICT}(t, a_s, a_c, a_m, t - \text{map}_{F_T})$ 
27:          end if
28:        end if
29:      end if
30:    end for
31:  end for
32: end procedure

```

For this task we have discussed the possibility to protect annotations and to store two different annotations. Furthermore, the algorithms for propagating annotations based on a trace have been refined accordingly. The following paragraphs discuss the assumptions that we posed as well as the benefits of this approach and spotlight related work.

Assumptions The presented propagation algorithms make several assumptions on the evolution of product lines. We refrain from regarding the evolution of the feature model and the target domain model as well as from deleting or updating source elements.

Although we do not explicitly discuss the evolution of feature models, this does not hamper their development in an evolution scenario. We assume that changing the feature model triggers a validation of all existing models in the product line by the MDPLE tool. If annotations do not conform with the modified feature model, this problem must be solved when changing the feature model. At the most, the multi-variant transformation could be enriched by a pre- and post-processing step to evaluate the correctness of annotation with respect to the feature model. In addition, we do not focus on domain model changes. Accordingly, situations in which the target domain model changes differently than the source model are not considered. Particularly, it is not possible that an element of the multi-variant target domain model assumes a manual value which cannot be derived by executing the reused model transformation. This assumption helps

to ensure commutativity of annotations also in the incremental scenario. If only source elements are added and the transformation rules are local, functional and monotonic, corresponding target elements should be integrated in the target model seamlessly without threatening commutativity (if corresponding trace elements record the target elements).

Benefits Using fixed mappings pays off when incomplete traces are present and a completion strategy computes the missing annotations as proposed in Sec. 7.3. For instance, the completion strategies always map the root feature onto the compilation units of a Java model, which are created for each UML class in a UML class to Java transformation, although the compilation units might represent optional elements. Thus, the developer refine the broad annotations and desires to preserve the manually invested work in consecutive executions. Since modifying those annotations each time a batch propagation takes place is a laborious and redundant task, this manual effort saved each time the multi-variant transformation is executed incrementally again after the change.

As another point, the algorithms for incrementally propagating annotations are described in a generic way, independent of the MVMT approach. The concrete realization of how to determine the annotation and how to verify whether an annotation may be changed may diverge. Similarly, handling non-matching annotations by the means of a three-way comparison requires a tool-specific way to represent these two-fold annotations as well to compare annotations. Since we do not assume a tool-specific or transformation language-specific behavior (e.g., for recording modifications or for storing mappings concretely), the concepts are generally applicable.

Pitfalls Particularly, when two types of annotations, a manual and an automatically computed annotation, are mapped onto a model element, the question arises which annotation weighs more. The definition of the selective filter (Def. 7.5.2) favors the manual annotation and determines the presence of the model element based on this annotation. However, in the incremental forward propagation it may not be clear how to handle the situation when the stored and the newly computed annotations deviate and are not the same as the manual annotation. Our solution prefers the manual annotation and assumes that the stored annotation can be overwritten when no manual annotation is present (assuming that if the stored annotation was wrong, it would have been replaced with a manual one). Despite the automation of this behavior, the annotation may deviate from the developer’s intention. Therefore, we propose to report real conflicting annotations if the stored, computed and manual annotations diverge.

Finally, in a unidirectional propagation the distinction between automated and manual annotation makes only sense for the target of a source model. It is assumed that the source model is annotated completely and that the source annotations are correct regardless whether the annotation were provided manually or the source model resulted from a prior transformation.

Related Work Regarding related multi-variant transformations, protecting and incrementally maintaining annotations has not been considered apart from this thesis, to the best of our knowledge. The category-based framework for evolving product line transformations [Tae+17], regards the evolution of the feature model as one artifact besides the evolution of the source domain model. However, it does not consider the annotations explicitly. To the best of our knowledge we assume that the target annotations will be overwritten, if they already exist. A new approach, developed by the authors of this thesis, considers the maintenance of annotations in synchronization scenarios (i.e., in bidirectional transformations) which goes beyond the scope of this thesis which focuses on unidirectional (incremental) transformations [GNS22].

7.6 Summary

In summary, this chapter proposes strategies to propagate annotations in situations in which the traces are not complete and in which the granularity level of trace information deviates from the level of the mappings in a trace-based propagation (i.e., violating Prop. 5.3.8 and Prop. 5.3.7).

The first sections, Sec. 7.1 and Sec. 7.2, illuminate the situations where each target element representing an object can be annotated but information about context elements or mappings of structural features, respectively, is missing. Firstly, the descriptions show that a propagation based on a *generation-complete* trace can create a completely annotated multi-variant target model. However, the algorithm may map an annotation onto the target elements which is too broad such that these elements may be present in single-variant target models which cannot be created with the reused single-variant transformation due to the missing context element.

Secondly, for retrieving mappings of structural features, we offer a solution which analyzes the bytecode model of the transformation specification. As a consequence, the analysis requires to access the bytecode model and works in an independent way of the concrete transformation (i.e., the source and target metamodels) and, conceptually, of the transformation language.

For situations in which traces are *incomplete*, Sec. 7.3 proposes and discusses three completion strategies to determine the missing annotations. These heuristics exploit the tree-structure of the target models to compute missing annotations and assume that parts of the model are already correctly annotated. A theoretical analysis shows that the accuracy depends on the presence of annotations at parent or children graph nodes and that the computational complexity is in the *worst* case cubic.

If *no trace* is present, the information of corresponding source and target model elements needs to be restored in a different way. As black-box approach, which works independent of a transformation language, propagation DSLs, such as the ModelSync DSL, can be *manually* specify corresponding elements as described in Sec. 7.4.2. A specification in the DSL basically rewrites a declarative transformation specification by specifying corresponding source and target objects and employs this information to propagate the annotations from source to the corresponding target elements matching the specified patterns *automatically*.

In contrast, comparing the source and target model and to determine matching elements automatically represents a solution situated at a higher degree of automation and genericity. However, without assumptions on the metamodels to which the models conform, the accuracy of matched model elements by the model comparison approach presented in Sec. 7.4.3 remains low. Due to low levels of accuracy, granularity and automation of model matching approaches and the propagation DSL, we refrain from integrating proofs of concepts in our multi-variant model transformation framework presented in Chp. 9.

Finally, annotations computed by the heuristics and based on generation-complete traces may be too broad or too specific for pertaining model elements in filtered variant. Therefore, Sec. 7.5 presents how to protect manually modified target annotations in an incremental annotation propagation.

Chapter 8 Model-To-Text Annotation Propagation

*Simplicity does not precede complexity,
it follows it.*

Alan Perlis, Epigrams on Programming, 1982

~

While the previous contributions propagate the annotations of one *model* to another based on *traces* or comparable information, this chapter offers a solution to integrate the annotations into *text* generated by a model-to-text transformation. Instead of traces, the chapter employs concepts of *aspect-oriented programming* in order to propagate annotations without a need to change the execution semantics of the transformation engine.

Firstly, Sec. 8.1 illustrates the problem of deriving customized source code without an annotated multi-variant source code platform. As one solution to overcome this problem, Sec. 8.2 explains the essential background information on template-based model-to-text transformations and of aspect-oriented programming. Based on this description and further observations, Sec. 8.3 formalizes M2T transformations by employing the Graph formalism, too, and defines the computational model and corresponding algorithm for aspect-oriented annotation propagation. Sec. 8.4 discusses the design decisions of the computational model and of annotating source code fragments while additionally reflecting on related work on the topic. A summary provided in Sec. 8.5 concludes the chapter.

The chapter is based on the publications for employing aspect-oriented annotation propagation [GW18a; GW18b].

8.1 Problem Statement

Serving as introduction to this chapter, this section motivates and stresses the importance of feature traceability, in general, and of annotations in source code, in particular. Firstly, a motivating example demonstrates the effect of manually integrating source code into method bodies. Without a multi-variant source code platform the product line developer has to repeat the integration in each derive variant. In this way, the example emphasizes the resulting deficiency in productivity and draws consequences to reduce the manual efforts.

8.1.1 Motivation

To motivate the necessity of annotated source code, (instead of generating single-variant models from which the source code is derived), this section employs the family database example from the previous chapters (Fig. 4.1.3). In contrast to the previous examples, this example focuses on an excerpt of the model representing the derived attribute `name` of the class `Person`. Accordingly, in the source code an implementation of a method `getName()` is necessary.

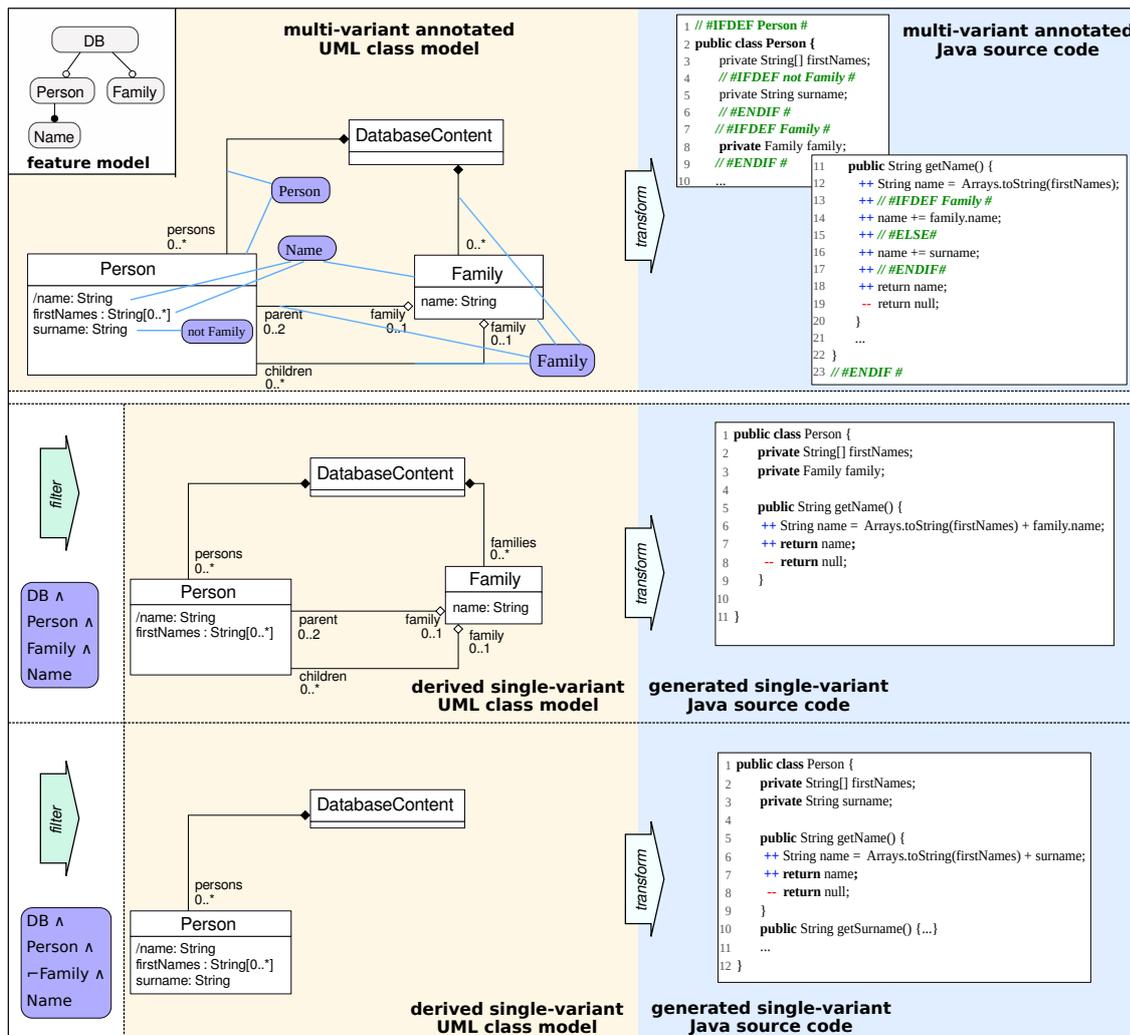


Figure 8.1.1: Integration of method body in multi-variant vs. single-variant source code.

Scenario Fig. 8.1.1 illustrates the scenario on the top left. The UML class model comprises the classes `Person` and `Family`, which both store names. The upper left corner depicts the corresponding feature model excerpt. If a feature configuration selects the feature `Family`, the property `surname` will be missing in the person class of the corresponding derived model, as depicted on the left side of the *second* row in the figure. The surname will be present and the family class will be missing, if the feature `Person` is selected but the feature `Family` not, as depicted on the left side at the bottom of the figure. Model elements without mapped annotations are integrated in each product.

Template-Based UML Class Model to Java Transformation In this scenario the employed transformation represents a simplified real world transformation: the Java source code generation for UML class models. Although Fig. 8.1.1 focuses on the transformation of a class and its attributes, we summarize the basic behavior of the entire transformation in short:

It creates a class declaration for a UML class only (and no interface). Accordingly, first, the transformation creates a class file for each class declaration which contains a *package* and *import* statements before opening the class declaration. For each property of a UML class, field declarations are inserted followed by integrating the method declarations. While field declarations contain only a type, a visibility and a potential modifier, a method may also include a parameter declaration which is inserted with the separator ‘,’ inside the enclosing rounded brackets. The return type is inserted before the method name. For derived properties of the UML class, such as the name of the class `Person` in the example, no field declaration but an access method `getName()` is created which declares the type of the property as return type. Furthermore, access methods may be created for each ‘normal’ field declaration which are left from the example. Finally, the transformation integrates a default return statement in the body of each method declaration to generate a valid source code file without compilation errors.

Process Without Automation Support To retrieve source code for a product, without a multi-variant transformation the developer needs to derive a product from the multi-variant model and generate the corresponding source code by transforming the models into Java. The lower part of Fig. 8.1.1 illustrates this procedure by showing the generated source code on the right side. Since the property `name` of the class `person` is a derived attribute, it requires an implementation of the method body. Per default, the code generation creates the method header and a default return type (Lines 5 and 8) only. Accordingly, a developer provides the method body in the way sketched in the figure. The developer removes the default return type `null`, indicated with red `--` in both products. However, due to the different information present in the derived models, the implementation of the method bodies varies for the two products. Blue `++` indicate the manually added lines of source code which compute the full name, composed of the first names followed by the family name depending on the available information.

Redundant Error-Prone Effort Continuing the product derivation (i.e., filtering the multi-variant UML class model and generating the corresponding source code) requires to implement the method body repeatedly. Typically, a developer may copy-and-paste and adjust the method bodies, if needed. Despite missing automation and, thus, an increased *time-consuming* and *expensive* manual effort, this procedure is also *error-prone*: The manual implementation might be modified inadvertently or on purpose, resulting in diverging behavior for the same kind of product.

Solution Idea Generating multi-variant source code from the multi-variant source model instead solves the problem of redundant implementations and reduced automation. Even though multiple approaches to generate source code for class models exist, such as the default Ecore code generation [Ste+09] or the UML modeling tool Valkyrie [Buc12], these generators are unaware of variability and, thus, not able to integrate annotations in the generated text. Accordingly, such generator can produce multi-variant source code but for deriving customized products annotations are missing. For this task, it is necessary to map annotations onto corresponding source code fragments.

The upper right side of Fig. 8.1.1 sketches the result of a multi-variant M2T transformation which includes annotations. The multi-variant transformation maps the annotation of a source object onto the corresponding created source code fragments as preprocessor directive. Since the example demonstrates the creation of Java source code, the preprocessor directive (which are not supported out-of-the box) is integrated as a comment such that the source code can still be compiled. Furthermore, the method `getName()` misses an implementation which is added manually. In contrast to deriving a UML class model variant, generating its source code and implementing the method bodies, the method body for the multi-variant source code is implemented only once. Consequently, conditionally compiling the multi-variant source code to derive a single variant integrates the method body automatically and does not require interaction with the developer.

8.1.2 Consequences

The example illustrates the importance of generating *annotated* multi-variant source code and deriving products from that source code platform.

At minimum, annotated multi-variant source code offers the following benefits, which affect development cost and correctness positively:

- **Automatic Single-Variant Code Generation** A *customized* product can be derived from the multi-variant source code automatically without deriving a single-variant model first.
- **Reduced Copy-And-Paste of Manual Implementations** Changes to the source code, such implementations of method bodies, can be provided in the multi-variant source code platform. Consequently, derived products from the source code platform include manually added source code fragments concerning only the selected features. This removes the necessity to copy and paste manual modifications in every newly derived product.
- **One Dedicated Place for Maintenance** A multi-variant source code platform (and remaining links to derived products) may offer the possibility to integrate, for instance, bug fixes from a product, into the platform and distribute them to all other already derived products. Linking products with the platform as well as their reactive integration, however, is out of the scope of this thesis.

Despite this artificial example for motivating the contribution, a similar study [Ji+15] confirms the need for feature traceability across design and source code artifacts. Using preprocessor directives is an easy to embed and light-weighted way of annotating source code in the first place [Ape+13]. However, it must be noted that, despite the light-weight integration of annotations into source code, without tool support the maintenance of this code may become cost-intensive due to potentially scattered features and diverging behavior which requires an increased amount of cognitive complexity [Mar+21; AK09; SW19].

8.2 Aspect-Oriented Multi-Variant Source Code Generation

While a *trace-based* multi-variant model transformation establishes a tool-independent, post-processing approach to create annotated multi-variant *models*, it does not enclose concepts for creating annotated multi-variant source code out-of-the-box. M2T transformations rarely maintain persistent trace information but rather rely either on matching source code elements or a manual specification of *tracing* information by using IDs to recognize text fragments corresponding with source model elements. Even if sufficient trace information were available, technically an a posteriori trace-based propagation would require to re-execute the transformation to incorporate annotations internally in form of preprocessor directives or to build an external mapping file which uniquely identifies text fragments with annotations.

```

1 [module ecore2java('http://www.eclipse.org/emf/2002/Ecore')]
2
3 [template public main(p : EPackage)
4   [comment @main/]
5   [for (cl : EClassifier | ec.eClassifiers)]
6     [let c : EClass = cl ]
7     [eClass2Java(c)/]
8   [/let]
9 [/for]
10 [/template]
11
12
13 [template public eClassToJavaSC(ec : EClass) ? (ec.interface = false)]
14   [file (ec.name + 'java', false, 'UTF-8')]
15   public class [ec.name/] {
16     // Constructor
17     [c.name/]() {}
18     [for (attr : EStructuralFeature | ec.eStructuralFeatures)]
19       [eStructFeat2Field(attr)/]
20     [/for]
21   }
22 [/file]
23 [/template]
24
25 [template public StructFeat2Field(attr : EStructuralFeature)
26   ? (attr.derived = false and attr.many = false)]
27   private [getType(attr.eType)/] [attr.name/];
28 [/template]
29 ...

```

Listing 8.2.1: Example of an Aceleo module composed of three text production rules (`template`).

Road Map This section presents the background information on M2T transformations and aspect-oriented behavior the concept of both are used to embed annotations in the target text *while* executing the reused transformation. First, Sec. 8.2.1 examines the two template-based M2T languages Aceleo and Xpand to derive commonalities and basic concepts of such transformations. Thereafter, Sec. 8.2.2 illuminates the concepts and implementations of aspect-oriented programming in more detail.

8.2.1 Template-Based Model-To-Text Transformation

This section introduces the basic concepts for creating text from a given model via *template-based* M2T transformations. Examples taken from Aceleo, the implementation of the MOF2Text standard issued by the OMG, and Xpand serve to illustrate and infer the basic concepts shared by template-based languages. The first part of this section explains language constructs for specifying the transformation while the second part presents the execution behavior which are both summarized thereafter.

I Text Production Specification

To demonstrate the specifics of template-based text productions, this following paragraphs present the language concepts and execution semantics of the transformation engines Aceleo and Xpand.

Aceleo [Obj08] is a template-based M2T language, which implements the MOF2Text standard. To demonstrate the language constructs, Listing 8.2.1 presents an excerpt of an Aceleo transformation which generates a Java class from a given Ecore class in a simplified form. Accordingly, the first line declares the module's name, `ecore2java`, and the Ecore metamodel as the metamodel, the instances of which are transformed with this specification.

At the coarsest level, Acceleo allows to compose transformations of *modules*. Modules can import functionality from other modules (`import-directive`) or extend other modules, i.e., Acceleo supports inheritance between modules. A module comprises *template* definitions, which are rules declaring for one type of the source model how corresponding text should be constructed from the object's properties, and *queries*, which – similar to methods – may implement behavior used at various places or may build strings by employing OCL expressions. Furthermore, static Java methods may enrich the production rules for querying complex computations which are hard to express in OCL.

Inside modules, templates prescribe which text should be produced for each object *o* matching the type declared in the template. Such *text producing rule* ϑ comprises *static* text and dynamic productions. Lines 3-10, 13-23 and Lines 26-28 of the listing represent templates.

The creation of a *file* can be initiated in a template and, thus, serves as physical storage where the text production is written to. Line 14 of Listing 8.2.1 initiates the creation of in the rule which generates a class declaration by providing the name and path as well as the encoding format (e.g., in the example: UTF-8) for the generated text.

Furthermore, the static text can be intermixed with *dynamic* text productions which compute the printed value: On the one hand, a dynamic directive may be resolved directly in the rule ϑ , similar to a variable which is dynamically replaced with the concrete value of the matching object's *o* property. On the other hand, the rule may invoke another rule ϑ' if the text should be retrieved from another object *o'* directly referenced by the object *o* which matches the invoking template ϑ . Retrieving the name of the class to create the class declaration in Line 15 with the directive `[ec.name/]` is an example of a directly stated dynamic directive in Acceleo. Invoking another template, to generate text for all attributes in the given `EClass` is demonstrated inside the for-loop in Line 7. The *let-expression* stated in Line 6 performs a type cast and provides an alias such that only `eClassifiers`, which are of type `EClass`, are input to the corresponding invoked template.

In addition, *guards* may restrict the kind of object that match a template by the values of their structural features. A guard, initiated by a question mark in the concrete syntax of Acceleo, can be stated after the declaration of the type that matches a template and constrains its application, similar to a *positive application condition* in M2M transformations. In the example, the `eClassToJava` template will be only executed if the given `EClass` is no `interface`. Similarly, *if-conditions* inside the template can constrain the text production based on certain values and branch the control flow. *for-loops* serve to further prescribe the order of the control flow when iterating the given objects' structural features. The for-loop in Lines 18-20 iterates all structural features of the class to invoke another template which transforms `EAttributes` and `EReferences`.

Xpand In contrast to Acceleo, the language Xpand [Eff+04] does not realize the MOF2Text standard. Nevertheless, it encompasses similar concepts, such as for-loops and if-statements as well as the invocation of other text producing rules. The language resides in the openArchitectureWare [EV06] framework and is integrated in the language Xtend to define complex String computations. To illustrate the syntax specifics in the following explanations, we employ the same transformation as for explaining the Acceleo constructs. Accordingly, the transformation specification in Listing 8.2.2 realizes the same functionality as implemented in Listing 8.2.1.

Xpand does not realize a module mechanism, such as Acceleo, nor offers queries. Instead, templates can access *extensions*, which are specified in Xtend and may express and compute more complex text productions such as the iteration of a collection of elements for determining the printed String. Furthermore, *import*-statements declare a namespace which obviates the need to mention the fully qualified name each time a type is stated. Even though they are not called *templates*, an Xpand transformation specification consists of *define*-blocks which represent a text producing rule similar as an Acceleo template. An MWE(2) workflow collects necessary additional information for executing an Xpand template, such as the metamodel, the instances of which should be transformed.

Listing 8.2.2 imports the Ecore namespace in Line 1 while Line 2 imports an extension file, called `path` and located in the directory `extensions`, which summarizes additional methods specified in Xtend. The methods in the extension file compute the path names of files or packages.

```

1 «IMPORT ecore»
2 «EXTENSION extensions::path»
3
4 «DEFINE main FOR EPackage»
5 «EXPAND classifier FOREACH eClassifiers»
6 «ENDDEFINE»
7
8 «DEFINE classifier FOR EClassifier»
9   «IF metaType.name == "ecore:EClass"»
10  «EXPAND EClass FOR (EClass) this»
11  «ENDIF»
12  «IF metaType.name == "ecore:EEnum"» ... «ENDIF»
13 «ENDDEFINE»
14
15 «DEFINE EClass FOR EClass»
16   «REM» create corresponding class «ENDREM»
17   «IF interface == false»
18     «FILE getPackagePath() +"/impl/" + name + "Impl.java"-
19
20     package «getPackageName()».impl;
21     «EXPAND templates::implementations::class FOR this»
22     «ENDFILE»
23   «ENDIF»
24 «ENDDEFINE»
25
26 «REM» template: 'templates::implementations' «ENDREM»
27 «DEFINE class FOR EClass»
28 public class «name»Impl {
29   «EXPAND EStructuralFeature FOREACH eStructuralFeatures»
30 }
31 «ENDDEFINE»
32
33
34 «DEFINE EStructuralFeature FOR EStructuralFeature»
35 «IF derived == false»
36 protected «getType()» «name»;
37 «ENDIF»
38 «ENDDEFINE»

```

Listing 8.2.2: Example of Xpand text production rules for EClasses.

A define-block, denoted as «DEFINE», is a text-producing rule declaring one type of object, and combines static and dynamic text generation directives. A *file*-directive («FILE») inside the define-blocks initiates the creation of a file in a batch transformation or accesses a matching one in incremental transformations. Any static text inside the directive is written verbatim and any dynamic text is written in its resolved form to that file in the order declared by the block. In contrast to Aceleo, it is not possible to further restrict the declared type of the define-block directly with a guard. If different text needs to be produced depending on the values of structural features of the matched object, *if*-directives inside the define-block need to initiate these branches.

Besides *if*-directives, the language offers several mechanisms to iterate a collection of structural features. A *for*-loop can be declared inside a define-block and produces text at the declared position. Alternatively, similarly to invoking other templates on referenced objects, it is possible to invoke another define-block by writing a directive EXPAND, such as in Line 29 of Listing 8.2.2. This could be regarded as an indirect *for*-loop which iterates a collection of referenced objects by invoking another text producing rule. Apart from the main definition block, only define-blocks invoked by other rules are executed.

In Listing 8.2.2, the *main* define-block (Lines 4-6) invokes the respective define-block for each type of classifier contained in the matching package. As define-blocks and *expand*-directives do not

```

1  bean = org.eclipse.emf.mwe.utils.StandaloneSetup{
2      registerGeneratedEPackage = "org.eclipse.emf.ecore.EcorePackage"
3  }
4  component = org.eclipse.xpand2.Generator{
5      id = "generator"
6      metaModel = org.eclipse.xtext.typesystem.emf.EmfRegistryMetaModel{ }
7      expand = "templates::Main::main FOR model"
8      fileEncoding = "UTF-8"
9      outlet = {
10         path = '${targetPath}'
11         postprocessor = org.eclipse.xpand2.output.JavaBeautifier{}
12     }
13     advice = advices
14 }

```

Listing 8.2.3: Excerpt of MWE2 workflow for generating text from an Ecore model.

allow for guards that constrain the objects, the if-directive in Line 9 determines which define-block should be executed for the given type of classifier when being invoked by the main define-block. In this case, the main block invokes the define-block for producing the text of `EClasses` (Line 15). For each matching class a file is created first in Line 18. Anything that is declared inside that directive which ends in Line 22 (i.e., static declarations and invoked dynamic declarations) is included in the file.

After stating and computing the package declaration in the file, another define-block specific for `EClasses` is invoked which generates the class declaration followed by invoking a define-block transforming all structural features. Even though polymorphism is stated to be supported [Kla07], a similar directive as in Listing 8.2.1 matching the concrete type of a structural feature, i.e., an attribute or a reference, cannot be compiled with current versions of Xpand¹. Finally, “REM”-blocks (e.g., in Line 16) signify comments which are not printed to the produced file.

Further differences between Aceleo and Xpand mainly regard additional functionality to express the transformation [Kla07]. For instance, Aceleo allows to define macros and queries can be expressed inside a transformation module directly with OCL syntax whereas Xpand requires to write an extension in Xtend for complex computations. However, overall the template-based transformation of a model based on Aceleo and in Xpand employs mostly similar concepts.

II Text Production Execution

After having introduced the basic language constructs and concepts of template-based M2T transformations, this section derives the commonalities of *executing* such transformations.

Entry Point Rule Each of the two template-based approaches employs an explicit and *unique entry point* at which the text production initiates. Aceleo marks this “main” rule of a module with the comment [comment @main/]. In Listing 8.2.1, packages serve as entry point. For that reason the template, named `main`, matches `EPackages` and states the declaration as main-rule in Line 4 as first statement in the rule. Similar to the Aceleo listing, the Xpand listing (Listing 8.2.2) names one of its define-blocks `main` (c.f., Line 4) even though this is only by convention. As in Xpand, the MWE(2) workflows are decisive for executing the transformation, the workflow declares the entry point of the transformation and in this way where the respective main define-block of this Xpand template is stated. Listing 8.2.3 presents the relevant excerpt of an MWE2 workflow to generate source code. As mentioned above, the metamodel is not stated in the Xpand templates but the registered package is declared in this workflow (in the example in Line 2). The entry point is defined for the `Generator` component as `expand` property in Line 7.

Rule Execution Order For executing the transformation, both, the Aceleo and Xpand engines, iterate the given source model implicitly and search for objects matching the declared type of the entry point. Then, the file and text production is performed in the order as declared in the

¹ Tried with Eclipse Modeling Tools 2020-03 (4.15.0), Xpand Version 2.2.0.v201605260315

text producing rules. A composes static declarations but may also invoke further rules. The text production of the invoked rules is embedded exactly where they are invoked. Thus, the execution is rule-based and works monotonically and hierarchically. Starting at the entry point for matching objects in the input model, subsequent rule applications are extended from that rule and produce text ordered as defined in the rules. To this end, the text production produces a “forest’ of files.

Incremental Executions To be beneficial in scenarios similar as sketched in Sec. 8.1.1, where manual source code is added to the created text files, M2T transformations have to preserve these manual modifications in consecutive executions of the transformation. Acceleo and Xpand offer two concepts for this kind of incremental behavior: *tracing* and *protected* blocks.

Acceleo and the Mof2Text standard encompass a `trace`-directive (`[trace]`). The contents of a trace block are associated with the parameter given to the trace command. To identify the block unambiguously, a unique identifier must be provided. The same holds for protected blocks (`[protect]`) which mark a block as modifiable by the user and retain anything inside in subsequent transformations of the same input model. Protected blocks will only be retained, if the surrounding block exists. If not, the protected area will be removed from the file as well. In contrast to protected blocks, Acceleo incorporates the command for creating trace blocks but, to the best of our knowledge, current engines² do not support this explicit traceability.

To realize incremental behavior, Xpand offers two ways to recognize changes of the input model, on the one hand, and may trace input elements to generated text fragments, on the other hand. While such trace model is created by a callback of the Xpand generator on the fly, for computing the *diff model* for the input model one out of two possibilities can be specified in the workflow: Either an *incremental generation facade* or an *incremental generation callback* can be employed. The facade expects the old and the new input model as well as a trace model and requires to define the *outlet* (i.e., the path and a declaration whether the text can be overwritten). Using the generation callback instead, allows for more control over the steps that are performed in the execution, particularly, to store the input model as a backup copy to be compared in a subsequent execution or to delete files which are obsolete after the execution. [Eff+04].

Furthermore, to retain already created (manually added text), protected areas can enrich the Xpand define-blocks. The *protect*-block requires the definition of characters which allow to integrate a comment in the created text without changing the semantics of the original text production. Furthermore, a unique and stable ID is necessary to identify the text production for the same element. Explicit traceability is not foreseen neither in the template definitions nor in the workflow. However, the language report [Kla07] suggests to implement tracing functionality by using aspects which are supported in Xpand and explained in Sec. 8.2.2.

III Summary

From inspecting the syntactic constructs of Acceleo and Xpand, which realize and implement the same concepts of the MOF2Text standard of the OMG, we can infer the following commonalities of M2T rule specifications.

File Creation Firstly, one M2T specification may be defined in multiple files, i.e., templates (modules in Acceleo) and libraries or helper files summarizing queries which may access a GPL for complex computations. A statement that initiates the creation of a *file* is essential to generate or access a location to embed the text productions. One functionality of such directive is to create the respective directory if necessary.

Single Source Element Secondly, text-producing rules (i.e., templates in Acceleo and define-blocks in Xpand) match one type of object (which can be restricted with application conditions defined over the object’s properties) and combine *statically* generated text *fragments*, which the rule declares literally, with *dynamically* generated ones.

² Tested with Acceleo Core SDK 3.7.11.202102190929 in an Eclipse Modeling Tools Version 2021-12 (4.22.0)

Rule Invocation Firstly, an entry point rule initiates the execution. Secondly, rules may invoke further rules as part of their capability to integrated dynamically determined text. A dynamic production may access a structural feature of the input object (e.g., of type `String`) and directly states the conversion of it into a text fragment. In the more complex case, the dynamic production either invokes a query, which also may receive the object or a structural feature of the object and returns text, or another text-producing rule is performed on elements referenced by the matching source object. Accordingly, one text producing rule ϑ may invoke several other rules θ' . The order of calling the rules is given by the order of invoked text-producing rules and the created text by the order of the matching objects.

8.2.2 Aspect-Oriented Programming

This section recapitulates the basic concepts of *aspect-oriented programming*, in general. The usage of AOP in M2T transformations is sketched shortly in the presentation of model transformation approaches in Sec. 2.2.2 and its usage for realizing compositional variability in Sec. 3.4.2. The following paragraphs first consider the terminology of the central concepts of aspect-oriented programming [Kic+97] and their execution. Thereafter, we shed light on the realization of these concepts in M2T transformations.

Concepts

In programs P, whenever condition C arises, perform action A. [FF00]

This quote by Filman and Fridman describes the essential dimensions of aspect-oriented programming: requiring a *quantification* of the conditions that can be specified to execute an action defined by some *interface* and a *weaving* mechanism that allows to integrate the action into the program. In our context AOP serves as one strategy to realize variability in a organized and reusable way. Sec. 3.4.2 sketches the concepts of aspect-oriented programming as a solution for realizing a compositional variability mechanism. Accordingly, the approach maintains a base version declaring join points in the source code and an advice realizes the extension of the base version feature-wise.

Terminology According to Apel et al. [Ape+13], an *aspect* is the unit that contains an alternative or extending implementation of the original implementation. The representation of this unit may vary depending on the aspect-oriented realization approach. For instance in AspectJ an aspect is comparable with a Java class declaration. Conversely, an *advice* declares the actual implementation of aspects. The *aspect weaver* extends the execution engine and *weaves* the aspects into the original implementation whenever appropriate. During the aspect-oriented execution, a *joinpoint* denotes the *event* when an aspect is executed whereas a *point cut* denotes the actual declaration of the *location and condition* when a joinpoint has to be executed.

Execution Accordingly, in general, an aspect-oriented execution (i.e., not necessarily a model transformation) behaves as follows: The original implementation is executed. Whenever a point cut is reached and a matching advice is available, the original implementation is refused and the implementation of the advice is executed instead. Even though the original implementation is neglected at first, it may be invoked by the advice and executed nevertheless but in the way defined in the advice. Moreover, an advice may not only target one kind of point cut but may be executed for several types [FF00]. For instance, a *wildcard* declaration may realize this functionality.

Aspect-Oriented Model-To-Text Transformation Using an aspect-oriented approach in M2T transformations does not execute a program but generates the text, instead. Thus, in an aspect-oriented M2T transformation, instead of extending source code, the aspects extend text producing rules. Consequently, an aspect is another text producing rule that extends the base templates used in the transformation. Point cuts and the rule execution behavior have to be added to the language which supports the weaving of aspects.

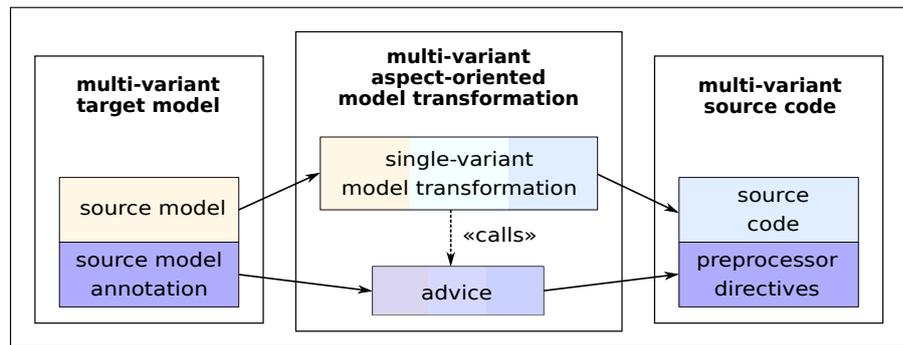


Figure 8.3.1: Schematic overview of aspect-oriented propagation.

Aspect Definition Every aspect-oriented language encompasses the functionality to execute the original implementation: For example, in Xpand a method `targetDef.proceed()`, where `targetDef` represents the element matching the type declared in the point cut, can be invoked for this purpose. Moreover, typically *around*, *before* or *after* statements declare text productions which should be executed in the respective place related with the original production.

8.3 Foundations

Based on the properties of templates-based M2T transformation languages introduced in the previous section, this section defines the foundations how to employ the aspect-oriented paradigm to propagate annotations. Particularly, the section derives a computational model for the aspect-oriented propagation of annotations in M2T transformations.

Road Map The first section, Sec. 8.3.1, offers a descriptive overview of how to propagate annotations and of how to abstract the text generation as execution and target *trees* (in contrast to the STT graphs of M2M transformations). Based on these descriptions, Sec. 8.3.2 derives a computational model of aspect-oriented M2T transformations which is formally noted in Sec. 8.3.3.

8.3.1 Descriptive Overview

This section offers a descriptive overview of how the annotations can be embedded into source code by employing a generic aspect. In contrast to the preceding approaches, an aspect-oriented multi-variant text generation behaves in the way presented in Fig. 8.3.1. While the source model of the propagation does not change (i.e., it still consists of a multi-variant source model and corresponding annotations), the right side reflects a multi-variant text representation. To annotate the source code, *preprocessor directives* are embedded in the multi-variant source code, yielding an annotated source code platform.

Aspect-Oriented Propagation Similar to multi-variant M2M transformations, a *reused single-variant* M2T transformation creates the multi-variant target text representation (without annotations). In concrete, the single-variant model transformation is a *template-based model-to-text* transformation. In contrast to multi-variant M2M transformations, it does not rely on a trace from the single-variant model transformation and is, thus, a black-box approach. Furthermore, the aspect-oriented multi-variant model transformation does not realize a post-processing approach. It employs a weaving mechanism which integrates the annotations during the execution by *calling* an aspect. Thus, it is an *inter-processing* approach to realize a multi-variant model transformation.

Example For demonstrating the basic concepts necessary to realize aspect-oriented behavior at an informal level, we employ the example, sketched in Fig. 8.1.1, of transforming the UML class model for database contents into Java source code. Fig. 8.3.2 demonstrates how the original text production creates the target text and with which source elements the target text fragments

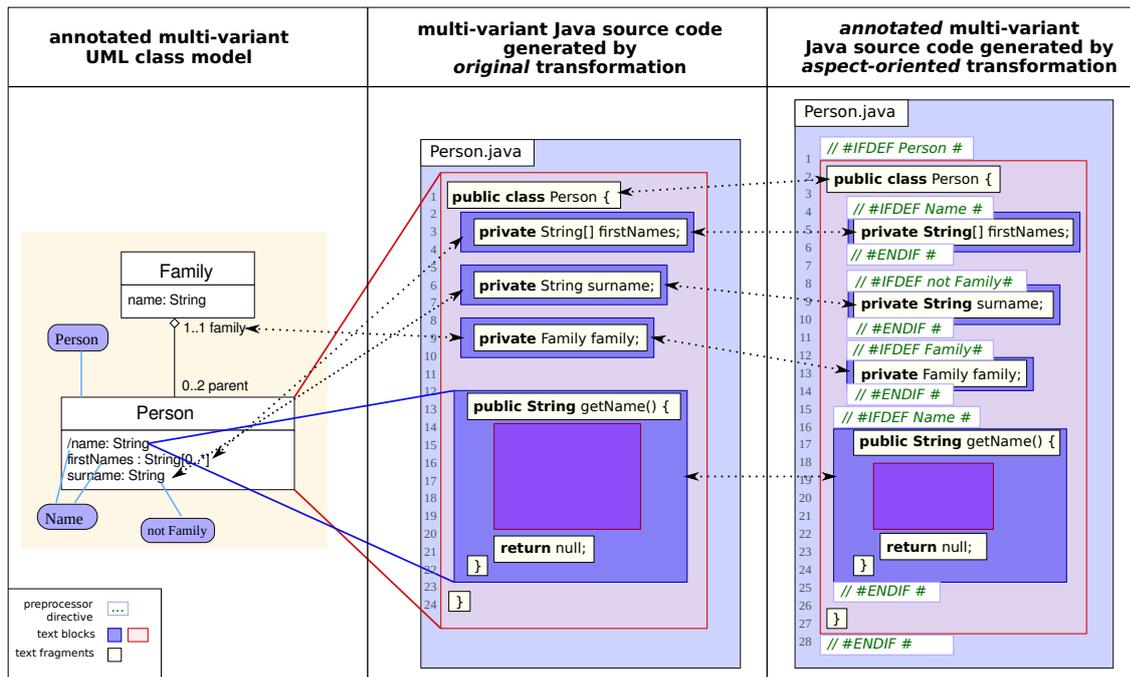


Figure 8.3.2: Preprocessor directives enclosing text fragments in hierarchy of text blocks.

correspond. The file depicted in the middle of the figure represents the result of applying the *original* text production to the class `Person` depicted on the left side. The original transformation creates a file for each class and incorporates the class statement which composes field and method declarations (produced by invoking other rules). The created elements can be abstracted as blocks and fragments: The surrounding text block is created for the Java class `Person` and represented by the underlying red rectangle. It encompasses fragments, such as the class declaration ‘`public class Person`’, and further blocks, for example, the ones holding the fragments created for the UML properties (e.g., ‘`private String surname;`’). Furthermore, for the derived attribute `name` a method header and a default implementation which returns the default value are created.

The right side of the figure visualizes the result of extending the original text production with an aspect that propagates annotations: The aspect embraces each block created by the original text product with a preprocessor directive stating the annotation of the source element. As an example, the class declaration `Person` is surrounded with the directive ‘`// #IFDEF Person #`’. This example lays the ground for deriving informal properties in this section, starting with the representation of the created text, followed by abstracting the transformation specification.

I Target Representation

The result of a template-based M2T transformation (i.e., the generated text) shares common properties, due the nature of the M2T execution:

Text Structure The overall encompassing container of each text production is a *file*, which needs to be created by some rule. *Several files* may result from transforming a source model, for example, a file can be generated for each class in a UML class model. Even though rarely direct creations of a directory exist, typically the files are organized in one overarching directory (with potential sub-directories). Furthermore, as Fig. 8.3.2 shows, the generated text is composed of *blocks*, sketched as rectangles in the figure. Blocks contain either sole text fragments, generated by a rule with static or dynamic directives or other blocks. However, in the final text statically created fragments cannot be distinguished from dynamically created ones. Furthermore, a rule can create multiple blocks and fragments combined arbitrarily but their order is fixed by the rule.

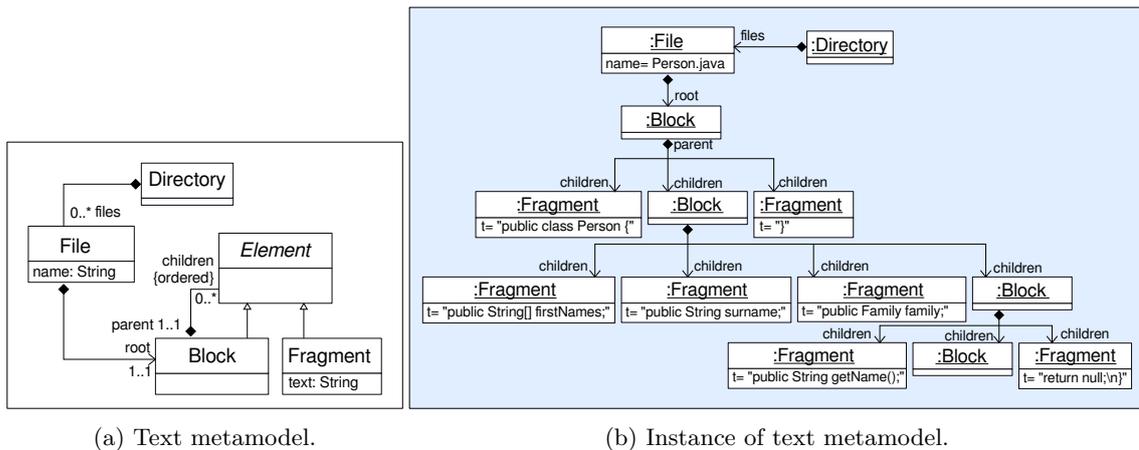


Figure 8.3.3: Representations of text generated by a M2T transformation.

Text Metamodel From these facts, we deduce a metamodel for the generated text. Fig. 8.3.3a presents a generic text metamodel which represents arbitrary text generated by a template-based M2T transformation. The root of the model is a directory which summarizes all files that are created during the transformation. We refrain from representing a hierarchy of directories but assume that all created files can be integrated into one by flattening the hierarchy (which requires unique file names). While the order of the files inside the directory can vary, its contents can not. A file encompasses one root block. Blocks consist of an ordered sequence of children, which are either further blocks or fragments holding the eventually printed text. The rule determines the order of the contained elements. Fragments reside at the bottom of the hierarchy.

Example 8.3.1: Instantiation of Text Metamodel

Fig. 8.3.3b displays an instance of the text metamodel for the Java class *Person*, shown in Fig. 8.3.2.5

The UML class *Person* is transformed into a file called *Person.java* which comprises a main block as *root*. The file is stored in an overall container, which is a directory but not specified in more detail in this example. The root block of the file *Person.java* states the code line `public class Person {` as first fragment followed by another block consisting of the elements contained inside the class and another fragment `}` which closes the class declaration. Furthermore, the block inside the class includes the fragments which are created for each attribute and reference. For the derived attribute *name* a block is generated by the respective rule which encompasses a fragment stating the header of the respective method, `public String getName() {`, and another block which represents the method body, followed by a fragment which holds the return statement and closes the block with the respective bracket. An important characteristic of this model is that for each invoked rule a block, which still has no children, is created by the invoking rule.

Abstraction The description of abstracting generated text as a metamodel allows to infer that the result of a text production can be represented by a *hierarchical graph* which is typed over the classes of the metamodel. A node in the typed graph represents an object and a labeled edge represents a link between two objects which cannot form cycles. While blocks are typically inner nodes, except if they serve as placeholder, fragments are always leaf nodes of the graph. Furthermore, the order of children nodes is relevant and corresponds with the order defined in the text producing rule. The text and name properties of the metamodel can be represented as attributes in *data* nodes linked with the respective *graph* nodes. As a result, the generated target source code can be abstracted by an *attributed typed hierarchical tree*.

II Propagation Process

The propagation process of annotations consists of two parts executed simultaneously: the original *text production* and the *aspect weaving*. This part abstracts the basic characteristics of a template-based M2T transformation as summarized in Sec. 8.2.1, first and describes the aspect-oriented execution thereafter by introducing the abstraction concept of execution tree.

Text-Generating Rules The transformation consists of a set of rule applications Θ which match exactly one source object and create an ordered and hierarchical set of target elements. Furthermore, a rule can invoke another rule the elements of which are integrated in the graph at exactly that position between fragments and blocks where it was invoked. An entry point rule ϑ_r initiates the transformation and matching source objects are iterated implicitly. ϑ_r invokes further sub-ordinary rules yielding the creation of sub-trees in the target text tree.

Execution Tree A hierarchy of rule applications represents the M2T transformations. The transformation execution expands a directed acyclic graph, the nodes of which reference the source object and order the created target nodes as well as the rule invocations according to the rule definition and the respective elements of the source model. This process yields a forest of *execution trees* which create a (sub-)tree in the target graph for each source node matching the entry point rule. Without loss of generality, we assume that this forest of target trees is grouped into one overarching directory which is either created by the first rule application (regardless of the matching source object) or already created but serves as the root of the target graph, nonetheless. When propagating annotations, the overall target root may not get a specific annotation but must always be present, i.e., the annotation `true` must be assigned implicitly or explicitly, such that the element is not removed by any filter.

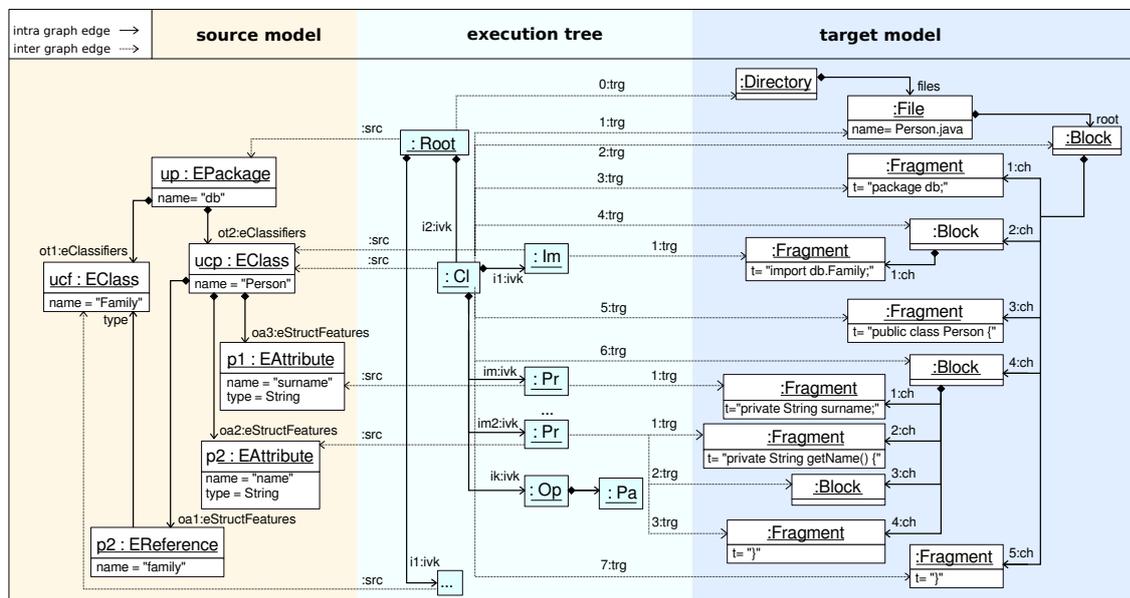


Figure 8.3.4: STT-graph representation of Ecore2Java M2T transformation. The target model conforms to the text metamodel (Fig. 8.3.3a).

Example Ex. 8.3.2 demonstrates an example in which an M2T transformation creates an ordered execution tree of rule invocations and a hierarchically structured target text model.

```

1 «DEFINE C1 FOR EClass»
2   «IF interface == false»
3     «FILE getPackagePath() + name + "Impl.java"-»
4 package «getPackageName()».impl;
5     «EXPAND Im FOREACH importedTypes().reject(e|e.metaType.name != "ecore::EClass")»
6     «EXPAND classContent FOR this»
7     «ENDFILE»
8   «ENDIF»
9 «ENDDDEFINE»
10
11 «DEFINE Im FOR EClassifier»
12 import «getPackageName((EClass)this)».«name»;
13 «ENDDDEFINE»
14
15 «DEFINE classContent FOR EClass»
16 public class «name»Impl{
17   «EXPAND Pr FOREACH eStructuralFeatures»
18   «EXPAND Op FOREACH eOperations»
19 }
20 «ENDDDEFINE»

```

Listing 8.3.1: Xpand rules which generate a Java class declaration for a given `EClass`.

Example 8.3.2: M2T Execution Tree

Fig. 8.3.4 demonstrates the source and target of an M2T transformation of *Ecore* models to Java source code. The source `EClass` consists of the attribute `surname`, the derived attribute `name` and the reference `family`. The execution tree, depicted in the middle, results from applying the rules sketched in Listing 8.3.1.

The middle of the figure depicts the execution tree, which reflects how the execution proceeds by organizing the rule applications, which are typed over the rule names. For instance, a rule application which transforms an `EClass` is typed as `C1` and those transforming `EAttributes` are typed `Pr`. Inside the execution tree, solid lines mark graph edges, which represent the invocation of another rule on elements referenced by the source object whereas dotted lines reference the single source object and the created target objects in the order defined in the rule.

Firstly, the M2T engine executes the entry point rule (not shown in the listing), which is named `Root` and represents the single `EPackage db` by a directory. Secondly, the entry point rule iterates the classifiers contained in the source package in the order in which they are contained. The solid lines in the execution tree typed as `invk` represent the rule invocation and are ordered in the same way as the order of invoking source elements. Thus, in the example the first transformed classifier is the class `Family` and the second one the class `Person`. The root of the execution tree contains respective rule application elements. Moreover, to integrate the text of an invoked rule ϑ' , the invoking rule ϑ creates a block which serves as root to embed the text of the invoked rule.

The figure only depicts excerpts of applying the rule `C1` to the class `Person`. The rule defines the order of the created elements, referenced as `trg` by the corresponding element in the execution tree, as well as the order of invoking sub-ordinary rules:

At first, `C1` creates a file holding a block which states the package name as first fragment. The execution tree element `C1` records the block and the fragment as the second and third target element after the file. Next, the invoked rule named `Im` creates an import statement for each of non-primitive type used in the Java class. Thereafter, the class declaration is added as fragment and another block is created followed by a fragment that closes the class after the block.

For the text production inside the class, the rule `C1` invokes the rule `Pr` on each attribute of the source object `ucp`. The initially empty target block serves as root in which the rule will

```

1 «AROUND * FOR Object»
2 «getOpeningDirective()»
3 «targetDef.proceed()»
4 «getClosingDirective()»
5 «ENDAROUND»

```

Listing 8.3.2: Generic advice to enclose every text production with a preprocessor directive.

integrate fragments stating the attribute type and name if it is not derived. Conversely, if the attribute is derived, the rule will create a method declaration head as fragment followed by a block for the method body and a closing fragment. Similarly, also $EOperations$ can be transformed, hinted in the figure as rule invocations Op and Pa for their parameters.

Please note: Ex. 8.3.2 describes the creation of one directory for the source model. If multiple packages compose the input model, the execution tree becomes a forest of similarly structured trees. A (virtual) entry point rule application can serve as root comprising these trees resulting in a *single execution tree* and the target hierarchy of directories may be flattened to a single directory to regain a flat structure.

Model-To-Text Aspect Execution If an aspect weaver is present additionally, it may extend the *original* text production. Each time an object of the input model is processed, which matches the pointcut declared in an aspect, the corresponding implemented advice is executed instead of the *original* text-producing rule. Depending on its implementation, the advice may execute the original text production nonetheless but extend or embrace it with additional text.

A *generic* advice, such as the Xpand advice presented in Listing 8.3.2, can be utilized for embedding annotations. An around statement implements the behavior for embracing the source code with an annotation. By stating a wildcard ('*') as point cut (Line 1), we implement an advice which encloses the original text production of every object which matches a text-producing rule³. The advice has to look up the annotation of the source object in the mapping function, map_{FGS} of the multi-variant input model, MG_S , and assigns the annotation by stating it in an opening preprocessor directive before the original text production of the rule. After executing the original text production, the advice closes the preprocessor directive.

In the advice presented in Listing 8.3.2, the function in Line 2, serves as black box which implements the look up of the mapping function as well as the composition of the printed annotation, e.g., as a specific preprocessor directive. Similarly, the function `getClosingDirective` (Line 4) marks the end of the directive with a corresponding annotation. The instruction `targetDef.proceed()` executes the original text production which resides between the opening and closing directives stating the annotation.

Preprocessor Directive Depending on the capabilities of the target language, the opening and closing declaration stating the annotation may vary. For instance, if a preprocessor for GPLs is employed, the preprocessor may not allow for Boolean expressions in the directive or may not support hierarchical directives. Moreover, the concrete syntax of directives may vary with respect to the target language. As a consequence, the directive may become arbitrarily complex, e.g., to respect the hierarchy of blocks.

At an informal level, Fig. 8.3.2 demonstrates how the original text production is extended with preprocessor directives. The generic advice surrounds each text block which serves as root for the text production with preprocessor comments. In this example, exactly the annotation of the respective source object is mapped onto the block. For example, the class declaration is surrounded with the preprocessor directive `// #IFDEF Person #`. As Java source code is generated, the directive is represented as a Java comment which assumes a preprocessor which recognizes the phrase `#IFDEF` as the opening of a directive.

³ Producing text by invoking external methods, implemented in queries or in a GPL, does not support tracing and is not considered a point cut. Thus, these implementations do not trigger the execution of an advice.

8.3.2 Computational Model

Based on abstracting the source code generation and annotation propagation as execution and text trees, this section firstly summarizes the characteristics of a M2T transformation execution which uses aspects and creates hierarchically structured text. Then it defines the commutativity criterion for this type of transformations and derives properties for the aspect-oriented transformation which are necessary to satisfy commutativity, at an informal level.

Transformation Rule Behavior Based on the abstract example demonstrating how a M2T transformation creates a target tree (Ex. 8.3.2), we can summarize the following facts about M2T transformations. This behavior corresponds not only with the execution tree itself but also with its links to the target model.

1. Each application of a transformation rule creates an *ordered sequence* of nodes, representing fragments and blocks, and edges, which build a tree structure in the target model.
2. A transformation rule integrates the created tree under a given *root node* as sub-tree in the target model.
3. A transformation rule ϑ creates a *block* node without children in the target tree for each sub-ordinary rule ϑ' invoked by ϑ . This block node serves as root to integrate the sub-tree created by ϑ' .
4. A transformation rule creates a node in the *execution tree* for each invoked rule in the order of the processed source elements and expands the rule to create the respective sub-graph in the target model.

Commutativity To ensure the correctness of annotations propagated with the advice on top of the original text production, it is essential that the single-variant M2T transformation creates the same text fragments for a derived single-variant source model as are created for the same elements in the multi-variant source model. In concrete, Fig. 8.3.5 shows the corresponding commutativity diagram:

Let G' denote a single-variant model derived from the multi-variant model (MG_S) and H' be the source code tree resulting from transforming G' without injected aspects. Furthermore, let MH_T be the source code tree representing the annotated multi-variant text created by the aspect-oriented multi-variant model-to-text transformation and H'' be the source code derived from MH_T for the same feature configuration. If the single-variant text production Θ' generates the same text elements in H' as the multi-variant text production Θ_F for the same source objects, the same text elements will be present in H'' upon derivation with a preprocessor or similar functionality.

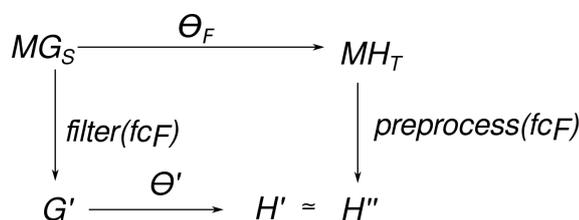


Figure 8.3.5: Commutativity of M2T transformations.

Consequences As a consequence, to satisfy commutativity, the single-variant transformation specification has to create the same text fragment when only a sub-graph of the multi-variant source graph serves as input. This has to hold regardless whether the object is contained in a multi-variant model or in a single-variant model derived from the multi-variant one. Consequently, it must be possible to embed the execution tree Θ' resulting from transforming G' into the execution

tree Θ_F resulting from transforming MG_S . Formally, a morphism embedding the single-variant execution tree into the multi-variant execution tree $m_\Theta : \Theta' \rightarrow \Theta_F$ must be a monomorphism.

Properties To guarantee commutativity, the multi-variant source model and the target tree as well as the text producing rules have to satisfy several properties. The following paragraphs regard the termination of the transformation first and continue with postulating properties to embed transformation rules. Finally, the advice and the mapping and filter mechanisms are considered.

Termination First of all, *termination* of the transformation has to be guaranteed. The following properties are necessary to satisfy termination.

Property 8.3.1: Unique Rule Application

A rule can be applied to a source node only once.

This criterion ensures that a rule cannot be executed multiple times on the same node provoking the creation of the same elements and potentially a non-terminating transformation. Accordingly, it also prohibits recursive rules being applied to the same source node. Thus, a source node may match different rules but may be processed for the *same* rule only once.

Property 8.3.2: Left Preserving

Applying a rule does not modify the source graph.

In this way, no new source elements are created which might provoke a non-terminating transformation. As a side effect, no source elements are deleted or modified which might change the result of the text production, too. This property should be supported by the syntax of a M2T transformation language natively because the language serves to create text as output and not a model which is the input. As such, functionality to manipulate the source model is not foreseen in those languages. However, when allowing to invoke helper methods written in a GPL, generally almost any side-effect could be implemented which might modify the source model and, thus, violate this property.

Property 8.3.3: Leaf Rules

Rules must exist which do not invoke any other rule.

Property 8.3.4: Finite Depth of Execution Tree

Each explored execution branch must terminate with a leaf rule (c.f., Prop. 8.3.3).

Leaf rules which are invoked at the end of each execution branch are one key factor to guarantee termination of the execution. Ending each execution branch with a leaf rule suffices to guarantee the termination of the transformation. Since the source model comprises a finite set of source nodes, which is not modified (Prop. 8.3.2) and a source node is processed only once by the same rule (Prop. 8.3.1), the set of matches for one rule is guaranteed to be finite. Moreover, due to the fact that the set of rules is finite and no new source matches are created, the set of *all* matches is finite as well. This fact together with the fact that rules cannot invoke each other in endless loops and must end with leaf rules ensures termination and a finite execution tree.

Embedding Single-Variant Into Multi-Variant Execution Furthermore, for embedding a single-variant execution tree into the multi-variant execution tree, the rules have to satisfy the

following properties.

Property 8.3.5: Functional Rule

Rules have to be functional: The application of the same rule to an equivalent source node constructs the same execution sub-tree and the same text sub-tree.

Although a rule is applied to a source node only once during one transformation, this property ensures that when executing the same transformation another time, the same execution tree and target sub-trees are created for the same input node. The property of functional rules carries over to the entire transformation.

Property 8.3.6: Local Rule

Rules have to be local: For the same matching source node the same execution and target sub-tree has to be created regardless of the surrounding elements.

As an example, locality prohibits if-conditions and queries inside the rule that depend on the presence of specific values of other source nodes.

Property 8.3.7: Monotonic Transformation Rule

Transformation rules may not delete elements in the target graph.

A non-monotonic rule could violate the effect of another functional rule because it may delete its created target elements.

If a rule is functional and local, it guarantees that the same target elements are created for the same source node of a single-variant or a multi-variant model. Together with the monotonic behavior the combination of the three rule properties is *sufficient* and guarantees that the execution sub-tree remains the same in the single- and multi-variant context for the same source node.

Please note: If a transformation language violates locality, in general, the annotation granularity and style may ensure commutativity, nonetheless. For instance, if application conditions, guards or if-conditions restrict the application of a rule only based on values of the structural features of the source node and the mapping mechanism only annotates nodes (not their structural features), the same attributes are present on the same objects regardless of any feature configuration. Consequently, the execution behavior cannot vary due to non-local rules caused by attribute-values of matching source nodes.

Furthermore, we consider source graphs which are ordered with respect to their children nodes. Accordingly, the execution tree is ordered in the same way as the matched source nodes and, as a result, the created text sub-trees are ordered as well in two dimensions: the *sub-trees in the execution tree* representing a rule application are ordered and the elements of the created *sub-tree in the text tree* are ordered in the sequence as prescribed in the corresponding rule.

Property 8.3.8: Order Preserving Transformation Execution

The transformation needs to be order preserving: Rules may not change the order of the source graph, the execution and the target tree.

Given Prop. 8.3.8, the orders remains stable even if elements or sub-trees are missing in between one of the children sequences of the execution and the target tree. Based on this necessary property, a missing source node, and consequently a missing node and potentially corresponding sub-tree in the execution tree, does not change the overall order of the execution tree. The same holds for a missing leaf node or sub-tree in the text tree.

Advice Besides the transformation rules and the execution, the advice which extends the execution of each rule has to guarantee similar properties.

Property 8.3.9: Graph Preserving Advice

The advice must neither delete elements from the execution and the text tree nor add elements to them.

In the abstract representation the advice updates the mapping annotation function but does not modify the generated text tree. Thus, the elements that are created in the text tree are not influenced by the aspect and only the properties of the rules and the transformation are decisive for embedding single-variant execution trees and the text trees into the multi-variant execution and text tree, respectively.

Property 8.3.10: Hierarchical Annotation

The annotation of the source object must be combined with the annotation of the target root node.

Accordingly, the text fragments produced by a rule are only visible if their container is present. Since the created text tree is hierarchically structured and created from top to bottom, it is guaranteed that for each text production an annotated root is present and that the created text has a container. This property propagates transitively from the root node to the leaf nodes.

Annotations and Filters have to adhere to the following properties.

Property 8.3.11: Coarse-Grained Annotation

Annotations must be mapped onto source objects only.

Thus, the annotation granularity corresponds with the mapping granularity of the target model, which only annotates the nodes. Structural features of source objects, which may restrict the application of a rule or be used to produce dynamic text, may not carry an annotation which is more restrictive than the annotation of the source object. Then, the same rules are executed for the single-variant graph as for the same nodes in the multi-variant graph.

Property 8.3.12: Order Preserving Filter

The filters must preserve the order of children nodes on the source graph and target tree.

This ensures that the filtered source models is a sub-graph of the multi-variant source graph and that the same execution tree, and as a result, the same target sub-tree are created.

Effect on Commutativity The interplay of these properties guarantees commutativity: The set of source nodes is transformed into a target tree and builds a corresponding execution tree. A rule is applied to each matching source node once (Prop. 8.3.1) which ensures together with the fact that the execution is finite (Prop. 8.3.2 - Prop. 8.3.4) that a well-defined hierarchically ordered target tree is created.

If the set of source nodes is restricted due to filtering, the single-variant execution builds an execution tree which can be embedded into the multi-variant execution tree. This is guaranteed because *rules* are functional (Prop. 8.3.5), do not delete already created elements (Prop. 8.3.7) and create the remaining sub-trees in the same order as in the multi-variant model (Prop. 8.3.8). Furthermore, the *advice* does not modify the target tree (Prop. 8.3.9). As a consequence, the

resulting single-variant target tree can be embedded into the multi-variant target tree. Furthermore, the transformation rules preserve the order of tree elements (Prop. 8.3.8) and the filter does not affect this order (Prop. 8.3.12), too. Finally, if the annotations are constructed hierarchically (Prop. 8.3.10), it will be guaranteed that the root nodes are present for creating adequate sub-trees in derived variants.

Since annotations are assigned to each target node created by a rule application and the same rule applications are executed in the single-variant as in the multi-variant derivation, the filtered target tree comprises the same elements in the same order as the tree resulting from executing the single-variant transformation on the derived source graph. Altogether, these properties are independent of a specific feature configuration and, thus, they ensure commutativity.

8.3.3 Formal Foundations

In contrast to a trace-generating transformation, an aspect-oriented source code generation behaves differently: Firstly, instead of a model, text is generated which can, however, be abstracted as hierarchical attributed tree consisting of nodes and edges as explained in Sec. 8.3.1. Secondly, the advice surrounds each text production for one source object with a preprocessor directive stating the annotation of the source object. We simulate this behavior by a mapping function where each created graph node and graph edge receive the annotation of the corresponding source object combined with the annotation of the root of the created sub-tree.

Road Map To present the foundations of aspect-oriented execution, the first part of this section defines the representation of the created single-variant and multi-variant text files as node-ordered hierarchical trees accompanied by a mapping annotation function. Based on these definitions, the following two parts explain single-variant out-place derivations of the execution and the target tree as well as the multi-variant transformation including the algorithm for the aspect-oriented propagation of annotations. The final part presents the commutativity criterion.

I Model and Text Representation

In the descriptions so far, the target of a transformation were models. In contrast, now ordered text is the target which we represent as an instance of a text metamodel and as such does not provoke the necessity to adapt the formalism significantly.

Model Tree Based on the considerations of Sec. 8.3.1, we restrict the previous representation of models as *graphs* to *trees* in this chapter. As explained in Sec. 8.3.2, the source and target representation are hierarchical graphs with ordered children nodes. Furthermore, these *trees* are not only typed over specific type systems but they are also *attributed*. Attributes of the source tree are of arbitrary primitive types whereas the attributes of the target tree are Strings and represent the generated text which is associated with a target node typed as **Fragment**. A data node carrying a label `text` and originating from a graph node typed as **Fragment** may serve to store the generated text which is required for converting the abstract representation as tree into a real file. The presence of a data node, however, depends on the corresponding graph node. Therefore, the annotation of the graph node only is decisive for the presence of data nodes and edges such that we refrain from representing them explicitly (e.g., as E-graphs [EPT04]).

Node-Ordered Trees As a consequence, we enrich the previous definition of hierarchical graphs (Def. 7.3.4) with ordering functions for children nodes of a container node.

Definition 8.3.1: Node-Ordered Hierarchical Graph

Let T_N and T_E be the finite sets of node types and edge types, respectively. A node-ordered, hierarchical graph over T_N and T_E is a typed graph, i.e., a tuple $G = (N, E, l_N, l_E, s, t, c, root, pred, succ, nOrd)$, where

- N is a finite set of graph nodes,

- E is a finite set of graph edges, where $(N \cap E = \emptyset)$,
- $l_N : N \rightarrow T_N$ and $l_E : E \rightarrow T_E$ are node and edge labeling functions, respectively,
- $s : E \rightarrow N$ and $t : E \rightarrow EL$ are source and target graph edge function, respectively,
- $c : E \rightarrow \mathbb{B}$ is a Boolean container function,
- $root \in N$ is the root node of the graph,
- $succ : N \rightarrow N \cup \{\epsilon\}$ is a direct successor function,
- $pred : N \rightarrow N \cup \{\epsilon\}$ is a direct predecessor function,
- $nOrd : N \rightarrow \mathbb{N}^+$ is a node ordering function and

The container n_p of a graph node $n \in N$ is unique and is accessible with a function $parent_E : N \rightarrow N \cup \{\epsilon\}$ (Def. 7.3.5).

The graph is acyclic, such that the transitive closure over all parents of a node does not contain the node: $\forall n \in N : \neg(n \xrightarrow{parent^+} n)$.

Furthermore, the predecessor and successor functions are acyclic, such that:

$$\forall n \in N \setminus \{root\} : \quad \neg(n \xrightarrow{pred^+} n) \quad (8.1)$$

$$\forall n \in N \setminus \{root\} : \quad \neg(n \xrightarrow{succ^+} n) \quad (8.2)$$

The node ordering function as well as the predecessor and successor functions together satisfy the following conditions:

$$\forall n \in N : \quad pred(n) = \epsilon \wedge succ(n) = \epsilon \Rightarrow nOrd(n) = 1 \quad (8.3)$$

$$\begin{aligned} \forall n_1, n_2 \in N : \quad parent_E(n_1) \neq parent_E(n_2) \Rightarrow \\ \neg(n_1 \xrightarrow{pred^+} n_2) \wedge \neg(n_1 \xrightarrow{succ^+} n_2) \end{aligned} \quad (8.4)$$

$$\begin{aligned} \forall n_1, n_2 \in N : \quad n_1 \xrightarrow{succ^+} n_2 \Rightarrow parent_E(n_1) = parent_E(n_2) \wedge \\ nOrd(n_1) < nOrd(n_2) \wedge n_2 \xrightarrow{pred^+} n_1 \end{aligned} \quad (8.5)$$

$$\begin{aligned} \forall n_1, n_2 \in N : \quad n_1 \xrightarrow{pred^+} n_2 \Rightarrow parent_E(n_1) = parent_E(n_2) \wedge \\ nOrd(n_1) > nOrd(n_2) \wedge (n_2 \xrightarrow{succ^+} n_1) \end{aligned} \quad (8.6)$$

Consequently, the children nodes are ordered and can access their direct predecessor and successor, if they share the same parent node (8.4) and a successor or predecessor are present. Moreover, the node order increases such that the order of a successor is higher than the order of a predecessor (8.5 and 8.6).

Node Appending For integrating a new node n_n as a child of some parent node n_p , a function $last : N \rightarrow N$ receives a parent node and returns the child with the highest order and which has no successor. The following descriptions assume that the source model, the execution model and the target mode, are node-ordered hierarchical graphs, which is in fact are trees.

Type System Furthermore, the types present in the text metamodel (c.f., Fig. 8.3.3a) restrict the type system of generated text trees. Accordingly, the sets $T_N = \{Directory, File, Block, Fragment\}$ and $T_E = \{files, children\}$ represent all possible node and edge types of a node-ordered tree, respectively, which reflects the text generated by a M2T transformation.

Multi-Variant Text Representation To integrate variability information into the generated text, we could surround the original text production with comments stating the corresponding annotation in the concrete syntax. Expressed in our formalism, annotations would be represented as further nodes of type *Fragment*. Instead, similar to the model formalism, a mapping function represents the association of an annotation with a source code fragment. Due to the hierarchical dependencies of the generated text as opposed to the model formalism, a source code fragment or block can only be integrated in the derived product, if all of its containers are also integrated in the derived product.

Accordingly, a multi-variant text comprises the text tree together with a mapping function for each of its elements, similar to a multi-variant graph defined in Def. 6.2.4. For the generated text graph, it suffices to map annotations onto graph nodes only. Graph edges connecting nodes are only present in filtered products if both, their source and target node, are present as well. Furthermore, the attribute nodes holding the actual text, which we refrain to represent explicitly, will be only integrated in a filtered tree, if and only if the graph node with which they are associated is integrated in the derived tree.

Model Filter The propagation of annotations assigns the annotation of the source node combined in a conjunction with the annotation of the source node's parent node. If the annotations that are provided by the propagation always combine the annotation of the source node with the annotation of the parent of the created target node, a *flat filter* (Def. 6.2.5) suffices to derive valid source and target trees which do not violate the tree properties.

II Single-Variant Out-Place Derivation

Similarly to trace-generating derivations based on STT-rules, a single-variant derivation creating a text tree is an out-place transformation. Instead of a trace graph an execution tree is built by these models. Accordingly, we still employ source-to-target graphs (Def. 6.3.5) but with more restrictive properties, such as being sub-trees and not arbitrary sub-graphs:

Definition 8.3.2: Source-to-Target M2T Graph

A source-to-target M2T graph (**STT M2T graph**) is a graph $M2T - G$ typed over node types $T_N = T_{N_S} \dot{\cup} T_{N_{Exec}} \dot{\cup} T_{N_T}$ and edge types $T_E = T_{E_S} \dot{\cup} T_{E_{Exec}} \dot{\cup} T_{E_T}$. It comprises three mutually exclusive node-ordered hierarchical sub-trees $M2T - G = G_S \dot{\cup} G_{Exec} \dot{\cup} G_T$ and mutually exclusive edge sets connecting these sub-trees.

More specifically, it is composed of the following elements:

- $G_S \subseteq M2T - G$: the source tree, typed over T_{N_S} and T_{E_S} .
- $G_T \subseteq M2T - G$: target tree, typed over the constant sets $T_{N_T} = \{Directory, File, Block, Fragment\}$ and $T_{E_T} = \{children, files\}$.
- $G_{Exec} \subseteq M2T - G$: execution tree typed over $T_{N_{Exec}}$ and $T_{E_{Exec}}$ whereby
 - execution tree nodes are typed over rule identifiers, i.e., $T_{N_{Exec}} = ID_\Theta$ for some rule set Θ .
 - the edge type set $T_{E_{Exec}}$ contains the edge types: $T_{E_{Exec}} = \{ivk\}$.
- Execution-tree-to-source edges of type *src* from execution tree nodes N_{Exec} to elements of the source graph.
- Execution-tree-to-target edges of type *trg* from execution tree nodes N_{Exec} to elements of the target graph.

We write $M2T - G = G_S \leftarrow G_{Exec} \rightarrow G_T$ to indicate that $M2T - G$ is an STT graph comprising the components as defined above.

Thus, in contrast to STT graphs, an STT M2T graph does not possess context edges. Furthermore, it consists of hierarchically, ordered trees where the target tree is typed over a constant set of text types. Additionally, *text-producing STT-rules* are more restrictive:

Transformation Rule The left hand-side comprises a single node in the source sub-graph which matches the single source object. Similarly, the left hand-side comprises a single execution node which represents either the entry point rule or the rule which invokes the current rule. Finally the target sub-graph of the left hand-side consists of a single target node only which represents the empty block created by the invoking rule. The execution and target sub-graphs of the right hand-side, instead, comprises the elements as created by the rule.

Definition 8.3.3: Text-Producing Source-to-Target Rule

Let $\vartheta = (L, R, \Theta_C)$ be a rule, where $L = n_S \leftarrow n_{Exec} \rightarrow n_T$ and $R = n_S \leftarrow R_{Exec} \rightarrow R_T$ are STT graphs and Θ_C the set of rules that are invoked by ϑ such that $\vartheta \notin \Theta_C$ and with $\vartheta_c \in \Theta_C$, such that $\vartheta_c = ((n_{S_c} \leftarrow n_{Exec_c} \rightarrow n_{T_c}), (n_{S_c} \leftarrow R_{Exec_c} \rightarrow R_{T_c}), \Theta'_C)$.
 ϑ is a text-producing source-to-target rule (tp-STT rule) if the following conditions hold:

$$\forall (L_c, R_c, \Theta'_C) \in \Theta_C : \quad \exists n_b \in R_T : n_b = n_{T_c} \quad (8.7)$$

Accordingly, the text-producing rule ϑ comprises an additional set of rules which are invoked by ϑ . One of its created nodes n_b serves as root for one of the rules in the set of invoked rules Θ_C . This definition, however, is silent on how the execution tree for the processed node is constructed which is defined in Def. 8.3.4.

Definition 8.3.4: Execution Tree Building tp-STT Rule

Let $\vartheta = (L, R, \Theta_C)$ be a tp-STT rule, where $L = n_S \leftarrow n_{Exec} \rightarrow n_T$ and $R = n_S \leftarrow R_{Exec} \rightarrow R_T$ are STT graphs.

Executing the following steps subsequently creates the execution-tree building STT rule $\vartheta' = (L', R')$ where $L' = n_S \leftarrow n'_{Exec} \rightarrow n'_T$ and $R' = n_S \leftarrow R'_{Exec} \rightarrow R'_T$:

1. Initialize ϑ' with ϑ : $\vartheta' := \vartheta$.
2. Add a single execution tree node n_e to the execution sub-tree R'_{Exec} . n_e is typed by the identifier id_ϑ of ϑ :

$$l'_N(n_e) = id_\vartheta \quad (8.8)$$

3. For the root node n_{Exec} , create an edge e' of type *ivk* from n_{Exec} to n_e :

$$l'_E(e') = ivk \wedge s'(e') = n_{Exec} \wedge t'(e') = n_e \quad (8.9)$$

4. Create node order:

- (a) If the root node n_{Exec} , already possesses children nodes labeled as *ivk* edges: retrieve the last child in the sequence, such that $n_{pred} = last(n_{Exec})$ and set it as predecessor of n_e and n_e as its successor:

$$pred(n_e) = n_{pred} \quad (8.10)$$

$$succ(n_{pred}) = n_e \quad (8.11)$$

- (b) If the root node n_{Exec} does not possess children, n_e becomes the first child:

$$pred(n_e) = \epsilon \quad (8.12)$$

$$succ(n_e) = \epsilon \quad (8.13)$$

5. Increase the order of the predecessor by one and assign it to the created node n_e

$$nOrd(n_e) = nOrd(n_{pred}) + 1 \quad (8.14)$$

6. Create an edge e' of type src from n_e to the source node $n_s \in N_S$:

$$l'_E(e') = src \wedge s'(e') = n_s \wedge t'(e') = e_l \quad (8.15)$$

7. For each new target node $n_n \in N_{RT} \setminus \{n_T\}$, create an edge e' of type trg from n_e to the target node n_n in the order given in R :

$$l'_E(e') = trg \wedge s'(e') = n_e \wedge t'(e') = n_n \quad (8.16)$$

Based on this definition execution tree building rules create *ivk*-edges which reflect the order of executed rules. Furthermore, the rules do not only create ordered invoke edges but also order the target edges. Ex. 8.3.3 demonstrates how the execution tree is built when transforming an **EClass**.

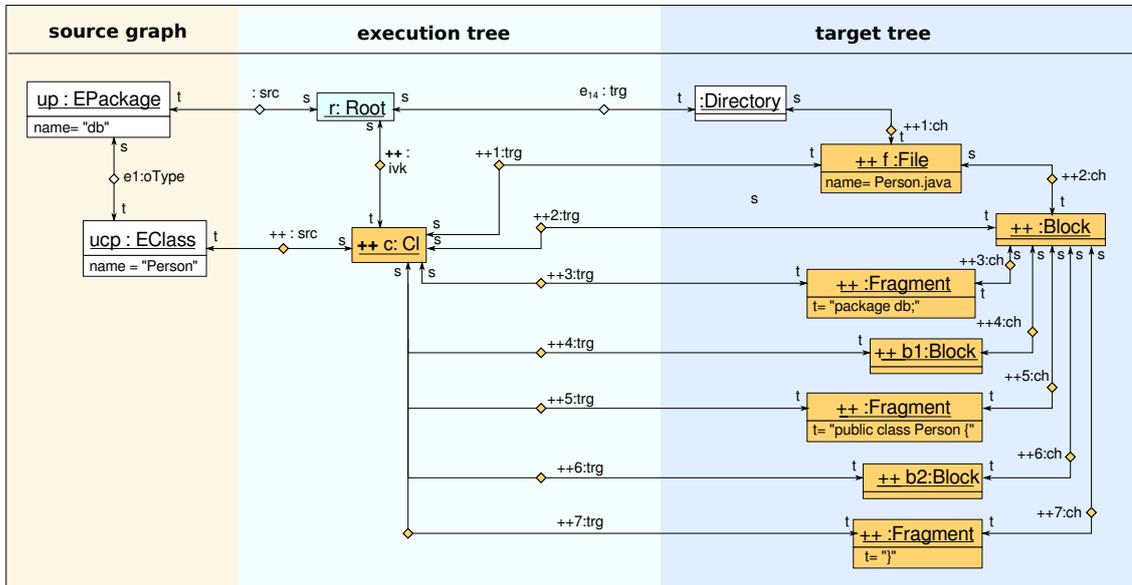


Figure 8.3.6: Execution tree building tp-STT rule (Def. 8.3.4).

Example 8.3.3: Execution Tree Building tp-Source-To-Target Rule Application

Fig. 8.3.6 demonstrates the first application of an execution tree building *tp-STT* rule which creates source code for a given **EClass**. The execution tree node labeled as *Root* serves as root for the execution sub-tree of the rule. Similarly, the *Directory* serves as root for the target side. The source node which matches the rule is the **EClass** *Person*. Therefore, the rule is executed on that class and behaves similar as explained in Ex. 8.3.2: it creates a file which states the package and the class declaration as fragments and integrates empty blocks for adding field and method declarations.

The execution tree building rule adds a new node to the execution tree, in addition. This node is labeled by the executed rule, i.e., it is named *Cl* in this example. The node references all created target nodes in the order in which they are generated with labeled target edges. Furthermore, a new *ivk* edge which origins from the root execution node *r* and targets the newly added node *c* is inserted. Finally, a source edge is added which

references the source node.

When the next class of the package is processed (i.e., the class *Family*), another execution node is integrated in the execution tree as child of the execution node r . Its order will be increased by one and the execution node c will record the execution node created for the class *Family* as its successor. The file and its contents which are created for the class *Family* will become the second sub-tree of the *Directory* node. Accordingly, the file f will be the predecessor of the file representing the class *Family*.

Text-Generating Transformation Based on the definition of tp-STT rules, Def. 8.3.5 defines a *text-generating derivation* as follows:

Definition 8.3.5: Text-Generating Derivation

Let $G = G_S \leftarrow G_{Exec} \rightarrow G_T$ and $H = H_S \leftarrow H_{Exec} \rightarrow H_T$ be tp-STT M2T graphs. Furthermore, let Θ be a set of trace-generating STT rules. H is STT-derivable from G if $G \xrightarrow{\Theta^*} H$ holds.

A complete derivation Θ^* is an ordered execution tree representing the order of rule applications and target element creations. The order of target elements is relevant in M2T transformations to integrate fragments or blocks in an overarching block. We simulate this behavior by ordering the children of a node, i.e., if a sub-tree is added it can only be added at the prescribed place.

III Multi-Variant Aspect-Oriented Text Derivation

An aspect-oriented derivation of text Θ_F propagates annotations by executing an advice on top of each rule application. Accordingly, the derivation behaves as described in Alg. 13. The algorithm employs recursion and ends when no source node is left to be processed.

Input and Output A set of source nodes N_S from G_S , the corresponding total mapping function map_{FG_S} and the target text tree H_T are input to the algorithm as well as the currently processed rule ϑ_r . While the source tree is total and not modified, the target tree is built subsequently during the execution as well as the mapping function for the target graph map_{FH_T} which represents the output. The execution tree is maintained by executing the execution tree building tp-STT rules. The propagation of the annotations and creating the target tree proceeds as follows:

Determining Target Elements and Target Root Node First, the algorithm computes the set of target elements created by the current rule in Line 16. Second, the algorithm determines the annotation of the root node n_T of the created target tree and stores it in a_p . If n_T is the root of the entire target tree, the annotation **true** will be stored in a_p .

After having extracted this information, the algorithm iterates the set of given source nodes. For each source node matching the type of source node specified in the rule it performs the following actions: The algorithm computes the annotation which should be assigned to all created target elements by combining the annotation mapped onto the matching source node with the parent annotation in a conjunction in Line 27. This ensures that upon filtering a text tree, a container is present for each pertained target node. Since the annotation propagation proceeds top-down the hierarchical order, similar to the container completion strategy (c.f., Alg. 4), a parent mapping will always be present (except for the root node of the target tree). Therefore, the annotations guarantee existence relationships in the text tree.

Thereafter, the algorithm executes the actual transformation of the source node. For each target element n_{trg} , the text production is performed which adds the element in the correct order to the target tree. Furthermore, it adds a node representing the rule application to the execution tree and *trg* edges which reference the created elements n_{trg} as target and as source the execution node. Afterwards, Line 30 maps the computed annotation to the created target element. Finally, the algorithm iterates the list of invoked rules and searches a match in the given list of source

Algorithm 13 Multi-variant aspect-oriented text derivation.

```

1: procedure PROPAGATE_ASPECT_ORIENTED( $N_S, map_{FG_S}, H_T, \vartheta_r, map_{FH_T}$ )
2:   in  $N_S$  ▷ ordered set of source nodes
3:   in  $map_{FG_S} : EL_G \rightarrow A_F$  ▷ Mapping annotation function of the source graph  $G_S$ 
4:   in  $H_T = (N_T, E_T, l_{N_T}, l_{E_T}, s_T, t_T, c_T, root_T, pred_S, succ_S, nOrd_S)$  ▷ Target text tree
5:   in  $\vartheta_r = (n_S \leftarrow n_{Exec} \rightarrow n_T, n_S \leftarrow n_{Exec} \rightarrow R_T, \Theta_C)$  ▷ Entry point rule
6:   out  $map_{FH_T} : N_T \rightarrow A_F$  ▷ Annotation function for the source code graph  $S$ 
7:
8:   var  $a_F \in A_F$  ▷ Annotation to be assigned to target elements
9:   var  $a_p \in A_F$  ▷ Annotation of the parent node
10:  var  $Trg$  ▷ Set of target nodes created by the rule
11:  var  $n_{trg}$  ▷ Created target node
12:  var  $W_{N_S}$  ▷ Ordered working set of source nodes
13:  var  $n_{src}$  ▷ Processed source node
14:  var  $\vartheta_c = (n_{S_c} \leftarrow n_{Exec_c} \rightarrow n_{T_c}, n_{S_c} \leftarrow n_{Exec_c} \rightarrow R_{T_c}, \Theta_{C_c})$  ▷ Invoked rule
15:
16:   $Trg := N_{RT} \setminus \{n_T\}$  ▷ Determine set of created target nodes
17:   $W_{N_S} := N_S$  ▷ Initialize ordered working set of source nodes
18:  if  $n_T \neq \epsilon$  then ▷ Compute the parent annotation
19:     $a_p := map_{FH}(n_T)$  ▷ Annotation of root of the target sub-tree
20:  else
21:     $a_p := true$  ▷ Root node has no parent
22:  end if
23:
24:  for  $n_{src} \in W_{N_S}$  do ▷ Iterate ordered working set of source nodes
25:    if  $l_N(n_{src}) = l_N(n_S)$  then ▷ Processed source node matches source node of rule
26:       $W_{N_S} := W_{N_S} \setminus \{n_{src}\}$  ▷ Remove source node from working set
27:       $a_F := map_{FG_S}(n_{src}) \wedge a_p$  ▷ Annotation of source node
28:      for  $n_{trg} \in Trg$  do ▷ Process all created target nodes
29:        EXECORGTXTGEN( $H_T, n_{trg}$ ) ▷ Add  $n_{trg}$  to  $H_T$  and update  $H_{Exec}$ 
30:         $map_{FH_T}(n_{trg}) := a_F$  ▷ Annotate the target element
31:      end for
32:      for  $\vartheta_c \in \Theta_C$  do ▷ Iterate the invoked rules
33:        if  $n_{S_c} \in N_S \wedge n_{T_c} \in Trg$  then ▷ Matching invoked rule
34:          PROPAGATEASPECTORIENTED( $N_S, H_T, map_{FG}, \vartheta_c, map_{FH_T}$ )
35:        end if
36:      end for
37:    end if
38:  end for
39: end procedure

```

nodes. If a match is found, the same algorithm is performed again with ϑ_c as input rule and the updated target graph and mapping function. Consequently, to ensure the termination of the algorithm, there must be text-producing rules which do not invoke other rules. If each execution branch ends with such rule, the execution terminates as well.

It must be noted that the transformation initiates with executing an entry point rule. We assume that the first applied rule creates a unique root, (i.e., this rule is processed only once) and comprises a set of rules that integrate nodes beneath this root node subsequently. Thus, in contrast to any other rule, the target side of L comprises only the empty element ϵ while the target side of R comprises the single root node. Only then, rules which mention the root node on the target side of L are processed subsequently, as described in the algorithm.

Properties Altogether, if a matching transformation rule is invoked, the aspect-oriented propagation algorithm assigns for each source node an annotation to the created target elements. Thus,

the transformation is total, if rules for each type of source node are invoked once in the transformation. Removing each source node which matched the set of rules once in Line 26 guarantees that a source node is not processed multiple times by the same rule (Prop. 8.3.1).

This also prohibits that the algorithm performs a recursive execution with the same source node. Since the invoked rules may be performed on the entire set of input nodes and the invoked rules may invoke other rules, the algorithm itself cannot guarantee that each execution branch ends with a leaf rule. However, we prohibit the usage of non-monotonic rules but define rules which do not alter the source model and which preserve and build a stable order of the execution and text trees.

Multi-Variant Derivation To this end, we define a multi-variant aspect-oriented derivation as follows:

Definition 8.3.6: Multi-Variant Derivation

Let Θ be a set of execution tree building *tp-STT* rules and $H = H_S \leftarrow H_{Exec} \rightarrow H_T$ be the STT graph resulting from a complete STT derivation starting with $H_S \leftarrow \emptyset \rightarrow \emptyset$. Let further $MH_{S_F} = (H_S, map_{F_{H_S}})$ be a multi-variant source graph and let $map_{F_{H_T}}$ be the annotation function resulting from employing the aspect-oriented propagation algorithm (Alg. 13) with N_{H_S} , H_T and $map_{F_{H_S}}$ as initial input.

Then, the multi-variant target graph $MH_{T_F} = (H_T, map_{F_{H_T}})$ is multi-variant derivable from the multi-variant source graph MH_{S_F} ($MH_{S_F} \xrightarrow{\Theta_F^*} MH_{T_F}$).

IV Commutativity

For proving commutativity, the text tree H_T'' resulting from the *transform-filter* path must be the equivalent to the text tree H_T' resulting from executing the *filter-transform* path, i.e., H_T' and H_T'' are equal up to isomorphism. Let N'_S be the *ordered set* of source model nodes in the filtered source graph G_S . H_T' results from applying a complete derivation, represented as execution tree Θ' , to N'_S starting from the same root-rule ϑ_r as the multi-variant complete derivation, which is represented as execution tree Θ_F^* .

Embedding H_T' in H_T Each rule is processed on each matching source node and for each processed source node a node in the execution trees is built. Thus, each processed source node in N'_S is referenced by some execution node in Θ' . Additionally, the same nodes are also processed in the multi-variant transformation and initiate the creation of one execution node each in Θ_F^* which reference the matching source node. Furthermore, the rules are functional, local, monotonic and order preserving. Thus, independent of the values of their structural features, different execution runs compute the same target sub-tree, which is unique up to isomorphism, for the same input source node. Consequently, the nodes of the execution tree Θ' reference ordered target sub-trees which are equal up to isomorphism to sub-trees referenced by the nodes corresponding with the same source object in Θ_F^* . Since rules can invoke sub-ordinary rules non-recursively resulting in acyclic execution trees, the presence of a node in the execution tree means that also a parent node is present which reflects the rule that invoked the rule corresponding with the child node.

For that reason, the execution tree $\Theta_F|_{N'_S}$ tracking the transformation of the filtered source node set, possesses the same execution nodes, in the same order and with the same target trees as are present in Θ' . Accordingly, a monomorphism $m_\Theta : \Theta' \rightarrow \Theta_F^*$ exists which allows to embed the single-variant execution graph into the multi-variant one.

Propagated Annotations So far, the properties ensure that the same target sub-trees are present in H_T as in H_T' but they do not guarantee that they are present in H_T'' . By applying Alg. 13 for each of the processed source nodes of N'_S its annotation combined with the parent annotation is mapped onto the created target sub-tree. Since the source nodes in N'_S are present in N_S and processed by Θ_F^* , each of the created target sub-trees receives the annotation as assigned

to the matching nodes combined with the parent annotation. Accordingly, when filtering H_T by the same configuration, these sub-trees remain in the filtered text H_T'' . However, this does not guarantee that no other elements are integrated in H_T'' .

If there were other elements in H_T annotated with an expression, composed of their source node annotation and the annotation of their root node, that is satisfied by the feature configuration those would be added to H_T'' , additionally. Accordingly, there would be a matching source node processed by Θ_F which is removed by the source filter and therefore not included in N_S' . If it were present in G_S , it would be present in N_S' and processed by the total derivation Θ' and its target elements would be part of H_T' . This would imply that the combined annotation assigned to the target elements would be less restrictive than the annotation of the source node.

Let a_s be the annotation of the source node and a_r the annotation of the root node of the target sub-tree that passed the filter. If a_s cannot be satisfied by a feature configuration, i.e., it evaluates to **false**, the conjunction $a_s \wedge a_r$ assigned to the target sub-tree cannot become true under the same configuration which contradicts the assumption. As a consequence, only the target sub-trees present in H_T' is present in H_T'' .

Well-Formedness Finally, since the trees are ordered, it is essential that not only the same elements are present in H_T' and H_T'' but also in the same order (not necessarily with the same order-value).

Transformations and filters preserve the order of elements and the rule execution (Prop. 8.3.8 and Prop. 8.3.12), respectively. Consequently, the order of the elements remains stable even if some elements (e.g., in the sequences of children nodes) are missing. The order invariants defined in Def. 8.3.1 (Equation 8.3-Equation 8.6) remain valid.

Since executing a rule may create an entire sub-tree, all elements created for the matching source node receive the annotation of the source node combined in a conjunction with the annotation of the root node of the sub-tree. As a consequence, a node can only remain in the filtered target model if its parent is present as well which ensures the existence-relationship of trees.

8.4 Discussion

To conclude this chapter, the following sections elaborates on the benefits and restrictions which need to be considered when M2T transformations are enriched with variability information by an aspect-oriented approach. Accordingly, Sec. 8.4.1 discusses the properties of the computational model with respect to real-world M2T transformations whereas Sec. 8.4.2 illuminates alternatives how to annotate the behavioral representations of a product line and includes a discussion of the cognitive complexity of a multi-variant text representation.

8.4.1 Computational Model

In general, certain restrictions of the computational model presented in Sec. 8.3.2 can be discussed and may be relaxed in specific situations.

I Transformation Properties

Unique Rule Application Prop. 8.3.1 postulates that the same node is processed by the same rule only once in order to ensure termination. While the example of an execution tree, depicted in Fig. 8.3.4, shows that a source node may be referenced as source node by several execution nodes, it cannot be targeted as source element twice by the same rule type.

As a consequence, firstly, a leaf rule could be processed multiple times on the same node as long as the process terminates, in general. Since this kind of recursive behavior typically depends on a termination condition which must be met at some point, this dependency leaves room to a violation and a non-terminating execution. Secondly, if the same rule is executed on the same source node multiple times until a condition is satisfied, it is not guaranteed that this conditions holds in a single-variant and multi-variant model in the same way provoking different target trees to be created. Therefore, the property is integrated in the computational model.

Annotation Computation For ensuring well-formedness upon filtering and a safe embedding of the single-variant into the multi-variant text production, the annotation of the source node is combined with the annotation of the target root node of the invoking rule. In practice, this would mean to access the text element created for the source node of the invoking rule and to look up its annotation during the aspect-oriented execution. A generic aspect which generates arbitrary text is not able to access this information. On the contrary, the aspect looks up the annotation of the element containing the source node (i.e., the parent annotation). Thus, the annotations can only guarantee correctness if rules are invoked on children nodes, stored in a containment hierarchy only. For any element apart from that (i.e., cross-referenced nodes), a different annotation may be mapped onto the actual container that is accessed.

Completeness of Annotations We assume that the given source model is completely annotated such that the mapping annotation function for the source model is total. Even though the formalization does not regard edges, the approach assumes that the presence of an edge depends on the presence of the two nodes it connects.

Consequently, Alg. 13 propagates annotations to all target nodes of the right hand-side of the text-producing rule. Since all source nodes are annotated, annotations are mapped onto the nodes of all created sub-tree elements after the aspect-oriented execution.

II Granularity and Style of Annotations

The generic aspect invokes two methods which serve as black box to provide the opening and closing directive for embracing the original text production with an annotation. Furthermore, Listing 9.3.2 exemplifies the style of a resulting text production which integrates preprocessor directives into Java source code. The enclosing directive, which represents the annotation, needs to respect the capabilities of the engine (e.g., the preprocessor) which filters the generated text:

Annotation Syntax First, the *syntax* of opening and closing directives may vary, such as opening the directive with a statement `#IF`, `#IFDEF` or `#ifdef`. For instance, the C/C++/Objective-C preprocessor employs the opening directives, `#if`, `#ifdef` and `#ifndef`. While the first can be followed by a Boolean expression over Macros and comparable primitive types, the latter two are shortened versions of “if defined” and “if !defined”. Thus, the annotations have to be adapted to the annotation style in realizations of the black box methods.

Complexity of Annotation Expression Second, the complexity of the expression following the opening directive keyword, may vary and either state only a single macro (representing a feature in our use case) or a Boolean expression over them. For instance, if the preprocessor does not support AND-expressions, it may be necessary to nest the directives, which requires the respective support by the preprocessor, too. Vice versa, if nesting is allowed, the computed annotation may not have to be combined with the respective root annotation of the text production (as proposed in our computational model) but it can be nested in the outer directive.

Artificially Added Preprocessing Third, if a GPL without built-in preprocessor directives is used, such as Java, it is necessary to embed the directives in a form that does not threaten the compilation. Accordingly, in Java we employ (JavaDoc and single-line) *comments* which requires to use a customized preprocessor style in order to distinguish the preprocessor comment from a regular comment. Finally, a preprocessor per se is unaware of variability. Typically, the features or similar constructs of a variability model are transformed into preprocessor macros and states are assigned to them according to a feature configuration. Therefore, the validity of the feature configuration has to be checked before deriving a customized text representation.

Annotation Mapping Instead of Text Fragment Altogether, we did not integrate the annotating directives as elements inside the target tree but abstracted them in form of a mapping. Thus, we abstract from a specific target language because not every language compiler encompasses

preprocessing capabilities. Furthermore, not necessarily source code has to be generated but any kind of text (e.g., XML-files) which might require annotations, too.

Arbitrary Hierarchical Text The descriptions so far discussed generating and mapping annotations onto *source code* as well as the conditional compilation of a preprocessor for deriving products. As a first fact, even though *arbitrary text* can be generated, naturally template-based M2T transformation specifications create a text which is ordered and hierarchically structured according to the rule execution semantics. Thus the descriptions in this chapter can be generalized for the text production of any kind of text performed by a M2T transformation. In contrast, it depends on the target representation whether constructs similar to preprocessor directives can be embedded. If the target text conforms to a grammar, the preprocessor style should employ some escape characters which allow to integrate annotation without violating the grammar. For instance, in an XML file still comments could be provided.

Tracing Via Unique Identifiers Alternatively, as used in the proposed formalism, the aspect could construct a map instead which is capable to associate text fragments uniquely with annotations. The unique identification requires to provide a unique ID to the text fragment in a non-invasive way at the least. Furthermore, a parser needs to identify when the text fragment corresponding with one ID starts and ends. Then, the map can serve as input to derive a customized text representation by removing the fragments the annotations of which are not satisfied by the respective feature configuration. As such the map can serve as a replacement for the trace information. Depending on the time of *assigning the annotations*, i.e., during the execution by employing the aspect, or after the execution by employing the map distinguishes the aspect-oriented from the trace-based propagation.

8.4.2 Related Work

When discussing the integration of annotations into text, particularly into source code, some questions about the feasibility may arise. Therefore, this section discusses, how to integrate behavior in structural models instead, which may obviate the need for a multi-variant source code platform. Furthermore, Xpand and aspect-orientation was used to develop product lines. This section differentiates this existing from our approach. Finally, the level of cognitive complexity rises if multi-variant source code with different execution branches is created which is discussed at the end of this section.

Integration of Behavior in Structural Models In the motivating example, as well as for any other source code generating transformation, it would be possible to provide the behavior in form of another behavioral model, such as an ALF [Obj17a] model. These kinds of models may be beneficial to complement structural models with behavior, such as the logic implemented in method bodies. However, if arbitrary text is created which requires manual modifications, these models will not suffice in general. For that reason and because this work focuses on structural modeling, we do not consider behavioral models as complementary to text productions.

One solution integrates Java method bodies in Ecore model as annotations of method stubs [BS15]. The authors employ a T2M transformation which parses Java source code and turns it into a Java model. Subsequently, they iterate the methods defined in the Ecore model and search for method headers in the Java model which match the iterated method heads. If they match and a body implementation is part of the Java model, a M2T transformation converts the method body of the Java model into Java source code. The resulting text is integrated as annotation in the Ecore model. Thus, behavior is injected into the structural model similar as manually assumed in Sec. 8.1.1. However, variability of a product line is not considered: As a result, first, annotations would have to be mapped onto the Ecore model as well as on the implemented method bodies manually because the code generation of Ecore is unaware of variability. Then, the method bodies could be reintegrated into the Ecore model including the manually provided annotation. However, this approach cannot suffice the automation and genericity criteria for multi-variant model transformations.

Usage of AOP as Compositional Variability Mechanism The aspect-oriented programming capabilities of Xpand are employed in another approach to develop product lines [VG07]. However, this approach uses the capabilities to realize a compositional variability mechanism where each feature is realized in one aspect. These aspects will be added to a common code base if the a feature configuration selects the feature which they realize. Sec. 4.3.3 discusses the entire concepts of employing the oAW framework for developing a software product line [GV09]. However, the authors do not employ the aspect-oriented technique to propagate annotations. Instead, the approach proposed in this chapter employs a single aspect and realizes an annotative variability mechanism.

Cognitive Complexity One argument against creating multi-variant source code is the increased level of cognitive complexity. With an increased number of conditions in the source code it becomes hard for the developer to understand how the program behaves in different configurations. However, adequate tool support, such as the tool CIDE [Käs10] exists which can help to conditionally compile the source code platform and to show only single variants. Similarly, coloring different features may speed up and positively affect understanding variable source code [Fei+13]. Thus, the complexity can be hidden from the user by respective tool support. Furthermore, our approach integrates the annotations *automatically* into the source code such that the developer does not need to map annotations onto the source code fragments.

Our evaluation of automating the annotation task (c.f., Sec. 10.3.5) shows that the effort for mapping annotations onto target elements is (almost) reduced completely. Thus, the level of cognitive complexity to annotate the generated source code decreases in the same size due to the automated propagation of annotations.

8.5 Summary

All in all, this chapter illustrated the problem of deriving source code variants from structural models and motivated the necessity to build multi-variant source code platforms. Since these platforms require annotations mapped onto text fragments (instead of model elements), too, the chapter derives a computational model for propagating annotations in an aspect-oriented approach. For the derivation, the sections first illuminated the behavior of template-based M2T transformations and aspect-oriented programming as well as the requirements to propagate annotations in these scenarios. As a result, the formalization represents transformations that build ordered execution and target trees and assigns preprocessor directives by mapping the annotation onto all elements contained in the created sub-trees. Finally, Sec. 8.4 discusses the impact of creating annotated multi-variant source code and exposing it to the developers as well as the usage of preprocessors, particularly, when they are not integrated in the execution semantics of the target programming language which requires disciplined annotations.

Part V
Validation

Chapter 9 Implementation

*Most people find the concepts obvious,
but the doing impossible.*

Alan Perlis

~

The previous chapters of this thesis contribute approaches to propagate annotations from a source model to a target representation at a *conceptual* level. This chapter presents how we implemented corresponding prototypes as proof of concept. Accordingly, this chapter serves the purpose to demonstrate how the concepts can be realized and employed in the development of model-driven product lines. In this way, the chapter also lays the grounds for the following chapter which evaluates the contribution of the thesis.

The subsequent presentation of the implementation, however, considers only the approaches which have not been rendered as impractical or of restricted usefulness during their introduction. Accordingly, it demonstrates the trace-based propagation based on traces of each kind of completeness stage, particularly of completing annotations in sight of incomplete traces and of analyzing the ATL/EMFTVM bytecode model in sight of coarse-grained traces. While the the ModelSync and model matching strategies, which may be employed if no trace is available, are not integrated in the multi-variant transformation framework presented in this chapter, the chapter also explains the implementation of the aspect-oriented propagation in M2T transformations.

The chapter is organized in the following way: The first section of this chapter presents background information on employed existing technologies and frameworks on which the prototypes rely mostly whereas Sec. 9.2 presents an overview of the architecture of the multi-variant model transformation framework. The implementation of the trace-based propagation, the bytecode instruction analysis and the aspect-oriented propagation are detailed in Sec. 9.3 while Sec. 9.4 summarizes the contributions of the chapter.

between class and interfaces, the latter of which are represented as a separate `EClassifier` in UML.

Furthermore, in contrast to UML, references conform to more restrictive semantics than associations. An `EReference` is always unidirectional. A bidirectional reference is built from two unidirectional ones by setting the `eOpposite` field accordingly. The model supports neither n-ary references nor association classes. Strict containment references foster a unique container, prohibit cycles and build an existence dependency. A *contained* `EObject` (i.e., an instance of one of the defined `EClasses`), can only be present in a model instance, if and only if its container is present. For example, in the Ecore metamodel, depicted in Fig. 9.1.1, an instance of a structural feature must be contained in an instance of the `EClass`. Two unidirectional references model this relationship: The meta attribute *containment* of the reference pointing to the structural feature and name `eStructuralFeatures` is set true and its type is the class `EStructuralFeature`. Its `eOpposite` is the second reference, named `eContainingClass` where the *containment* is set false but the derived meta attribute value `container` is set true.

Source Code Generation For each Ecore model conforming to the Ecore metamodel, a `GenModel` can be created with the EMF. This model stores specific information to generate, and potentially customize the generation of, Java source code from the metamodel.

The built-in M2T transformation engine creates one package which contains interfaces for each `EClass` defined in the metamodel and one implementation package which comprises the corresponding class declarations which implement the interfaces. Furthermore, a package `util` contains a model-specific factory to create model elements conforming to the declared metamodel as well as further facilities to maintain the model elements in Java programs. Besides these basic realization artifacts, the code generator can also create `edit` and `editor` source code which allows to build customized editors to build corresponding models and which can be deployed as Eclipse plugins. Based on the generated source code, instances of the defined model can either be created by using the editor or the factory provided in the `util`-package. Furthermore, the *reflective API* allows to query the meta-properties of a model element, for instance, the name of the model `EClass` or the resource in which the `EObject` is contained.

Resource Management Besides the possibility to generate source code and editors, the EMF allows to persist models with its own resource managements API. A `ResourceSet` manages all models represented in its collected resources. In order to recognize the type of models, the corresponding model `EPackage` has to be registered in the resource set. Based on this information and a given resource URI, models can be persisted or loaded and modified.

9.1.2 Famile

As shortly introduced in Sec. 4.1.2, *Famile* [BS12] is an Eclipse-based tool supporting the development of annotative model-driven software product lines. Famile realizes an external mapping mechanism and builds on the following three main components: Firstly, a feature model expresses the commonalities and differences of the product line to be built. Secondly, arbitrary domain models, the metamodels of which conform to the Ecore meta-metamodel, can be used to design and realize the product line. Thirdly, *Feature to Domain Mapping Models* (**F2DMMs**) allow to map annotations, which conform to the feature model and are referred to as *feature expressions*, onto a domain model which they reference. Further, the tool functionality involves the creation of feature configurations, applying them to a F2DMM and derive a customized model variant. The employed hierarchical filter allows to define how to handle missing annotations and the F2DMM model may be enriched by a SDIRL model which is a DSL prescribing domain model specific repair operations in case model well-formedness would be violated by filtering a variant.

Feature Model The Famile offers a feature model editor to build feature models conforming to its respective metamodel. A feature model may encompass optional and mandatory *atomic* features and may organize them in feature groups. The feature group determines how many of its child features can be selected simultaneously. Thereby, it allows for representing OR, XOR and

AND groups. Furthermore, the model may relate features in requires- and excludes-dependencies apart from the hierarchical tree structure. A particular characteristic of the feature model meta-model is that it is also used for representing feature configurations. Consequently, each atomic feature and feature group possesses a selection state which is used to represent a complete feature configuration. Lastly, the feature model allows for cardinality-based feature modeling [CHE05], too. Thus, *attributes* which define an upper limit for its value can refine an atomic feature.

Feature To Domain Mapping Model Together with a domain model, the feature model is referenced by the F2DMM model. The F2DMM model assumes exactly the same hierarchical structure as the domain model for which it provides annotations. Thus, the root, the `MappingModel`, comprises a tree of `ObjectMappings` for each `EObject` of its corresponding domain model and organizes them in the same hierarchical order internally. As Famile allows for fine-grained mappings, the F2DMM encompasses mappings for the structural features, too, denoted as `AttributeMapping` and `CrossRefMapping`. Further functionality allows modeling beyond the capabilities of single-variant models by employing `AlternativeMappings` which can declare a different value for a structural feature. However, this functionality goes beyond the scope of this thesis which assumes single-variant semantics (Prop. 4.1.2) and is not further regarded.

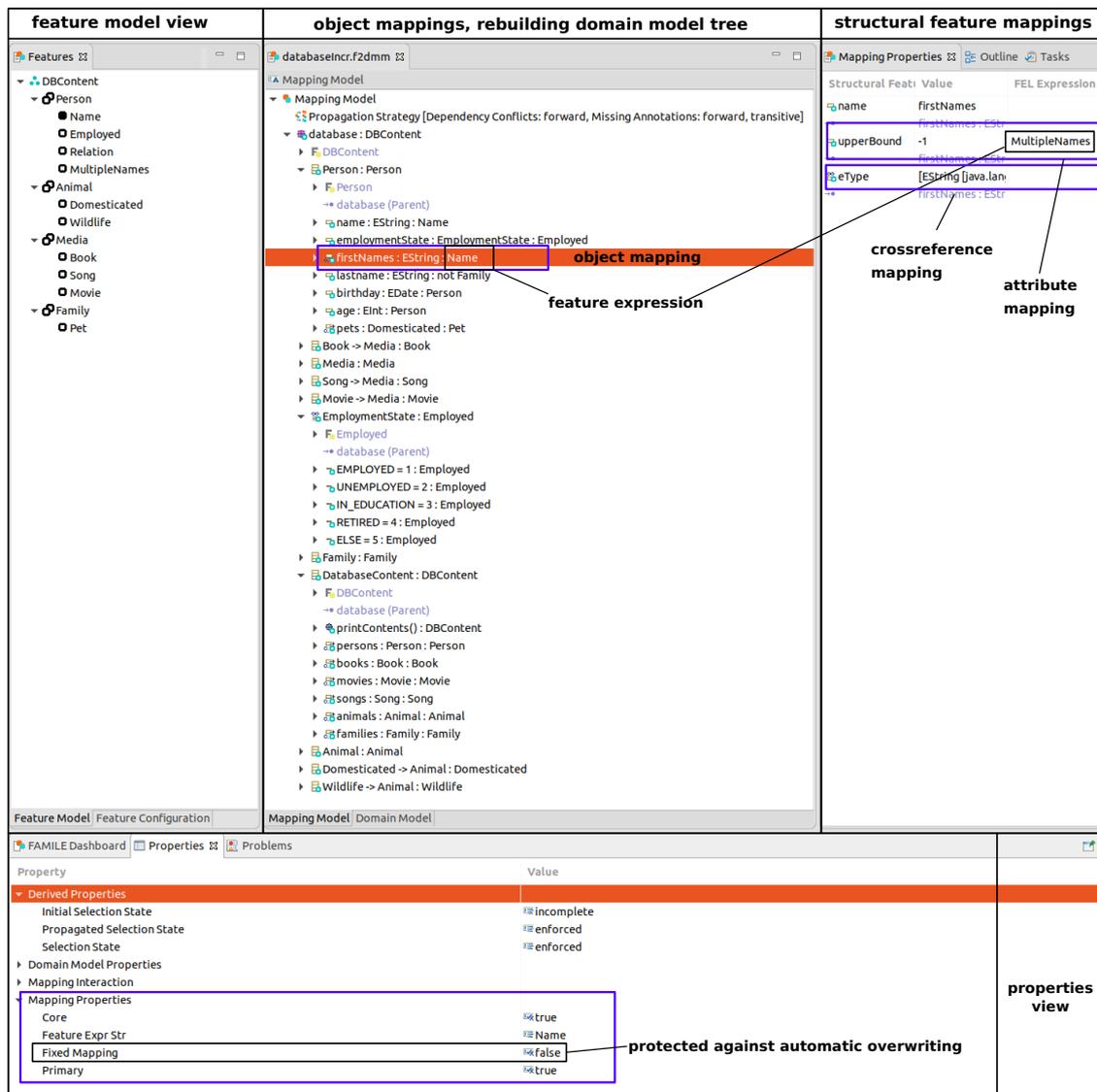


Figure 9.1.2: Famile mapping model editor.

Fig. 9.1.2 displays the mapping editor of Famile, which depicts the F2DMM model for the Ecore model representing the database product line introduced in Fig. 4.1.3. The left column of the editor shows the feature model, which is referenced by the F2DMM model situated in the middle of the editor. The feature model cannot be modified in this editor but used to drag a feature to map it onto an object of the F2DMM model. The right side shows mappings for structural features of an object. In this example, the highlighted object `firstNames` is the object which is of type `EAttribute`. Its annotations, the feature expression `Name`, is stated behind the respective object in the middle of the editor. The structural features of the `EAttribute` are shown on the right side. The attribute `upperBound` holding as value `-1` carries the refined annotation `MultipleNames`, as exemplified in Sec. 7.2.2. Below this attribute mapping, the feature `eType` demonstrates a crossreference mapping which is not annotated in a refined way.

The bottom of the editor incorporates the *F2DMM Properties* view, which lists important properties of the selected object mapping. Every mapping in the product lines we examined, is a *core* mapping, which means it forms part of the mapping model per default. Conversely, an *AlternativeMapping* is no core mapping. Besides the core mapping property, the essential properties of the object mapping is the attached feature expression, represented as String. The field whether the mapping is *fixed* is an extension to the mapping model made for the purposes of this thesis. After manually modifying an annotation, the developer is asked whether to pertain this annotation. In this way, an incremental transformation cannot change a fixed mapping. Additionally, the view mentions the selection state of the selected mapping for a given feature configuration. This is only possible if additionally a feature configuration is applied to the F2DMM model otherwise it remains incomplete (as in the figure). Internally, the F2DMM model is rebuilt and the selection state computed each time the mapping model, the feature model, or the domain model changes. Thereby, the correctness of an annotation (i.e., the analysis whether the annotation can be expressed with the contents of the feature model), is conducted based on the custom FEL-DSL for representing and analyzing feature expressions in Famile.

Adaptions to Famile Besides the possibility to fix manually modified mappings (see previous paragraph), for realizing the concepts of the thesis the following further adaptations have been made to Famile. The F2DMM editor highlights objects, onto which no annotation is mapped, as depicted in Fig. 9.1.3. In this example, the database Ecore model is not annotated completely yet. The excerpt shows that the enumeration literals of the `EmploymentState` enumeration as well as the attribute `species` of the class `Animal` still miss annotations. Highlighting the mapping elements without annotations helps the developer to identify these elements easily and to map an annotation onto them.

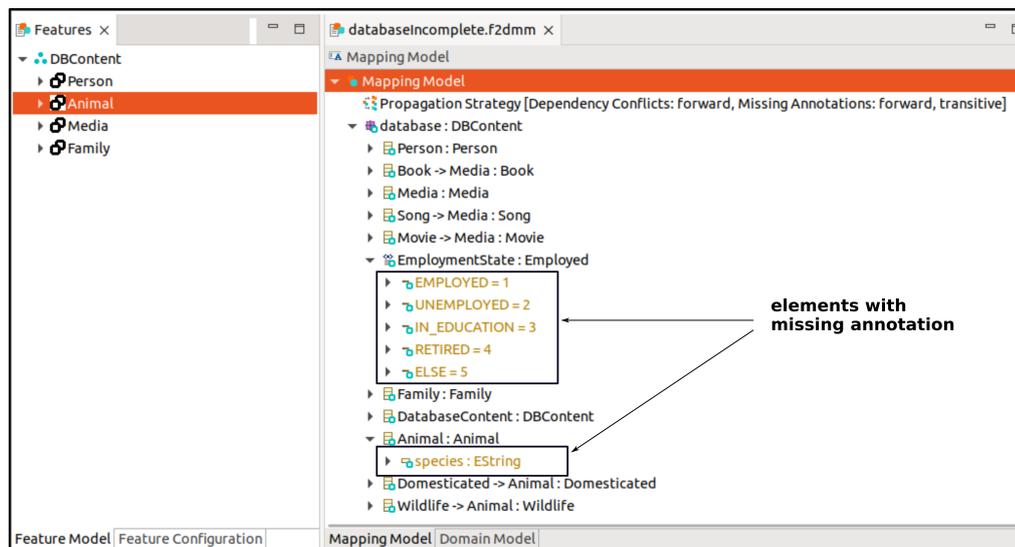


Figure 9.1.3: Database Content mappings with missing annotations in F2DMM editor.

9.1.3 Delimitation

The following descriptions will present implementation information about the central contributions of this thesis. We do not present implementations for concepts, which were rendered as impractical or not able to achieve a high amount of correctly and automatically propagated annotations. In concrete, we do not integrate implementation details for the ModelSync DSL which would be of the most practical use when combined with an automated detection of correspondences (e.g., generated by model matching). However, preliminary evaluation results of a Bachelor thesis [Hoe20] have shown that the automated detection, based on the model matching framework (c.f., Sec. 9.1.3, II), is not capable to recognize even matches between simple models if the structure of the compared models deviates: When comparing the similar project management [Ker03] metamodels representing Gantt diagrams and CPM networks, a matching algorithm based on attributes, references and types, was unable to detect corresponding elements sufficiently.

9.2 Multi-Variant Model Transformation Framework

The following sections present the architecture and key aspects of realizing the contributed conceptual approaches for propagating annotations in multi-variant model transformations. The concepts share the ability to execute the reused single-variant transformation and to employ a propagation mechanism which transfers the annotations from the source model to the target representation. This commonality as well as the concrete approaches are implemented in the framework for *Multi-Variant Transformations*, denoted as *MuVaTra*, in the sequel.

Road Map To realize and evaluate the feasibility of the propagation strategies which the previous chapters introduced, we implemented several Java projects. The following paragraphs give an overview of the implemented packages (Sec. 9.2.1) and their relationships, firstly. The subsequent section, Sec. 9.2.2, illuminates the core of the framework in more detail while Sec. 9.2.3 explains how to realize the interfaces based on an implementation for the tool Famile. This implementation lays the grounds for evaluating the propagation strategies in Chp. 10.

9.2.1 Overview

The package diagram depicted in Fig. 9.2.1 offers a simplified overview of essential projects and packages which realize the concepts introduced in the previous chapters. The packages of white color symbolize standalone projects, with equally named main packages, and the packages of orange color represent contained packages. Furthermore, the two blue colored interfaces represent Eclipse extension points. We refrain from demonstrating all implementation details, such as exceptions and utility classes, and focus on key classes and functionality.

The framework is built around the `muvatra` core package, which abstracts the basic functionality and is extended by further implementations in referencing packages but also by offering an extension point for reusing a single-variant transformation execution mechanism. While the two packages displayed in the top row of the figure represent core functionality which is independent of a specific propagation approach, the packages denoted as `execution.traces`, `execution.missingAnn` and `execution.ruleAnalysis` represent projects that allow to propagate annotations based on traces of arbitrary granularity written by arbitrary transformation engines. Similar as the `SVTransformer`, the `TraceConverter` interface serves as Eclipse *extension point* which can be extended to plugin the capabilities of a specific transformation engine. Finally, in the right bottom corner resides the packages which represents the implementation artifacts for execution the aspect-oriented propagation.

Please note: We split the single-variant transformation and the propagation to allow for separation of concerns. However, for black-box and gray-box post-processing propagation approaches, it might not be necessary to integrate the execution of the single-variant transformation because it can also be executed manually. Similarly, in an inter-processing approach it might not be possible to distinguish the reused transformation execution and the propagation as it is the case for the aspect-oriented propagation.

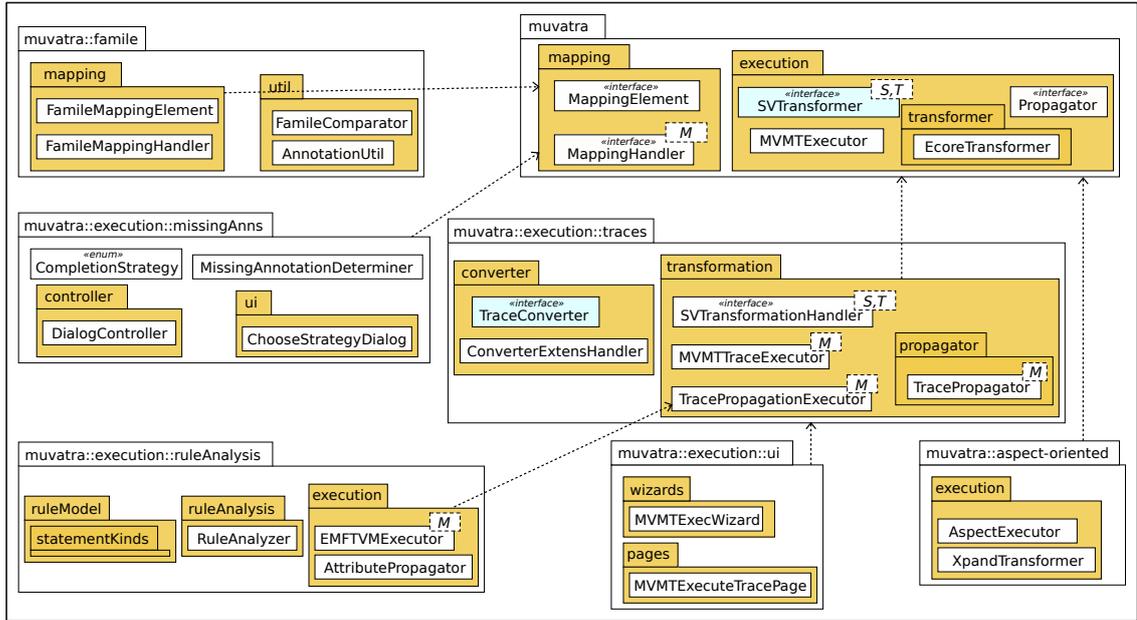


Figure 9.2.1: Overview of architecture of MuVaTra framework.

MuVaTra Core Package The `muvatra` core project offers the interfaces for executing the reused single-variant transformation and for propagating annotations. This package is explained in more detail in Sec. 9.2.2. Furthermore, the project `muvatra::transformer` summarizes basic functionality to execute EMF-based M2M transformations.

To abstract from a specific mapping approach, the core package further comprises a package `mapping` which offers an interface for representing an arbitrary mapping element and a generic `MappingHandler` which maintains concrete realizations of the `MappingElement` interface. To this end, the `MappingElement` serves as an *Adapter* [Gam+97] for propagating annotations to the concrete mapping realization and offers functionality, such as accessing the mapped model elements, its annotations as well as the annotations of its parent and children mappings.

MuVaTra Famile Package The `muvatra::familie` project comprises a concrete realization for mapping elements of F2DMM models, as explained in more detail in Sec. 9.2.3. In addition, the `util` package offers a mechanism to compare the Famile mapping elements, which is used for sorting the mappings hierarchically by the mapping handler and a utility class to maintain annotations, e.g., to simplify the Famile feature expressions.

MuVaTra Trace Execution Package For executing the trace-based propagation, on the one hand, the `muvatra::execution::traces` package, offers the interface `TraceConverter` as extension point. A corresponding handler executes the extension which converts a transformation *engine-specific* trace (e.g., an ATL/EMFTVM trace) into the common trace model (c.f. Sec. 5.2.3). On the other hand, the contained transformation package encompasses the facility to execute the trace-based propagation.

The `MVMTraceExecutor` initiates the reused single-variant execution engine, which is either given to the executor programmatically or the `SVTransformationHandler` extracts an installed extension of the `SVTransformer` interface.

The `TracePropagator` performs the actual trace-based propagation based on the common trace model which was converted before. The execution behaves exactly as described in Alg. 1 and employs an instance of the `MappingHandler` to retrieve the annotations from the source mapping and assign them to the target mapping. If no context elements are present, only the annotations of the source elements are combined and mapped onto all recorded target elements. After having iterated the generalized trace, the trace propagator checks whether the some target mapping elements still miss an annotation (e.g., due to an incomplete trace). If this is the case, it will em-

ploy functionality of the `muvatra::execution::missingAnns`¹ package, which is explained in the following paragraph. Finally, the execution package encompasses a `TracePropagationExecutor` which only employs the trace propagator but does not execute the reused single-variant transformation. Thus, it receives the target model as well as the trace which have been created (manually) before, e.g., by executing a transformation manually with the respective tool.

MuVaTra Missing Annotation Package To annotate elements missing an annotation after the trace-based propagation, the missing annotation determination package offers the required functionality. An enumeration summarizes the completion strategies presented in the second part of Sec. 7.3.3. The missing annotation determiner either receives the strategy which it should employ to compute the missing annotations or it asks the developer interactively by initiating a corresponding dialog. For assigning the annotations, the computation also employs the mapping handler which is provided by the propagator.

Graphical User Interface For executing the trace-based propagation, a graphical user interface in form of an Eclipse wizard is part of the framework. It consists of a wizard page and the corresponding control mechanism, the `MVMTExecutionWizard`, which initiates the program logic implemented in the `TracePropagationExecutor`.

Specific Propagation Approaches The projects `ruleAnalysis` and `aspect-oriented` comprise the implementation logic of the corresponding specific propagation approaches. In contrast, to the core packages, they extend the basic functionality with concrete implementation specifics which go beyond the trace-based propagation. On the one hand, the project `ruleAnalysis` relies on executing the trace-based propagation first. Therefore, it utilizes the `TracePropagationExecutor`. While the execution package comprises the execution logic and the propagator for annotations of attributes, the packages `ruleModel` and `ruleAnalysis` offer the logic to analyze EMFTVM-bytecode models. On the other hand, the project `aspect-oriented` comprises the logic to execute an Xpand transformation with the given advice.

9.2.2 MuVaTra Core

The core of the framework is implemented in the `muvatra` package and strongly relies on the *Strategy* design pattern [Gam+97] as well as the plugin mechanism provided by the Eclipse IDE. It consists of packages which offer interfaces for *executing* the multi-variant model transformation, thereby *propagating* annotations and *reusing single-variant transformations* as well as mechanism to maintain the annotations in form of *mappings*. The following paragraphs explain these four components subsequently.

Multi-Variant Model Transformation Engine As depicted in Fig. 9.2.2, the main class of the package `muvatra` is the `MVMTExecutor`. It may contain a `transformer` which is able to execute the reused single-variant transformation realized by the *Strategy* design pattern as well as a propagation mechanism. Except for the execution time, which can be used for conducting performance measurements, this abstract class does not prescribe how to execute the multi-variant model transformation. As multi-variant model transformation approaches vary, particularly the execution of an inter- and a post-processing annotation propagation cannot be generalized further:

Propagation While inter-processing approaches, such as aspect-oriented transformations, propagate annotations *while executing* the reused single-variant transformation, a post-processing approach, such as the trace-based propagation, executes the single-variant transformation firstly and propagates the annotations secondly. Similarly, if the single-variant transformation is executed manually, the `SVTransformer` will not be required for propagating the annotations. A trace, the source and target model with corresponding mapping information suffice to propagate annotations with the trace-based approach.

¹ For space reasons the original name `de.ubt.ai1.mvmt.missingAnnotationDeterminer` is abbreviated.

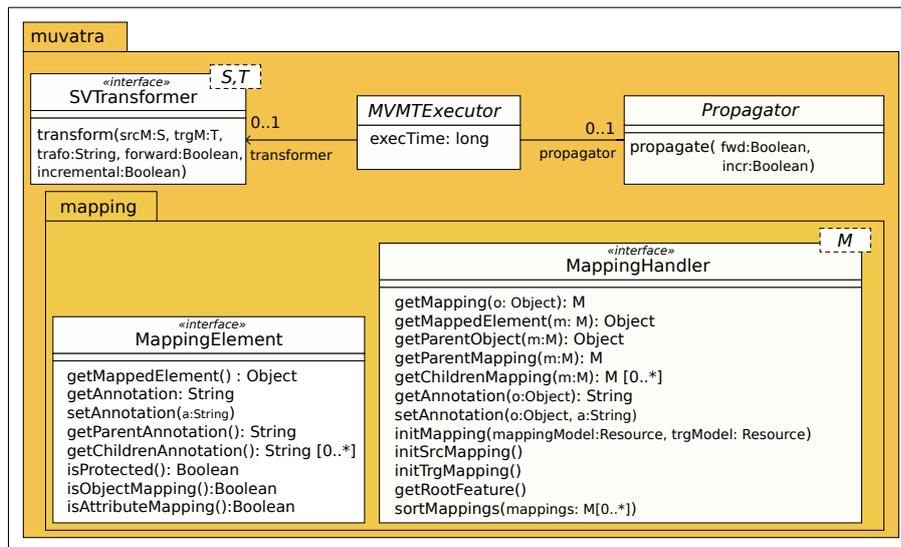


Figure 9.2.2: Core of MuVaTra framework.

Reused Single-Variant Transformation To execute the single-variant transformation, the transformer requires the source and target model representation as input as well as the information to initiate the transformation-specific engine, such as the transformation path or further parameters. The `SVTransformer` maintains the source and target based on generic parameters. Thus, the representation can, for instance, be given as Ecore resources or be the roots of the models which are transformed. The two boolean input parameters of the `transform` method may declare, whether the transformation should be executed in forward or backward direction and in batch or incremental mode. Even though this thesis targets unidirectional transformations, the interface mentions the possibility for a backward transformation which could be implemented by concrete realizations of the interface. Furthermore, the interface `SVTransformer` serves as extension point for Eclipse plugins and, thus, an implementation for the custom transformation language and engine can be implemented as *extension* and plugged into the MuVaTra framework in this way.

Mapping For propagating annotations, the realization of the interface `Propagation` needs to access and modify the mappings of the model elements. Therefore, the framework encompasses an interface for *mapping elements* which realizes the *Adapter* design pattern [Gam+97] and serves to access and modify the annotation mapped onto an element.

The `MappingElement` assumes that an arbitrary element, not only elements of Ecore models, can be annotated. Thus, the method `getMappedElement()` may return an arbitrary Java `Object`. By abstracting from a concrete type onto which the annotation is mapped, the implementation allows for representing annotations for text elements or to realize different types of mappings. The remaining methods are used to compute or map annotations onto the concrete mapped element. Distinguishing the type of mapping, whether it is attached to an object (i.e., a graph node) or one of its attributes allows to perform the specific algorithms accurately. If the mapping element references neither an attribute nor an object, it will be considered a *cross-reference* mapping, which corresponds with mapping an annotation onto a graph edge. A realization of the mapping and the application of the adapter pattern for the tool `Famile` is shed light on next.

9.2.3 Mapping Maintenance With Famile

As this thesis employs the tool `Famile` to implement and evaluate the annotation propagation, the following paragraphs demonstrate how to integrate the tool-specific mapping representation into the framework. In this way, they serve as an example of how to maintain mapping representations of different tools in the framework.

The MuVaTra core project incorporates interfaces for adapting mapping elements and for a handler

respective method on the given mapping element. The constructor of the handler initializes the `FamileMappingElements` and constructs an internal object-to-mapping map. The method `getMapping(...)` employs this map to retrieve the mapping for a given object. The root feature is retrieved from the feature model referenced by the source mapping model. For sorting mappings, the handler employs the interface `Comparator` and the fact that the `FamileMappingElement` offers the customized implementation of the method `compareTo()` which compares the height of the tree levels of the compared mappings. The generic trace propagation exploits the capabilities, offered by the mapping handler, in order to assign annotations to target elements as explained next.

9.3 Realization Specifics

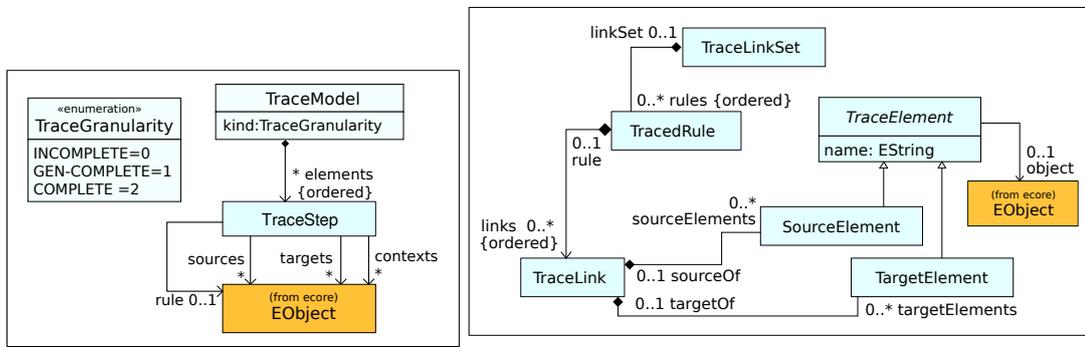
This section presents the implementation specifics of concrete propagation mechanisms. At first, Sec. 9.3.1 illuminates the realization of the trace-based propagation including its extensions when annotations are missing due to a fine-grained trace. In the second place, Sec. 9.3.2 presents details on the rule analysis of EMFTVM rules which builds on the fact that a trace-based propagation was performed before. The section closes with providing details how to realize the aspect-oriented model-to-text multi-variant transformation in Sec. 9.3.3.

9.3.1 Trace-Based Annotation Propagation

One essential contribution of this thesis is the concept of using trace information to propagate annotations. The trace execution package collects the basic implementation. For the trace-based propagation three elements are relevant: the common trace metamodel (instances of which are referred to as *generalized trace model*), the trace converter, which turns an *engine-specific* trace into the generalized trace model and the actual propagation mechanism. The following three subsection illuminate them subsequently.

I Common Trace Metamodel

In the first place, an Ecore model represents the common trace metamodel for traces which was introduced in Sec. 5.2.3. Fig. 9.3.1a depicts the generalized trace metamodel which is used for the trace-based propagation regardless of the granularity of the given trace. The `TraceModel` represents the root which comprises an ordered set of trace steps. Furthermore, it records the trace completeness. The remaining elements correspond with the theoretical descriptions of the common trace metamodel: Accordingly, a trace step records source elements and distinguishes target from context elements also residing in the target model. Model elements are considered to be `EObjects`. In addition, the rule for which the trace step was recorded can be persisted in the trace step and is assumed to be of type `EObject`, too.



(a) Generalized trace metamodel.

(b) Simplified ATL/EMFTVM trace metamodel^a.

^a Based on `org.eclipse.m2m.atl.emftvm.trace` Eclipse project.

Figure 9.3.1: Trace metamodels.

II Trace Conversion

To employ the generalized trace model, firstly, an *engine-specific* trace needs to be converted into the generalized trace model. The following descriptions illustrate the conversion based on converting the ATL/EMFTVM trace, respectively for other traces.

(ATL/EMFTVM) Trace Metamodel As an example, Fig. 9.3.1b depicts the trace meta-model of the ATL/EMFTVM transformation engine (in a simplified form). After executing a transformation, the engine persists an instance of this model. Thus, the root of the trace, the `TraceLinkSet`, contains traced rules which record trace links. A traced rule corresponds with a matched rule of the transformation specification that was executed and is only integrated if there was a matching source object that caused its execution. The traced rule records trace links. A trace link manifests the pairs of source and target elements in the trace link for each application of the rule. Ex. 9.3.1 demonstrates an instance of the ATL/EMFTVM trace metamodel.

Example 9.3.1: ATL/EMFTVM Trace Metamodel and Instance

As an example, Fig. 9.3.2 displays the ATL/EMFTVM trace resulting from transforming the database Ecore model into a UML class model. The main part of the figure shows the trace whereas the right side presents the target UML model into which the input Ecore model is transformed. The bottom left corner depicts an excerpt of the source model.

The ATL TracedRule which converts an Ecore class into a UML class is stored once in the TraceLinkSet. For each converted class which does not inherit from another class, i.e., in total five classes (Person, Media, Family, DataBaseContent and Animal), the traced rule encompasses a trace link, which in turn records the respective Ecore class as source and the UML class as target element, respectively.

The figure highlights the trace link and presents the respective values of the properties view for the Ecore and UML class Person. As source element, called `src`, it records the EClass Person and as target element, called `trg`, the UML class Person.

Trace Converter The trace converter receives the engine-specific trace resource, which is either provided manually or by executing of the single-variant transformation programmatically beforehand. In concrete, the `ATLEMFVMConverter`, which we provided in a respective package, iterates the set of traced rules. At first, it creates a new generalized trace model and sets the granularity to generation-complete. This assumption holds as long as there are no lazy and called ATL rules which would not be recorded in the engine-specific trace.

For each trace link, the converter creates a trace step and sets the source and target elements, accordingly. Since a trace link only records the set of target elements which are created by the corresponding rule and no context elements, there is no possibility to retrieve the information about other already existing target elements which would be used in the rule to create the new elements. After having iterated the traced rules and their links completely, the resulting generalized trace is stored and returned.

Further Trace Converters We have implemented similar extensions of the `TraceConverter` for traces persisted by BXtend and QVT Operational Mappings. The conceptual approach is similar as converting the ATL/EMFTVM trace. One specificity of the QVT-O deviates. Traces stored by the QVT-O engine are *complete*, inasmuch as they record all relevant target elements which are necessary to transform a source element. Thus, besides the created target elements it records context elements but not explicitly. For that reason, the respective `QVTOConverter` maintains a list of already processed target elements and looks currently processed target elements up. If they are already present in the list, they are integrated in the created trace step as context elements. Apart from that, the trace conversion for BXtend traces behaves in a straight-forward way where only one source and one target element are recorded and transformed accordingly.

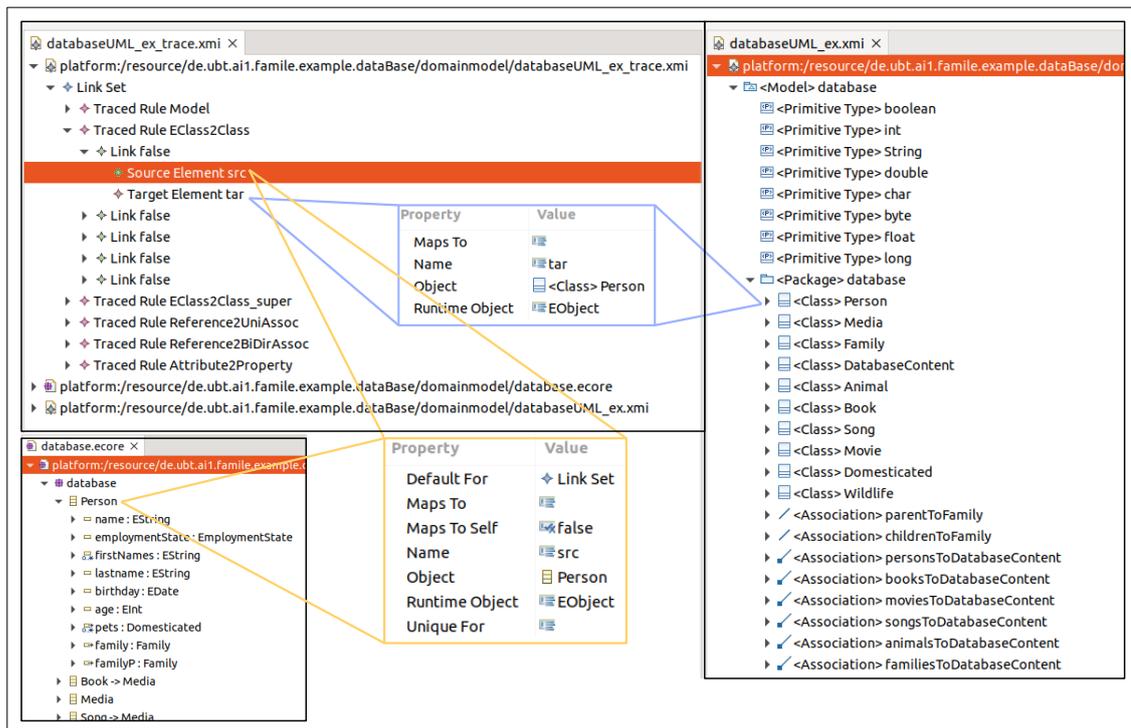


Figure 9.3.2: ATL/EMFTVM trace between Ecore and UML database content model.

III Annotation Propagation

For propagating annotations based on the information stored in a trace, the package `muva tra.execution.traces` provides the generic `TracePropagator`. Independent of the trace granularity, it receives the converted generalized trace model as well as a mapping handler which maintains the source and target mappings. Accordingly, the trace propagator assumes that the same type of mapping element is present for the source and the target domain model. This is a justified assumption for the trace-based propagation because both models reside in the same product line and are built with the same MDPLE environment.

Trace Propagation The propagator iterates the trace steps of the generalized model and assigns annotations to the target elements corresponding to a source element by employing the handler. It combines the annotations of the source elements and – if the trace is complete – of the context elements in a conjunction and maps this annotation onto all target elements. If a (syntactically equal) source annotation occurs multiple times in this expression, it is stated only once. During the iteration, the propagator records the mapping elements which already received an annotation. Depending on the trace completeness, the target mapping model is either completely annotated or some target mapping elements miss an annotation causing an incompletely annotated model.

Example 9.3.2: Trace-Based Annotation Assignment

The example presented in Fig. 9.3.2 shows how an ATL/EMFTVM trace links source and target elements. Accordingly, the generalized trace resulting from its conversion records a trace step that reference the Ecore class `Person` and the UML class `Person` as source and target element, respectively. When the propagation algorithm processes this trace step, the `TracePropagator` employs the mapping handler to look up the annotations of all source elements (i.e., the class `Person`) and context elements (none in this case), combines these annotations in a conjunction and employs the handler to map the computed annotation to the target element. After having processed the trace step, the target F2DMM model maps the annotation `Person` onto the UML class `Person`.

If the trace model records an incomplete trace, and target elements remain without annotations, one of the three completion strategies, described in Sec. 7.3.3, is employed. As sketched in the overview of implementation packages (c.f., Sec. 9.2.1) in Fig. 9.2.1, the completion strategies are represented by an enumeration datatype. The propagator computes a list of open mapping elements which still miss an annotation, sorts the list topologically and creates an instance of the `MissingAnnotationDeterminer` which receives the root feature and the list of open elements.

Annotation Completion The first step of completing annotations separates the mappings of nodes (i.e., `ObjectMappings` in Famile) from mappings for their attributes and those from mappings for cross references. Furthermore, the completion strategy is either provided by the trace propagator (in case it is executed programmatically for the evaluation) or the developer who initiates the propagation can select the strategy via a GUI action. After this initiation, the program iterates the list of open elements and assigns annotations by accessing either the parent mapping, the children mappings or both, depending on the strategy.

IV Trace-Based Execution

Two ways exist to execute the trace-based propagation which depend on the possibility to initiate the single-variant transformation by the means of a program. Either the trace is provided directly or a mechanism has to initiate the single-variant transformation which returns the trace dynamically after executing the transformation:

The `TraceExecutor` which manages the entire trace-based propagation is given the source mapping model and the target mapping model as well as the trace. Based on this information, the executor initiates the `TracePropagator` which performs the annotation propagation as explained above.

In contrast, the `MVMTTraceBasedExecutor` does not receive the trace as artifact but executes the single-variant transformation instead, to retrieve the trace thereafter. In this case, a mechanism to execute the single-variant transformation (i.e., an implementation of the `SVTransformer`) which generates the trace has to be provided and executed .

V Incremental Propagation

For propagating annotations incrementally, the trace propagator further checks whether an annotation is protected. The mapping element interface contains a respective Boolean method. Mapping elements of the tool Famile (i.e., a `FamileMappingElement`), implement this behavior by checking whether the attribute `fixedMapping` is set `true`. Thus, the corresponding function of the `FamileMappingElement` returns the value of the attribute `fixedMapping`.

All annotations which are not protected are overwritten: the trace-based propagation updates the annotations of all non-protected target mappings.

9.3.2 ATL/EMFTVM Rule Analysis-Based Propagation

For computing annotations of attribute mappings in a more fine-grained way than assigning the annotation of the corresponding object mapping, Sec. 7.2 presents a strategy to analyze the ATL/EMFTVM bytecode execution model. We implemented a proof of concept which analyzes the bytecode instruction model resulting from compiling an ATL/EMFTVM transformation specification according to the descriptions in Alg. 2. The implementation reuses the functionalities of the trace-based executor, explained in Sec. 9.3.1, on the one hand, and realizes the analysis of rules as well as a propagator for structural features, on the other hand. Since the analysis is trimmed to the ATL/EMFTVM engine, the trace converter and single-variant transformation executor for ATL are employed in the trace propagator.

Sec. 7.2.4 describes the relevant conceptual information for extracting and using the information to propagate annotations of structural features. Therefore, this section provides further information on the implementation specifics which complement the previous descriptions.

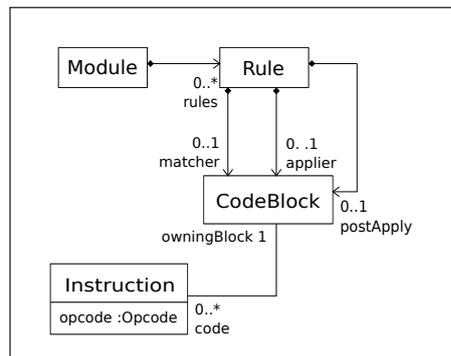


Figure 9.3.3: Simplified EMFTVM execution metamodel of its bytecode instructions.

Road Map The first part scrutinizes background information on ATL and the EMFTVM based on which the second part derives the structure for analyzing the relevant opcodes and patterns from the bytecode model. We employ the resulting rules for propagating the annotations from source to target structural features, described in third part. The section terminates with a reflection on the proposed propagation strategy.

I ATL/EMFTVM

The ATL syntax and execution semantics provide different kinds of rules realizing a hybrid transformation approach of declarative and imperative nature.

Rule Types The execution engine iterates the source model and executes these rules for each matching source element automatically. Accordingly, matched rules must match a source element uniquely. In addition, ATL supports *lazy* and *called* rules. These rules are invoked by another rule and expect implicit and explicit parameters, respectively. While using matched rules only builds a declarative transformation specification which focuses on the correspondences between the source and target metamodel, using called rules only allows to imperatively prescribe the execution order of the transformation specification and, thus, of the creation of target elements. As an imperative construct a matched rule can specify a *do*-block which is executed after the *to*-pattern was executed. In this block no new elements can be created explicitly but helper functions can be used to assign further values or lazy and called rules can be invoked.

Tracing in Rule Types For called and lazy rules ATL does not create traces regardless of the employed virtual machine. Therefore, the annotation propagation contributed in this thesis supports only declarative ATL transformation definitions composed of matched rules. Due to this fact, we can assume that a generation-complete trace results from the execution. If matched and called ATL rules were mixed, the resulting trace would be incomplete.

ATL/EMFTVM Transformation Artifacts Although each ATL transformation maintains a trace internally during the execution, only the EMFTVM compiler persists the ATL trace after the execution. This virtual machine for ATL persists the *trace* as well as an *execution model*, which is an instance of the `emftvm` execution metamodel depicted in a simplified form in Fig. 9.3.3. Consequently, it represents an ATL module which summarizes a set of rules. A rule consists of a `matcher` code block, which represents the *from*-pattern, an `applier` code block which represents the *to*-pattern and a potential `postApply` code block which represents the *do*-block of an ATL rule. The code block enumerates the code instruction which store one of the opcode integrated in the appendix (Fig. A.2.1).

The trace being an instance of the `emftvm` trace metamodel (c.f., Fig. 9.3.1b) is used to exemplify the conversion of an engine-specific trace, the ATL/EMFTVM trace model, into the general trace model in Sec. 9.3.1. In contrast, the ATL execution model reflects the ATL *module* that captures the transformation specification. It composes the rules in terms of different kinds of instructions

which each state the `opcode` which they execute in the order as specified in the rule. An example of the instruction sequence resulting from a transformation block was given in Sec. 7.2.2 in Fig. 7.2.3.

ATL vs. ATL/EMFTVM In addition, when compiling an ATL transformation with the ATL/EMFTVM compiler different behavior is supported for the same syntax. First of all, for employing the compiler the first line of the main transformation specification needs to state the EMFTVM as compiler in a comment: `@atlcompiler emftvm`. Furthermore, some functionality known from the default compiler is not supported, for instance, due to lazy evaluation. It is not possible to add or modify any target element apart from the rule where it is created. For instance, if an additional target element should be added to an existing list of target elements, it will result in a runtime error. Similarly, in our transformations we could not use helper-definitions as variables which are filled in one *do-block* and accessed thereafter in another *do-block*. Thus, their functionality is not implemented in the EMFTVM².

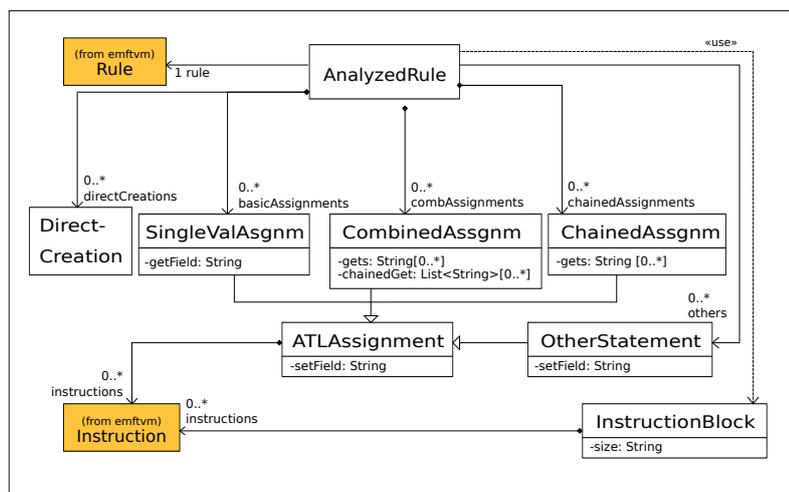


Figure 9.3.4: `AnalyzedRule` as data structure.

II Analyzing Rules

For propagating annotations, the execution model is analyzed as postulated in Sec. 7.2.4. In more detail, the corresponding implementation behaves as follows:

Analysis Procedure The analysis procedure receives an EMFTVM rule from the execution model and analyzes this rule. The result is stored in form of an *analyzed rule* together with the ATL rule as key-value pair, in order to analyze it only once. For a given ATL rule a new instance of the `AnalyzedRule`, depicted in Fig. 9.3.4, is created by performing the following steps:

1. Build a list of `InstructionBlocks` for the ATL *to-pattern* (called `applier`) of the rule. These blocks always end with a `Set` opcode.
2. Build a list of `InstructionBlocks` for the ATL *do-block* (called `postapply`) of the rule. These blocks end with a `SET` opcode or a `POP` opcode. Only include blocks which end with a `SET` opcode because the `POP` instruction branches the execution.
3. Iterate the list of instruction blocks of the *to-pattern* and the *do-block* and analyze the blocks of instructions applying the following categorization:
 - (a) if the first instruction does *not* `LOAD` a value, it is a `DirectCreation`, which pushes a static value or combination thereof only and assigns it to the loaded target structural feature

² Tested with ATL/EMFTVM Version 4.5.0.v202110180912

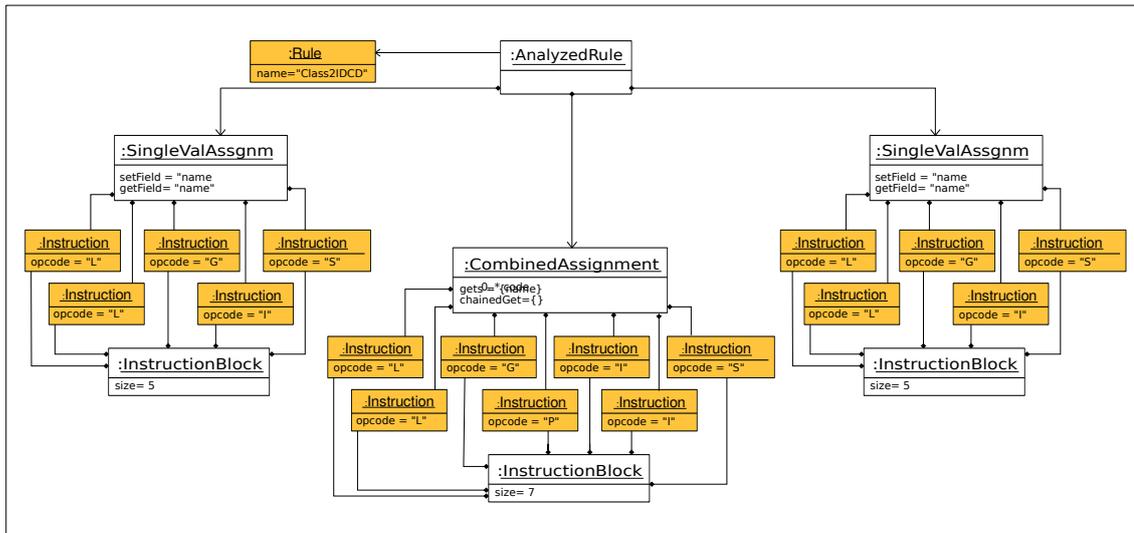


Figure 9.3.5: Instance of `AnalyzedRule` representing ATL rule of Fig. 7.2.3.

- (b) if the instructions start with a LOAD, followed by a get instruction, and only contain 5 instructions in total, it is a `SingleValueAssignment` of a single value
- (c) if two consecutive get operations occur, it is a `ChainedAssignment`
- (d) any block in which a branching instruction, such as an `ITERATE` or `IF` opcode, occurs is classified as `OtherStatement`
- (e) anything else is treated as a `CombinedAssignment` of multiple values to a single value

Pattern Determination The type of recognized assignment pattern determines which information to retrieve from the instruction block and how to handle it in the subsequent propagation: Any instruction block that is categorized as `OtherStatement` or `StaticAssignment` is stored in the analyzed rule for completeness but not considered further. In order to create a `SingleValueAssignment`, the name of the `get`-field is retrieved from the corresponding instruction block. To create a `ChainedAssignment`, the list of instructions is iterated from the first `GET` to the last `GET` in the sequence. It must be noted that the implementation recognizes only one sequences of gets. If there are multiple “chains”, they will be recognized as `CombinedAssignment`. To create a `CombinedAssignment`, the list of instructions is iterated and the `GET` instructions are only stored in the list of “single” `gets`, if no other `get`-field follows immediately thereafter (i.e., if there is no chain). If a chain is recognized, it will be added to a separate list.

The entire pattern recognition is implemented in a functional implementation style. Statements are extracted while creating the analyzed rule and their fields are also set in the constructor. Thereafter, these elements cannot be modified anymore.

Example 9.3.3: Analyzed Assignments in ATL Rule

The ATL rule and corresponding bytecode presented in Fig. 7.2.3 are summarized in the analyzed rule as shown in Fig. 9.3.5. The analysis recognizes two `SingleValueAssignment`s as well as a `CombinedAssignment`. The constructor of the analyzed rule builds the instruction blocks which serve as intermediate data whereas the rule stores the identified `ATLAssignment` patterns permanently. Furthermore, the assignments store the concrete instructions in the order present in the bytecode model.

For example, five instructions represent the first assignment `'name <- src.name'` which load (L) the source and target object, get (G) the value of the source `name`, invoke (I) the resolution of the target object and set (S) the value of the source `name` to the target `name`.

III Propagation of Structural Feature Annotations

After having analyzed the rules, the `AttributePropagator` iterates the trace model. For each source element it looks up the ATL rule which matches the source object. Based on the ATL rule, it determines the corresponding `AnalyzedRule`. If the rule was not analyzed before, it will be analyzed by creating a new instance of the `AnalyzedRule`. The constructor performs the pattern recognition such that the patterns can be used in the following propagation.

Then, the propagator iterates the target elements of the trace steps and looks up the mappings of `EObjects`. For each target object mapping, the list of attribute and crossreference mappings are iterated. The `RuleAnalyzer` checks whether their names matches the *set*-field in the identified patterns of the analyzed rule. If this is the case, it will check the structural features of the corresponding source object to find those which match the corresponding *get*-field in the pattern. The annotations of the matching source structural features are combined in a conjunction and mapped onto the target structural feature which matched the *set*-field.

IV Reflection

For summing bytecode instruction analysis up, this part reflects on the computational model, the benefits and the results of examining the implemented prototype.

Computational Model Propagating fine-grained annotations based on our approach requires to satisfy Prop. 7.2.1- Prop. 7.2.3 which postulate the assignment of a single value (i.e., no collections are created and assigned), a linear execution flow and particularly a self-contained transformation. This paragraphs illuminates to what extent the capabilities of the ATL/EMFTVM meet these properties. The language ATL is more powerful than the required properties. It allows to assign collections to multi-valued structural features and offers several syntactic constructs that may diverge the control flow such as if-conditions and for-loops. Furthermore, due to the module mechanism, ATL transformations are more powerful and not only allow to invoke helper methods in the same file but also of different referenced other files.

Consequently, the implementation cannot guarantee commutativity in general but it depends on the syntactic constructs used in the ATL transformation specification. For instance, if specific annotation are mapped onto structural features only which are mapped in a `SingleValueAssignment` commutativity is satisfied nonetheless. Furthermore, while the computational model is less restrictive, ATL only allows to match a source object once per rule. Thus, in the graph formalism, each source graph node of the STT graphs possesses exactly one incoming source edge.

Limitations and Benefits First of all, not the entire set of instructions can be recognized in the current state of the analysis. We focus on assignment patterns which occur often and can be analyzed without employing heuristics. For branching conditions, the recognition is excluded because the elements being iterated or guarding the execution branch may provoke the creation or deletion of elements in the single-variant execution which differ from the result of the multi-variant execution. Since this behavior threatens commutativity of the transformation and may require an unconstrained variability representation for the target model, it is out of the scope of this thesis (c.f., Prop. 4.1.2).

Moreover, as discussed before, the additional analysis of rules and corresponding iteration of the source and target model elements adds additional computational cost. Thus, for employing it (instead of manually refining single target annotations), there should be a relevant number of structural features carrying an annotation which is more fine-grained than the annotation of their object on which they depend.

Execution Results To demonstrate the feasibility and potentials of the approach, we performed the rule analysis in two scenarios. We used the motivating example of the database product line where the number of first names is only higher than one if the feature `MultipleNames` is selected as scenario, on the one hand. The propagation based on the rule analysis extension annotates the target attribute correctly.

On the other hand, in a scenario of transforming a Graph product line [LB01] (c.f., Sec. 9.3.2, II) Ecore domain model into a corresponding UML class model, the propagation performs successfully, too. An adaptation of this product line [GSW17] allows for weighted edges represented as an **EAttribute** of the class **Weight**. The **EAttribute** is present in each Graph regardless whether the optional feature **Weighted** is selected or not. While a concrete value can be provided when the feature **Weighted** is selected, a value of one is automatically derived when the respective feature is not selected. This product line assumes this behavior by mapping the annotation **not Weighted** onto the meta-attribute of the **EAttribute** **weight**.

In both the database and the graph product line, the annotations of the structural features are propagated correctly. However, it must be noted that **SingleValueAssignments** assign the values of the **derived** and the **name** attribute. Although the examination does not cover the chained or combined assignments, we refrain from evaluating the implementation mainly due to the following reason: Such propagation pays off if a high number of structural features is annotated more specifically than its objects (c.f., Sec. 7.2.6) which is not the case in the evaluated product lines (c.f., Sec. 10.2.2).

9.3.3 Model-To-Text Aspect-Oriented Propagation

To proof the concept of propagating annotations in the aspect-oriented way, proposed in Chp. 8, this section describes the realization of the *generic advice* in the M2T language Xpand. The *extension*-file used in the advice allows to propagate the annotations of the mapping model used in the tool Famile (c.f., Sec. 9.1.2). Furthermore, we employ a customizable preprocessor which can be adapted to several programming languages without built-in preprocessor tool support by allowing to configure the syntax of the directives as well as the escape characters which integrate the preprocessor directives in a non-invasive way in Java source code. Details on the preprocessor resulting from a student project are given in Sec. A.3.

The solution to propagate annotations while executing the reused single-variant transformation consists of two key components: A template containing the *generic advice* to modify the existing SVMT and a *MWE(2) workflow* which augments the execution of the original M2T transformation with the implemented advice.

Road Map The first two parts of this section explain how the advice has to be specified for propagating annotations generically and how the (MWE2) workflow weaves the advice into the original text production. In contrast, the third part of this section offers details on the realization of the aspect-oriented propagation which we implemented as proof of concept. To this end, the fourth paragraphs discusses the implementation with respect to the computational model and provides an outlook on the evaluation which follows in the next chapter.

I Advice Template

In the first place, the automatic transfer of annotations requires a generic instruction to annotate source code fragments which correspond with an annotated source model element. Therefore, we implement an Xpand advice which behaves as follows: First, the originally generated text should be surrounded with an annotation. Thus, an *around*-statement forms the key element of the advice which invokes the original text production. In order to regard any model element text is generated for, the *around*-statement needs to state a *wildcard* such that each object matches. The latter is required to retrieve the annotation which is mapped onto it. If an annotation is present, the *around*-statement will generate an opening and a closing directive which enclose the original text production. The following paragraphs present details, how this behavior is realized.

Generic Advice Listing 9.3.1 describes a simplification of the generic advice which aims at creating annotated (Java) source code:

The first line comprises the **AROUND** statement. By using a wildcard for the objects that should match the advice and stating `templates:*:*` any *define*-block in any regular Xpand-file (not

```

1 «AROUND templates::*:* FOR Object»
2 «LET getMapping((EObject)this) AS mapping»
3 «IF mapping != null && mapping.isSetFeatureExprStr()»
4 «getOpeningComment(mapping)»
5 // @ID: «getID((EObject)this)»
6 «targetDef.proceed()»
7 «getClosingComment()»
8 «ENDIF»
9 «ENDLET»
10 «ENDAROUND»

```

Listing 9.3.1: Simplified generic advice which embraces an original text generation.

```

1 /** #IFDEF !Family && Person # */
2 // @ID: _TACGEFc3Eetfr4BhVYAxQ
3 private String surname;
4 /** #ENDIF */

```

Listing 9.3.2: Java field declaration annotated with JavaDoc comments.

in extensions specified in Xtend) stored in the folder `templates` can be the target of the advice whenever the source type of the *define*-block is of type `EObject`³.

In the next step, the advice retrieves the mapping element for the input object in Line 3. The advice invokes a method `getMapping()` which is implemented in an Xtend extension and accesses the F2DMM object mapping for the given `EObject`. To accomplish this task, the *extension*-file employs a Java class which loads the F2DMM model of the source domain model automatically by searching the Famile project for files with an `f2dmm` file ending. If the product line comprises multiple F2DMM models, it will ask the developer to choose the appropriate one. After the first look up of a mapping element, the F2DMM model is stored for following look ups. If the mapping element is annotated, the annotation will serve to enclose the original text production with a preprocessor directive.

Please note: Although we assume that the input model is annotated completely, Famile does not guarantee this assumption. Therefore, the advice checks whether an annotation is mapped onto the matched source object in Line 3. Additionally, the method `getOpeningComment()`, which is implemented in Java, checks whether an annotation is mapped onto the parent and assigns either the annotation of the matching object solely or its conjunction with the parent annotation.

In addition, the advice computes a unique ID in Line 5 to identify the object in potential consecutive executions. This ID is stated as a comment before the original text production and can be used to identify the text production and corresponding source object in an incremental transformation uniquely. Furthermore, it could also be used to reconstruct trace information.

Resulting Annotation Listing 9.3.2 illustrates the preprocessor directives which result from applying the advice to transform the attribute `surname` of the class `Person` in the example depicted in Fig. 8.1.1. The first line opens the directive and states the annotation. It uses the respective preprocessor directive, in this case `#IFDEF`. As Java source code is generated which does not support preprocessing, the directives are stated as Java comments so that the source code is still compilable. Due to the preprocessor directive, the field is only present in a derived product if the feature `Family` is not selected but the feature `Person`. The last line shows the closing directive as corresponding with the preprocessor. The statement `'targetDef.proceed()'` executes the original text production which creates the text stated between the ID directive and the closing directive. Due to the comments which state the annotation of the source object for which text fragments were created, it is possible to inject the variability information in a text production. By combining the annotation of the source element with its parents annotation, it is ensured that a container

³ Our implementation expects F2DMM models which map annotations onto `EObjects` (and their structural features) only.

```

1 module de.ubt.ail.ecore2Text.Codegen
2
3 var targetPath = "src-gen"           // provide parameter for exchanging output path
4 var modelPath = "model/database.ecore" // provide parameter for exchanging input model
5 var advices = "templates/Main"      // provide parameter for injecting an advice from outside
6
7 Workflow {
8   bean = org.eclipse.emf.mwe.utils.StandaloneSetup {
9     registerGeneratedEPackage = "org.eclipse.emf.ecore.EcorePackage"
10  }
11
12  component = org.eclipse.emf.mwe.utils.Reader {
13    uri = "${modelPath}"
14    modelSlot = "model"
15  }
16
17  component = org.eclipse.xpand2.Generator {
18    id = "generator"
19    metaModel = org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel{ }
20    expand = "templates::Main::main FOR model" // entry point rule
21    fileEncoding = "UTF-8"
22    outlet = {
23      path = "${targetPath}"
24      postprocessor = org.eclipse.xpand2.output.JavaBeautifier{}
25    }
26    advice = advices // provide the slot for injecting an advice
27  }
28 }

```

Listing 9.3.3: Reused single-variant MWE2 workflow for an Ecore to Java transformation.

is not removed by the preprocessor. Depending on the preprocessor and target language, the methods `getOpeningComment()` and `getClosingComment()` of Listing 9.3.1 have to be adjusted.

II Workflow

MWE2 workflows consist of *variables*, *components* and *parameters* which enrich the components. Furthermore, *beans* declare the setup of the execution. To execute the advice, as second part of the realization, an MWE2 workflow has to specify the advice in its component **generator**.

Transformation Project Setup In general, the realization consists of two projects. One project contains the single-variant reused Xpand specification including all relevant *extension-files* and the MWE2 workflow, for instance, for a transformation of Ecore models into Java source code. The MWE2 workflow, which declares the input metamodel and the paths of the input model and the output directory, executes the transformation. The second project comprises the advice template and another MWE2 workflow. This first workflow weaves the generic advice template which allows to modify the input and output directories into the single-variant execution.

Single-Variant Transformation Workflow Listing 9.3.3 presents the workflow which executes the original text production and weaves the advice implementation presented in Listing 9.3.1 into the original production. After the `module` declaration in the first line, *variables* define the target path where the output text is written to, the *input* model path and the location of the advice. If the workflow is initiated from a Java implementation, these variables serve as parameters, the values of which can be provided by the implementation. The input model is given to the *reader* component which parses the input model file while the second component of the workflow defines the *generator* which creates the text production. In this example, the workflow creates Java source code for a given Ecore model and stores the generated source code in the target path. The parameter `expand` of the generator states the entry point rule of the single-variant M2T transformation. The parameter `advice` serves as placeholder which can be replaced by the second MWE2 workflow which executes the aspect-oriented transformation.

```

1 module de.ubt.ai1.mvmt.AdviceGenerator
2
3 var targetPath = "./mv_src-gen" // provide a custom output path
4 var adv = "./advices/advice" // specify the location of the advice
5 var modelPath = '../de.ubt.ai1.famile.example.dataBase/domainmodel/database.ecore'
6
7 Workflow {
8   bean = org.eclipse.emf.mwe.utils.StandaloneSetup {
9     scanClassPath = true
10    registerGeneratedEPackage = "org.eclipse.emf.ecore.EcorePackage"
11    registerGeneratedEPackage = "de.ubt.ai1.f2dmm.F2DMMPackage"
12    registerGeneratedEPackage = "de.ubt.ai1.f2dmm.fel.FELPackage"
13    registerGeneratedEPackage = "de.ubt.ai1.f2dmm.sdirl.SdirlPackage"
14    registerGeneratedEPackage = "de.ubt.ai1.fm.FeaturemodelPackage"
15
16    extensionMap = { // add 'f2dmm' as file extension
17      from = "f2dmm"
18      to = "de.ubt.ai1.f2dmm.util.F2DMMResourceFactoryImpl"
19    }
20  }
21  component = @de.ubt.ai1.ecore2java.Codegen { // the original text production
22    targetPath = "${targetPath}" // update target path
23    advices = adv // provide the advice
24    modelPath = "${modelPath}"
25  }
26 }

```

Listing 9.3.4: Advice-injecting workflow for Ecore to Java transformation.

Aspect-Weaving Workflow The aspect-oriented workflow executes the single-variant text production by employing its generator and providing values to its variables. Listing 9.3.4 presents the workflow which executes the single-variant Ecore to Java text production of Listing 9.3.3.

Firstly, Listing 9.3.4 may define another target path where the multi-variant source code is stored. Furthermore, it states where the actual generic advice is located. Most importantly, the standalone setup loads all registered packages which are necessary to load the mapping representation (here the F2DMM model of the tool Famile). The extension map is used to identify the ending of the mapping model and associate it with the corresponding factory.

Then, this advice-injecting workflow executes the original text-producing workflow. The workflow overwrites the target path of the `ecore2java.Codegen` module in Line 22 and injects the advice in the following line. Lastly, it defines the location of the input model, which replaces the input model of the original text production. The remaining functionality, which is necessary to execute the text production, is used in the way it is defined in the workflow of the original text production.

III Incremental Realization

For preserving manually added text in the generated text, Xpand offers the functionality of protected areas. As explained in Sec. 8.2.1, Xpand requires to mark a protected block with a unique ID. An incremental transformation looks up the contents of the protected block together with the ID and adds the contents to the block with the same ID created in the incremental transformation. Anything that is not protected will be computed anew.

Example In the exemplary Ecore to Java M2T transformation, we add a protected block to the method body. This allows to integrated behavior at the level of domain engineering and obviates the need to provide a manual implementation for each block.

Listing 9.3.5 presents the Xpand *define*-block which transforms an Ecore operation and preserves the method body from being overwritten. It differentiates the situation when a return type is given and not. In both cases the method body is protected in the default Xpand way. A *protected*-block initiates with the keyword `PROTECT` followed by defining the comment character sequence which needs to allow for differentiating normally generated text from a protection-comment. The keywords `CSTART` and `CEND` define the character sequence of such comment in Lines 4 and 9, in

```

«DEFINE EOperation FOR ecore::EOperation»
2 «IF eType != null »
3 public «getType()» «name»(«FOREACH eParameters AS par SEPARATOR ',' » «par.getType()» «
   par.name» «ENDFOREACH») {
4   «PROTECT CSTART "/*" CEND "" ID getID((EObject)this)»
5   return «returnValue()»;
6   «ENDPROTECT»
7 «ELSE »
8 public void «name»(«FOREACH eParameters AS par SEPARATOR ',' » «par.getType()» «
   par.name» «ENDFOREACH») {
9   «PROTECT CSTART "/*" CEND "" ID getID((EObject)this)»
10  «ENDPROTECT»
11 «ENDIF»
12 }
«ENDDDEFINE»

```

Listing 9.3.5: Protected text production of generating a method declaration for an EOperation.

this example the characters `/*`. Thereafter, the `ID` keyword request the object ID which serves to uniquely identify the source object for which text is generated (i.e., the operation in this example). For that reason, the method `getID()` expects an `EObject` and is implemented as a Java extension which determines a unique and stable ID for the object.

Inside the protection block, the text-generating rule may add text-producing instructions. This is the case in Line 5. In this example, the return type of the method is not empty meaning a value has to be returned by the method in order to compile the Java class without errors. The method `returnValue()` determines the default return value for the type of the given operation. The developer can change these text-productions manually which will not be overwritten by the following text production. The keyword `ENDPROTECT` closes the protection block.

Variability Inside Protected Blocks It is important to notice, that also inside method bodies conditional implementations may occur. For instance, if the database should print all its contents, not every feature may be selected and accordingly several references are only valid if the corresponding feature(s) are selected. As a consequence, when manually adding source code to a protected block, the developer needs to enclose text corresponding with optional features with respective preprocessor comments in the correct style.

Execution For executing incremental transformations which preserve manually added source code in protected areas, the generator component of the MWE2 workflow needs to be modified slightly, as presented in Listing 9.3.6: The workflow does not deviate much from the batch MWE2 workflow. For protecting text in protected blocks it suffices to add the parameter `prSrcPaths` to the generator component as in Line 15. As a consequence, if executed again the workflow scans the existing protected paths. If their ID matches the ID of the currently processed object and ID of the currently processed protected region, the previously stored text will be added to the matching section. Thus, it remains present in a subsequent execution.

Incremental MWE2 workflows are also capable to compare an old with a new version of a model and to store a trace. However, for protecting regions with manually added text it suffices to employ the ID-based mechanism.

IV Discussion

Based on the description of how to implement an aspect-oriented annotation propagation in Xpand, this section discusses the results with respect to achieving commutativity. At first, the section examines the properties of Xpand with respect to the computational model followed by giving an outlook on the evaluation.

```

1 module de.ubt.ai1.ecore2Text.incr.Codegen
2 ...
3 Workflow {
4   bean = ...
5   // incremental text generation:
6   component = org.eclipse.xpand2.Generator {
7     id = "generator"
8     metaModel = org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel{}
9     expand = "templates::Main::main FOR model"
10    fileEncoding = "UTF-8"
11    outlet = {
12      path = '${targetPath}'
13      postprocessor = org.eclipse.xpand2.output.JavaBeautifier{}
14    }
15    prSrcPaths = '${targetPath}' // consider protected paths in target model
16    advice = advices
17  }
18 }

```

Listing 9.3.6: MWE2 workflow for incremental Ecore to Java transformation.

Properties of Xpand Model-to-Text Transformations In the first place, it must be noted that Xpand does not satisfy the computational model for aspect-oriented transformations completely with the following respects:

Transformation In Xpand, a *define*-block represents a text-producing rule. As such, only a single source object is transformed into text and either matches the entry point rule or an invoked rule. A rule is left preserving (Prop. 8.3.2) because the source model cannot be changed, however, it can be invoked on itself (i.e., with the same input object). Thus, the same rule can be executed on the same object multiple times which violates Prop. 8.3.1. Due to this capability, a *finite number of matches and a finite depth of the execution tree* (i.e., violating Prop. 8.3.4) cannot be guaranteed for a transformation written in Xpand generally. However, a transformation specification may satisfy the computational model, nonetheless, if it does not employ recursive rules on the same node. Thus, commutativity depends on the rules specified in a concrete transformation.

Non-Local Even though template definitions cannot be guarded with conditions, which hinder locality, inside of expand declarations *if*-conditions may be implemented which vary the behavior with respect to different properties of the matching source object. If upon filtering a condition is satisfied but was not for the multi-variant source model or vice versa, the created text fragment will be missing in the multi-variant source code but will be present in the transformed single-variant source code.

In a similar way, an advice may change the implementation of a join point almost arbitrarily. An advice possesses the power to remove the original text production completely and to replace it with arbitrary text. As a consequence, the result of executing a single-variant transformation without injected aspects and the multi-variant version with injected aspects may deviate which threatens commutativity. Accordingly, injecting an advice may violate *left-preservation* (Prop. 8.3.2).

Non-Monotonic The Xpand syntax (without aspects) is monotonic. It does not offer language constructs to delete elements. However, Xpand may violate the *monotonic* behavior of the transformation (Prop. 8.3.9). As Xpand allows to employ extensions written in the GPL Xtend and in Java, those extensions possess the power to modify the source model and the created target files.

Functional and Order-Preserving Xpand transformations are *functional* (Prop. 8.3.5) and preserve the execution and text generation order (Prop. 8.3.8). Even if the execution of a rule triggers an advice, for the same input node, the same rule generates the same text in the same order.

Evaluation Outlook Despite the fact that specifications written in Xpand may violate the computational model, evaluation results have achieved 100% correctness in terms of commutativity for the Graph product line [GW18a]. On the one hand, annotations are only provided to source objects not to their attributes and, thus, satisfy the computational model. On the other hand, for each annotated source node a *define*-block is used in the original transformation. Even though a source object can match several define-blocks and accordingly be processed multiple times, these blocks do not execute any other block and are implemented non-recursively. Therefore, the transformation terminates and the annotations can be determined uniquely. Sec. 10.3.5 offers a detailed examination of the already partially introduced Ecore to Java transformation and a MoDisco Java model to Java source code generation.

9.4 Summary

On the whole, this chapter explains how we realized the different annotation propagation strategies which were introduced theoretically in the preceding chapters. Sec. 9.2 presents an overview of the MuVaTra framework which allows not only to realize the propagation based on traces of any completeness level but also the aspect-oriented propagation and the bytecode analysis. Although we employed the MDPLE tool Famile to realize the product lines and consequently maintain its external annotation representation in form of F2DMM model, the framework offers several possibilities, such as extension points and respective design patterns, to support different annotation mapping mechanisms and further transformation languages. The subsequent section, Sec. 9.3, explains how we realized the trace-based propagation, bytecode instruction analysis and aspect-oriented propagation in detail. In this way it serves as example of how to replicate or integrate the capabilities to server other MDPLE tools. The following chapter evaluates to what extent the approaches meet the research objectives and employs the implementations presented in this chapter to propagate the annotations of different subject product lines.

Chapter 10 Evaluation

*Science, for me, gives a partial explanation for life.
In so far as it goes, it is based on fact, experience and experiment.*

Rosalind Franklin

————— ~ —————

This chapter evaluates whether and to what extent the proposed concepts and their implementations satisfy the research objectives.

Accordingly, the concepts presented in Chp. 5-7 are examined with regards to being *automated*, *generic*, and *correct*, and *reuse* existing technology. The evaluation considers the approaches which we implemented as proof of concepts and which have not been rendered as impractical before. Therefore, in the sequel, we examine the trace-based propagation realization which regards complete and generation-complete traces as well as the extension for incomplete traces. Furthermore, the analysis of EMFTVM rules is discussed briefly and the aspect-oriented propagation in M2T transformations is discussed extensively.

The results confirm that the main purpose of reducing the manual efforts can be achieved with each of the proposed strategies. Even with incomplete traces more than 90% of the assigned annotations are computed correctly which also holds for the aspect-oriented approach where non-local rules are employed. Surprisingly, in the examined scenarios, a generation-complete trace suffices to satisfy commutativity to full extent.

The sequel starts with explaining the goal of the evaluation in Sec. 10.1 and continues with describing the setup of the evaluation, including the implementation, the subject product lines, and the transformation definitions in Sec. 10.2. While the following section, Sec. 10.3 disseminates the results of the evaluation, Sec. 10.4, summarizes and discusses them.

[Gre19], [GW19b], [GW19a] and [GW18a] lay the foundations of this chapter.

10.1 Evaluation Goal

The evaluation checks whether and to what extent the research objectives are achieved in transformation scenarios which deviate from the computational models. To recapitulate, the research objective of this thesis is to develop methods which *reuse* (**RO1**) existing single-variant model transformations in order to *automatically* (**RO2**) propagate annotations. These methods should propagate annotations *generically* (**RO3**) and *correctly* (**RO4**).

The conceptual idea of trace-based propagation (c.f., Chp. 5), its extensions (c.f., Chp. 7) and of *aspect-oriented propagation* (c.f., Chp. 8) demonstrate that the methodologies are automated and do not modify the reused single-variant transformations technology but only exploit existing capabilities. Thus, we achieve the reuse and automation objectives, **RO1** and **RO2**, by the design of the methods. Conversely, the correct and generic propagation (**RO3** and **RO4**) depends on the respective approach and genericity can partly be achieved with the implementation as the following two sections discuss.

10.1.1 Genericity of Propagation

As first criterion, this section discusses to what extent the proposed approaches are generic. The feature-based classification of multi-variant transformations, introduced in Sec. 4.3.2, classifies the *scope* of these transformations with respect to the transformation *definition* (i.e., the transformation definition) and the *language*: An entirely generic solution propagates annotations independently of the definition (i.e., independent of the metamodels over which it is defined) and of the *language*. Therefore, the following paragraphs categorize to which extent the trace-based propagation, the bytecode analysis and the aspect-oriented propagation are generic with respect to these two criteria.

Trace-Based Propagation Firstly, the *trace-based* propagation is an entirely generic approach: It does not rely on engine-specific traces but abstracts from the potentially distinct information contained in traces written by different transformation execution environments. As a consequence, the approach can be applied **independently** of a specific transformation **language**, such as ATL or QVT-O, whenever information about corresponding source and target elements is available. Additionally, since the only relevant information about the source objects recorded in the trace is their annotation, no specific information about the metamodel they are instances of is required. As the same holds for the target model, the approach works in a **definition-independent** way, too. In sum, the trace-based approach, as well as the proposed completion strategies, which only assume a tree-structured input model, are entirely generic.

Bytecode Instruction Analysis Secondly, the analysis of the ATL/EMFTVM bytecode model is a **language-specific** but **definition-independent** approach. While we identify value assignment patterns which may be used similarly in different transformation languages, we map them onto the EMFTVM bytecode instructions only. Thus, the approach can be applied exactly to ATL/EMFTVM transformation definitions and may have to be adapted to support other transformation languages and their compilers. Nonetheless, the approach works independent of a specific metamodel: The relevant information that is extracted by the analysis are the structural features of the additionally extracted source object (of whatever type) and the structural feature of the target object which is assigned a new value based on the source structural feature(s).

Aspect-Oriented Propagation Thirdly, the aspect-oriented approach is **language-independent** and **definition-independent**. While we specified the aspect in the language Xpand and use the language in the evaluation, the aspect can be defined in another aspect-oriented language in the same way. Only the directives have to state the annotation according to the used preprocessor. Furthermore, the kind of source metamodel and the type of target text which is generated do not influence the mapping of the annotations onto the generated text.

Consequences In summary, the trace-based and aspect-oriented approaches work completely generically (independent of the language and the definition) whereas the bytecode instructions analysis is trimmed to a specific language.

10.1.2 Correctness of Propagation

As correctness criterion, commutativity must be satisfied. The transformed single-variant target model and the derived single-variant target model must be equal up to isomorphism when being abstracted as graphs.

Commuting Annotations The correctness of trace-based propagation involving transformation rules and traces which satisfy the computational model (c.f., Sec. 5.3.3) is formally proven whereas its practical extensions dealing with incomplete trace information offer a high degree of flexibility which mitigates proofs of commutativity in every situation. The extensions, which target situations where no *complete* trace information is available, can satisfy commutativity, nevertheless, depending on the transformation definition and the source and target models. Similarly, M2T transformations specified in Xpand may satisfy commutativity despite the fact that Xpand does not satisfy the computational model for aspect-oriented propagation (Sec. 8.3.2) in its entirety.

First Evaluation Question: Correctness As a consequence, the relevant questions for evaluating the correctness are as follows:

EQ1 Does the propagation approach commute?

- a To what extent is commutativity achieved? (What is the size of the *absolute* and the *severity* error?)
- b How much manual effort has to be invested to repair wrong annotations?
- c Which completion strategy achieves the highest accuracy?

To measure, and thereby answer, the first main question, whether the propagation approach commutes, we employ an evaluation framework (c.f., Sec. 10.2.1) which executes the commutativity criterion (rigorously). As a result from comparing the filter-transform and transform-filter variants in this framework, the quantitative error regarding the number of differences between each pair of derived variants is measured and used to answer **EQ1 a**. Consulting the evaluation data in more detail, allows to detect mismatches in the compared variants and to fix corresponding target annotations based on that information and context knowledge. Counting this number allows to answer **EQ1 b** and to compute the actual error measure of annotations with a wrong effect. Finally, **EQ1 c** compares the computed error measures in each completion strategy (c.f., 7.3.3) to determine which strategy performs best in terms of accuracy.

To validate the degree of correctness, the evaluation employs three error measures, an *absolute* error, a *severity* error and the *actual* error:

Absolute Error The absolute error counts the number of feature configurations which violate commutativity and weighs them against the number of all valid feature configurations. At this coarse-grained level, it states how many percent of the feature configurations are erroneous.

Definition 10.1.1: Absolute Error

Let n be the number of all valid feature configurations for a given feature model and let w be the number of feature configurations in which the filter-transform and transform-filter models deviate. The absolute error err_{abs} constitutes as follows:

$$err_{abs} = \frac{w}{n}$$

As a consequence, this measure may render a configuration as erroneous as soon as *one* out of hundreds elements of one model differs from a counterpart in the model with which it is compared. In the worst case, this may provoke an error of 100% even though only one annotations of the multi-variant target model is not specific enough.

Severity Error For that reason, we have relaxed the error measure, to count the number of detected differences of each feature configuration and weigh it against the number of all target annotations in the multi-variant target model:

Definition 10.1.2: Severity Error

Let n be the number of all valid feature configurations for a given feature model. Let $|diff|$ be the number of differences between two models and $|m_t|$ be the number of all target elements in the multi-variant model.

The severity error err_{sev} constitutes as follows:

$$err_{sev} = \frac{\sum_{i=1}^n \frac{|diff_i|}{|m_t|}}{n}$$

According to the definition, the severity error requires to determine the number of differences between the model derived from the source model and transformed into the target model and the model derived from the multi-variant target model. As the number of model elements in the two compared models may deviate, we employ the number of all model elements in the superimposed model as base line. Furthermore, it must be noted that the number of differences may depend on the capabilities of the comparison tool.

Actual Error Since the number of differences depends on the accuracy of the comparison approach (e.g., the EMFCompare comparison mechanism), a third measure regards the actual number of annotations that have to be corrected in the multi-variant target model to achieve an absolute error of 0%. This number is weighed with the number of all possible mapping elements in form of the *actual error*.

Definition 10.1.3: Actual Error

Let $\#map_t$ be the number of all elements that can be annotated in the multi-variant target model. Further, let a_{F_c} be the number of annotations that have to be corrected to achieve commutativity.

The actual error err_{act} constitutes as follows:

$$err_{act} = \frac{a_{F_c}}{\#map_t}$$

Compared to the absolute and severity error, the actual error does not measure the *effect* of an annotation on the model elements but counts the actual number of annotations in the multi-variant target model that threaten commutativity.

Remarks Three points have to be considered when computing the actual error: Firstly, the error measure is only defined if it is possible to satisfy commutativity ($err_{abs} = 0\%$) with the transformation at all. In case a constellation of annotations in the source model provokes different behavior of the single-variant transformation when applied to the multi-variant and a single-variant transformation, it is possible that commutativity cannot be satisfied by the means of any target annotation. Then, the actual error cannot be computed.

Secondly, the number of all mapping elements map_t is higher than the number of annotations assigned to objects only: map_t considers each element that can be annotated by the means of the employed mapping mechanism. Accordingly, for instance, annotations which are mapped onto

crossreferences or structural features of an object are respected in the value, if those elements can be annotated. As a further consequence, this number may not correspond with the number of annotations which are mapped onto target elements by the propagation mechanism automatically. However, the value is beneficial because it remains stable for the same target model regardless of the propagation approach.

Thirdly, two annotations may be (syntactically) different but may be satisfied by the same set of configurations. As a consequence, it is possible to change annotations in multiple ways to achieve the same goal of commutativity. Thus, for measuring the actual error, on the one hand, the developer has to adjust annotations manually and count how many annotations have to be modified in order to compute the value. On the other hand, due to ambiguities, we assume, that the developer modifies only a minimal set to satisfy commutativity and that the developers determine meaningful annotations.

10.1.3 Propagation Benefit

Even though we have already stated that the strategies are automated, the benefits of the approach, such as the saved manual effort, may be measured. Due to violations of the computational model not all annotations may be computed correctly. For instance, the theoretical evaluation of accuracy in computing missing annotations (c.f., Sec. 7.3.4) shows that some computed annotations are not specific enough. Therefore, the second evaluation question considers how many of the annotations are propagated correctly compared to the number that have to be repaired manually.

Second Evaluation Question: Saved Manual Effort The main purpose of the automated annotation propagation is to reduce the manual effort of annotating domain models in one product line repeatedly. For that reason, the evaluation examines the following question, additionally:

EQ2 How much manual annotation effort is saved by the automated propagation?

To measure item **EQ2**, we compare the number of annotations that can be applied to the target model with the number of annotations that are assigned correctly. As a consequence, the value of the saved annotation effort is the inverted actual error (c.f., Def. 10.1.3):

Definition 10.1.4: Saved Annotation Effort

Let $\#map_t$ be the number of all elements that can be annotated in the multi-variant target model. Further, let a_{Fc} be the number of annotations that have to be corrected to achieve commutativity.

The saved annotation effort t_{sav} constitutes as follows:

$$t_{sav} = \frac{\#map_t - a_{Fc}}{\#map_t}$$

10.2 Evaluation Setup

For evaluating the goals, we employ laboratory experiments [SF18] which are conducted in our evaluation framework. The measured results as well as the setup are openly available [Gre22].

Road Map The first part of this section (i.e., Sec. 10.2.1) presents the technical setup of the evaluation, including a description of the machine executing the transformations used for evaluation and the implemented testing infrastructure which is used to measure to what extent the propagated annotations suffice to satisfy commutativity in the subject systems. After presenting the technical facets of the framework for evaluating commutativity, Sec. 10.2.2 introduces the subject product lines which are used to evaluate the correctness. To obtain a diverse set of evaluation data, Sec. 10.2.3 further explains transformation definitions in various languages which convert the respective source multi-variant product line models.

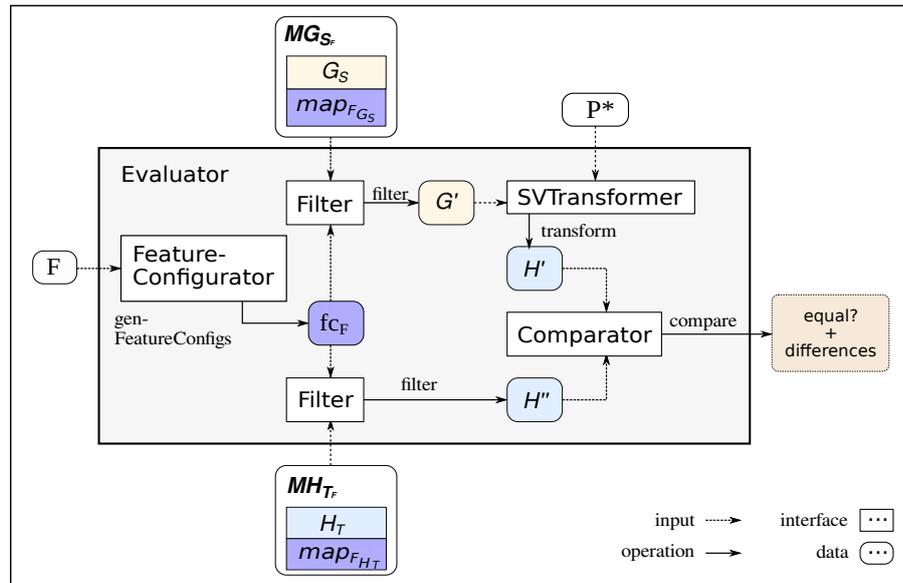


Figure 10.2.1: Schematic overview of commutativity evaluation framework.

10.2.1 Commutativity Evaluation Framework

For conducting the evaluation automatically, we implemented an evaluation framework to execute the commutativity criterion. This section illuminates the technical and conceptual characteristics of the framework subsequently.

Technical Setup The quantitative evaluation of the trace-based and aspect-oriented propagation is executed with the following technical setup: All transformations are conducted on an Ubuntu 20.04.4 LTS 64bit operating system. An Intel Core i7-8550U CPU 1.80GHzx8 with 23.4 GiB of memory builds the key part of the computing hardware. The evaluation framework is implemented in Eclipse Modeling Tools version 2021-12 (4.22.0).

Overview Fig. 10.2.1 provides an overview of the implementation of the evaluation framework. Input to the framework are the resources of the multi-variant source and target models, MG_{S_F} and MH_{T_F} , as well as the feature model (F) and the transformation definition (P^*). The evaluation compares the transformed source product H' with the filtered target product H'' for each valid feature configuration and manifests the results. A `csv`-file summarizes the results of comparing and differentiating the single-variant models and answers the question to what extent commutativity is satisfied in the examined scenario.

The following four parts offer more detail on each of the steps in the sequence they are executed. Particularly, the following descriptions explain how the necessary interfaces, depicted as rectangles in the figure, are realized as well as the representations of the generated artifacts and inputs, shown as half-rounded rectangles.

I Feature Configuration Generation

In the first step of evaluating the commutativity criterion, all valid feature configurations are created. Therefore, the feature model serves as input to the `FeatureConfigurator` and a set of feature configurations, shown only as a single feature configuration, f_{C_F} in the figure, results as output. If the set of optional features is large, which may make the consecutive comparison of all filtered models infeasible, a respective realization of the interface may only compute a significant *sample* thereof.

II Filter Mechanism

The feature configurations are input to a *filter* mechanism which receives an annotated multi-variant model as well as the feature configurations to create a single-variant model. The filter removes all model elements of the multi-variant model carrying an annotation which does not satisfy the given feature configuration to build the derived single-variant model. Depending on the kind of target representation, one (model) filter mechanism may suffice or another will be required. The latter may be the case if annotated source code should be configured which requires a filter exhibiting preprocessing capabilities.

Accordingly, the second step of the evaluation framework filters the source and the target product by each valid feature configuration. While the single-variant *source* model is transformed thereafter, the derived single-variant *target* representation is compared with the result of this transformation. The following three paragraphs present details how Famile realizes the model filter and how we conditionally compiled the multi-variant source code.

Famile Filter The tool Famile employs a hierarchical filter, denoted as `ConditionalCopier`. The mapping model stores a propagation mechanism which the copier uses to compute selection states of a model element. The propagation mechanism may be executed in *forward and backward* direction¹ which either is able to override annotations top-down or bottom-up, respectively. As a third propagation option, these dependencies can also be ignored. Two additional options allow for assuming all mapping elements without an annotation to be *included* before the conditional copier is executed and the possibility to propagate selection states *transitively*. Finally, Famile allows to define context-specific repair operations in the separate DSL `SDIRL` which may help to restore validity when the hierarchical filter would derive a model which violates constraints defined in its metamodel, such as having dangling references.

Famile Propagation Mechanism The F2DMM models representing the examined subject product lines (c.f., Sec. 10.2.2) employ the default propagation mechanism which propagates selection states in forward direction transitively and includes incomplete mappings a priori (i.e., it assumes the annotation `true`). For instance, if an attribute mapping is not annotated, the *include-incomplete* option assigns the annotation `true` implicitly such that the attribute will be pertained in the filtered product if the annotation of its container, the object mapping, satisfies the feature configuration. Please note: Although the include-incomplete option has the same effect as applying the parent annotation, we argue that our completion strategies are tool-independent and explicitly state the annotation whereas the actual annotation with the Famile propagation mechanism results from a transient computation which is not visible to the user.

Based on the *forward*, *transitive* and *include-incomplete* configuration, our default `FamileFilter` mechanism executes the `ConditionalCopier`. This class copies the domain model and includes only those element, for which a `SELECTED`-state is computed when applying a feature configuration to the mapping model. The mapping models of the subject product lines are *not* enriched with an `SDIRL`-model, such that no metamodel-specific repair operations are available.

Preprocessor Filter for Source Code In contrast, for ‘filtering’ source code we employ preprocessing technology that allows for conditional compilation and to derive source code variants from the multi-variant source code platform. Since Java does not comprise a built-in preprocessor and the support of (previously) existing Java Comment preprocessors discontinued or requires specific execution setups, such as Maven projects, a standalone Xtext-based preprocessor was developed in a student project supervised by the author of this thesis.

The preprocessor is flexible inasmuch as it can be configured to adjust the symbols which open and close comments in the programming language and the style of the preprocessor contents. It receives a *flag*-file which states the name of the features literally and a binary selection state for each of these features. However, the preprocessor itself is unaware of the feature model. Thus, it

¹ Forward propagation in Famile means that an *active* mapping element, which requires an *inactive* mapping element, becomes *inactive*, too. Vice versa, the *inactive* mapping element becomes *active* in case the reverse propagation strategy is selected.

cannot check the completeness of the feature selection in the *flag*-file which has to be ensured by the facility which transforms a feature configuration into that *flag*-file. The second configuration file of the preprocessor determines the escape sequences which mark comments in the source code and the keywords to open and close preprocessor directives in these comments. Based on these configuration options, the preprocessor cannot only be used for Java source code files but also for different languages without such built-in capabilities, such as Python.

III Single-Variant Transformation

The third step of the evaluation iterates the set of derived source variants and transforms them into the target representation. This step employs the same transformation definition which was reused to transform the multi-variant source model into the multi-variant target model.

Accordingly, this step employs a realization of the `SVTransformer` (c.f., Sec. 9.2.2) which created the multi-variant target model. This means that it must be possible to initiate the transformation engine programmatically. For instance, the documentations of the ATL/EMFTVM and the QVT-O engines describe how to execute a transformation definition written in its language based on a Java program. On the contrary, QVT-R transformations, which should be executed with `mediniQVT`, cannot be executed in a recent Eclipse environment without significant adaptations, such as downgrading the versions of the employed metamodels (if registered ones are used) and building an environment which employs a Java version smaller than Java 1.8. For that reason, this step of the evaluation framework (re-)uses only the single-variant transformers, which are implemented as realizations of the `MuVaTra SVTransformer` interface (i.e., a `BXtend`, `ATL/EMFTVM` and `QVT-O` but *no* `QVT-R` transformer). In case of M2T transformations, we employ an `XPandTransformer` which initiates a given MWE2-workflow with the source model and the path of the target directory given as further parameters.

IV Comparison

After having transformed the derived source models, the fourth step compares the transformed (H') with the filtered target model (H''). Accordingly, this step compares either two models or the Java classes contained in two directories.

EMF Compare In case of comparing two models, we employ the capabilities of the **EMF-Compare** framework (c.f., Sec. 7.4.3). While this framework, which serves mainly for versioning models, is highly configurable and can be customized for performing metamodel-specific comparisons, we employ one of its default comparison methods to compare two models conforming to the same target metamodel. As a consequence of comparing equivalent model elements with different (or even no) UUIDs, we do not use these values to compare the model elements but use the default *property-based* comparison. Thus, the framework tries to match two objects based on their kinds of structural features and their values.

According to the developer guide of `EMFCompare` [200+], the comparison consists of multiple steps which first matches `EObjects` and creates an initial difference model based on the computed mappings. Thereafter, the comparison method iterates the detected differences and tries to determine equivalent changes which produce the same result in the end. Those differences are collected in form of an *equivalence* in the resulting difference model. Besides matches, equivalences and differences the resulting comparison model may contain conflicts arising from differences which are not recognized as an equivalence. In concrete, the resulting object of type `Comparison` allows to retrieve the following information:

- `getDifferences()`, which returns all objects of type `Difference` and its children.
- `getEquivalences()`, which returns the list of contained equivalences.
- `getMatches()`, which returns the list of all recognized matching objects.
- `getConflicts()`, which returns a list of conflicts with a version stored at a repository.

While the number of differences and equivalences may vary in our comparison results, the number of matches is always one (and split up in more fine-grained submatches) and the number of conflicts is always zero. The latter is only available if a one local model file is compared with a remote one stored in a repository.

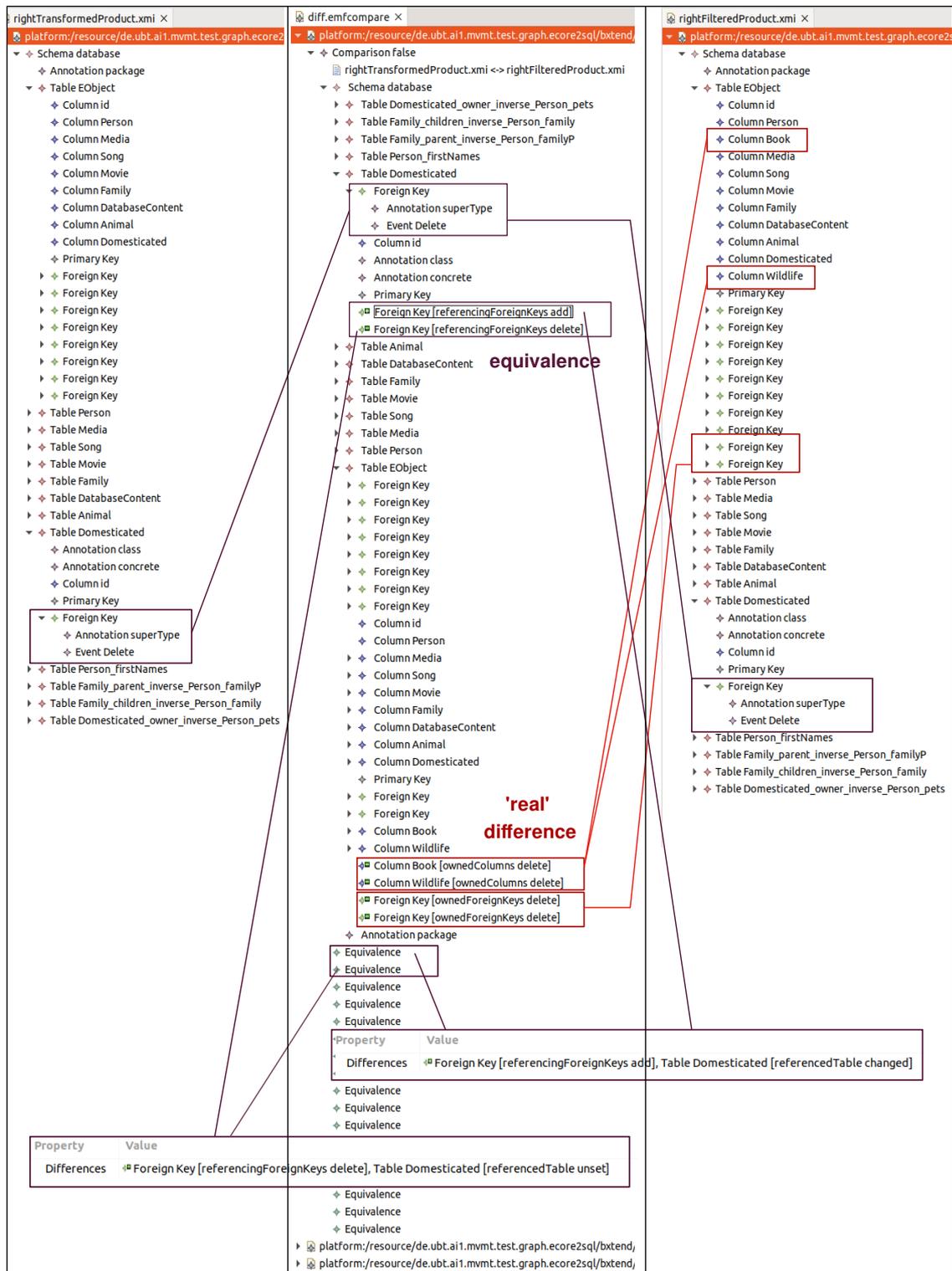


Figure 10.2.2: EMFCompare *difference model* of transform-filter and filter-transform database content variants.

Example 10.2.1: EMFCompare Difference Model

As an example of a comparison outcome, Fig. 10.2.2 displays a difference model created by the EMFCompare framework using the default property-based comparison. The difference

*model results from comparing the transformed and the filtered SQL models resulting for a configuration which deselected the features `Wildlife` and `Book` in the DBC product line. The figure demonstrates two kinds of mismatches: real differences and equivalences. Real differences occur in this transformation in the `EObject`-table. In this case, no columns for the classes `Book` and `Wildlife` are present in the transformed variant, depicted on the left side of the figure, whereas they are present in the filtered target variant, depicted on the right side. This is a result from applying the parent annotation mapped onto the `EObject` table (i.e., `DBContent`) onto the column and foreign key which are not linked by a trace element and, thus, are annotated by the completion strategy. Annotations provoking a real difference should be repaired to achieve commuting transformations. In contrast, the `EMFCCompare` mechanism detects and records differences which are recognized as equivalences in the consecutive comparison step. The difference model exemplifies this for the foreign key of the table `Domesticated`. While the transformed and filtered product store exactly the same foreign key, the comparison model records not only a match but also an *add-* as well as *delete-*difference. These two differences are recognized as equivalences and recorded this way at the bottom of the model. Consequently, this equivalence does not affect the commutativity of the transformation.*

For computing the correctness in terms of the size of the error, we employ the number of adjusted differences. In concrete, we subtract the number of differences which were recognized as equivalent from the total number of differences stored for the comparison object. This allows to compute the severity error and to draw a conclusion about the error introduced to wrong annotations.

Source Code Comparison In contrast to comparing two models with the model comparison framework `EMFCCompare`, comparing two source code projects requires to compare the Java source code syntactically. For each feature configuration the files of the created source code project are compared with the ones filtered from the multi-variant target source code.

The comparison removes all white spaces, in order to mitigate counting empty lines which may be introduced by the source code generator, inadvertently. Moreover, each line containing a comment is removed already during the derivation of the configured product². Furthermore, the comparison removes the contents of protected regions, such that only the generated text without preprocessor comments is compared and additional lines and any other white spaces can be neglected. After this step, the default Java String compares the resulting Strings syntactically. Despite the pre-processing and although a number is computed by the String comparison, the returned difference value³ does not allow to draw a conclusion on the severity of the error. Therefore, we count only the number of files incorporating a difference, sum it up and weigh it against the number of all files that are created during the commutativity check. Sec. 10.2.1, I provides the details on the computed error measures for evaluating M2T transformations.

Please note: This form of comparison is performed right after the *batch* generation of the source code. If source code was added manually to the generated multi-variant source code (e.g., the body of a method), a comparison after a subsequent derivation would render a configuration as erroneous because this information is not present in the source model and therefore not reflected in the source code generated for a filtered source variant.

Result Manifestation In the last step, the evaluation framework measures the correctness by counting the number of differences recorded in the performed comparisons. Based on that number, it computes the absolute and the severity error.

For verifying the result, the model comparison persists the created filtered and transformed variants as well as the difference model. The persistence of the compared variants allows to inspect

² The default derivation mechanism of the FlePP preprocessor retains the comments holding preprocessor directives in the source code to allow for reconfiguring the source code. Therefore, we added a postprocess which removes all comments which either mention a directive or invisible source code fragments which are deselected in the configuration.

³ The distance between the first two non-matching characters constitutes the returned value of the default String comparison (`str.compareTo(str2)`.)

individual configurations and the resulting derived variants manually. This information serves the developer to gain knowledge about wrong annotations in the target model.

The text comparison also persists the derived variants in order to inspect the differences. The error values are computed similarly but not equally as the measures for comparing models. Since the EMFCompare difference model counts the number of differences explicitly it is possible to regard that number per configuration. Conversely, the text comparison requires to compare the contents of different files per configuration. Since we employ the default String comparison at the granularity of file content, the error measure counts how many files are mismatching per configuration. Both adapted measures for the absolute and the severity error are stored in the resulting comparison file.

10.2.2 Product Lines

To diversify the evaluation, we select three product line models. They range from small to medium size in terms of the number of valid configurations. Furthermore, the feature models are as diverse as to incorporate all possible kinds of relationships between features and the assigned annotations to the model elements. The three parts of this section introduce the Database Content, Graph and Home Automation System product lines subsequently in the mentioned order.

I Database Content Product Line

The product line for *Database Contents* (**DBC**) product line is introduced in this thesis to demonstrate various properties of product line engineering and to eventually illustrate the contributed propagation mechanisms. Thus, it serves as one subject system for evaluating the propagation mechanisms. The product line allows to configure (i.e., filter) the contents of a database which may comprise data about persons and animals as well as different forms of media. For the sake of completeness, this sections summarizes its important characteristics for evaluating commutativity.

Statistics Feature Model The feature model comprises 14 features in total, out of which 12 are optional. Only the child feature `name` of the feature `Person` is mandatory in case the feature `Person` is selected.

The hierarchical constraints encompass the necessity that at least one feature directly underneath the root feature has to be selected (inclusive Or-group). Similarly, at least one feature of the group `Media` and `Person` has to be selected. The feature `Animal` forms the root of an Xor-group where at maximum one feature can be selected (i.e., `Domesticated` or `Wildlife`). Moreover, a crosstree constraint *requires* that whenever the feature `Pet` is selected the feature `Domesticated` (and transitively the feature `Animal`) is selected, too. As a result, the feature model constitutes 234 *valid* feature configurations.

Please note: The evaluation does not regard the extension of the feature model with the optional feature `MultipleNames` which exemplified a fine-grained mapping. As the evaluation does not examine the analysis of bytecode instructions for propagating fine-grained annotations, as explained in the final part of Sec. 9.3.2, the additional feature would not be considered in the propagation.

Statistics F2DMM and Domain Models Fig. 9.1.2 demonstrates the feature and mapping model of the DBC product line in tree representation as represented in the F2DMM editor. The Ecore domain model comprises *ten* classes, *one* enumeration type, 23 structural features and *one* operation. *Five* out of the *ten* classes inherit from another class. In total, the elements together with the literals and explicit inheritance objects sum up to 69 *object* mappings. As Table 10.1 enumerates, in total, the mapping model encompasses 200 source mapping elements (including those for structural features).

The UML class model which results from the Ecore2UML ATL/EMFTVM model transformation (c.f., second part of Sec. 10.2.3) is further transformed into a Java domain and mapping model (c.f., third part of Sec. 10.2.3). The input UML source mapping model of the DBC product line comprises 301 mapping elements.

In summary, it is a medium-sized artificial product line.

Table 10.1: Overview of subject product lines model statistics. The Database Content (DBC) and Graph product line originally conform to the Ecore metamodel and are transformed into UML models for further processing, the HAS model conforms to the UML metamodel.

Product Line	#features (optional)	n (all valid fc)	$ m_s $	# map_s (Ecore model)	# map_s (UML class model)
DBC	14 (12)	234	69	200	301
Graph	14 (10)	180	84	239	368
HAS	25 (21)	16560	221	–	559

II Graph Product Line

The product line for Graphs [LB01] allows to build different types of graphs. A Graph can possess colored nodes, weighted or directed edges and allows to perform Graph algorithms, such as different search mechanisms. Besides a BFS or DFS, which can be performed on the Graph, the class `Algorithm` offers distinct methods to execute graph algorithms, such as computing a minimum spanning tree. Fig. 10.2.3 demonstrates the corresponding mapping and domain model visualized in the Famile F2DMM editor.

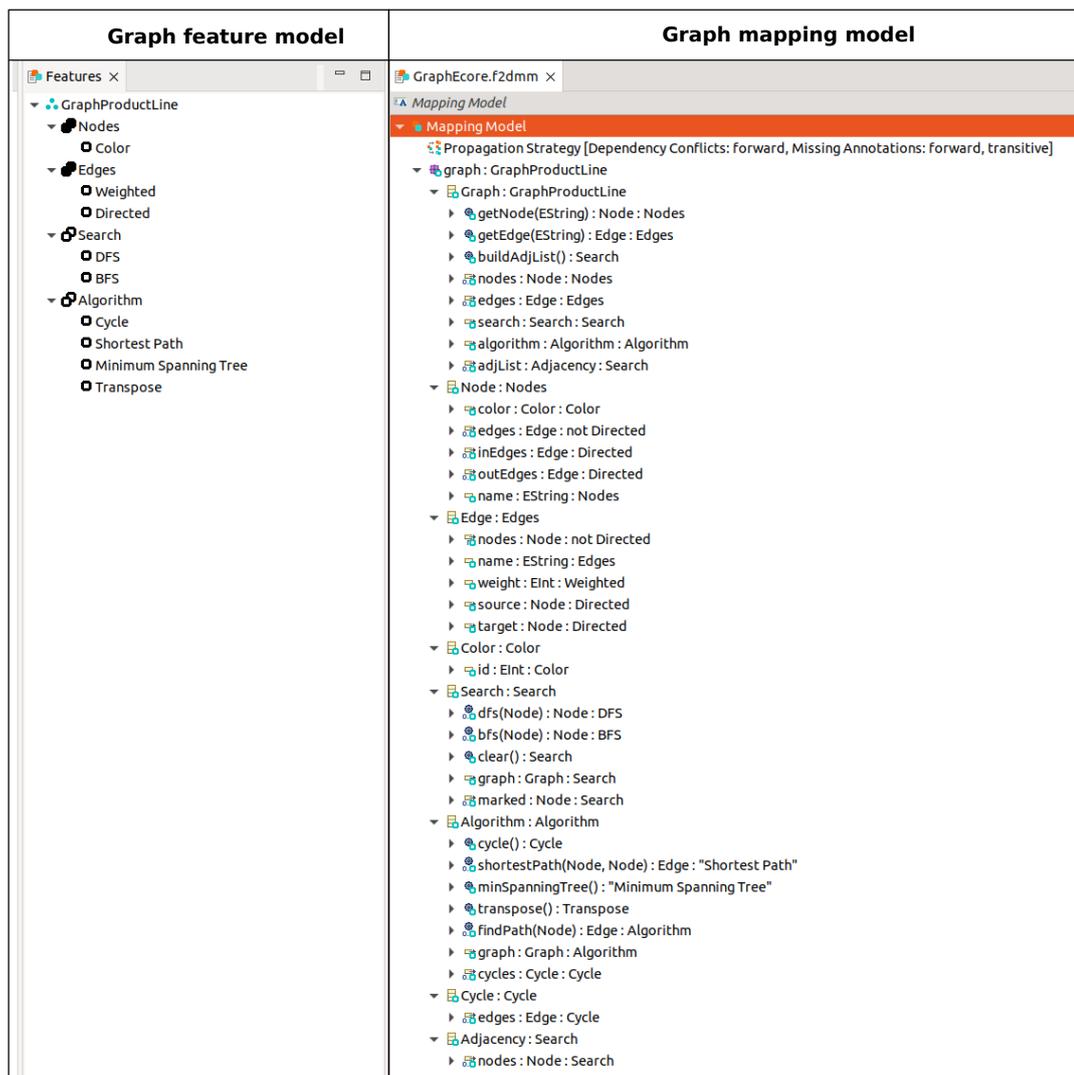


Figure 10.2.3: Graph product line feature model and F2DMM model.

Statistics Feature Model The feature model comprises 14 features in total, out of which *ten* are optional. The hierarchical constraints encompass an OR-group to refine the properties of edges which can be weighted, directed or both, an XOR-group for the search mechanism, which is either a DFS or a BFS search as well as another OR-group for realizing different algorithms. Furthermore, crosstree constraints foster that the features `ShortestPath` *requires* the feature `Weighted` whereas the feature `Cycle` and the feature `Transpose` each require the feature `Directed`.

As a result, the feature model constitutes 180 valid feature configurations. Although the DBC and the Graph product line comprise the same number of optional features, the Graph product line allows for less *valid* configurations due to the higher number of constraints in its feature model.

Statistics F2DMM and Domain Models As Fig. 10.2.3 displays, the Ecore domain model comprises *eight* classes without inheritance relationships, *eleven* operations and 22 structural features resulting in 84 *object mappings* for model elements. In total, the mapping model encompasses 239 mapping elements which include not only mappings for the `EObjects` but also for their structural features.

Similar as in the DBC product line, the UML class model which results from the Ecore2UML ATL/EMFTVM model transformation is transformed into a Java domain and mapping model. The input mapping model of the Graph UML class model comprises 130 *object mappings* and 368 mapping elements, in total.

Altogether, the product line is medium-sized and can be employed in real-world applications to derive variants of Graphs and implement the method stubs for the required algorithms.

III Home Automation System Product Line

In their foundational book for product line engineering, Pohl et al. [PBL05] describe a product line which realizes *Home Automation Systems (HAS)*. The contents of this product line were reconstructed as UML package, class and state models at the Chair of Applied Computer Science I of the University of Bayreuth. We employ this product line to further evaluate the trace-based propagation approaches based on a larger and more complex subject and transform the HAS class model into a Java MoDisco model.

Fig. 10.2.4 depicts the feature model as well as the F2DMM model at the granularity of classes. The HAS requires a possibility to connect remotely and an identification mechanism. Furthermore, it can control different kinds of peripherals, such as the heating and air conditioning systems and a microwave oven. Connections can be secured via VPN or SSH and add-on packages may allow controlling the system through apps or incorporating weather monitoring.

Statistics Feature Model The feature model comprises 25 features with 12 attributes. 21 features are optional. Furthermore, the following constraints are present:

- the feature `IEEE 11b` requires the feature `IEEE 11a`
- the feature `IEEE 11g` requires the features `IEEE 11b` and `IEEE 11a`
- at least one and at maximum two identification mechanisms can be selected
- the features `MagneticCard` and `FingerprintScanner` both exclude the feature `Keypad`
- the feature `SecureConnection` builds an XOR-group allowing for only one of either the `SSH` or the `VPN` feature to be selected at the same time
- the feature `Add-onPackage` constitutes an XOR-group allowing for only one of its children features to be selected

Due to these constraints, the feature model compiles 16560 valid feature configurations.

Statistics F2DMM and Domain Models Fig. 10.2.4 displays the HAS feature model and its F2DMM model, which demonstrates that the domain model comprises a plethora of distinct elements. The UML model organizes 15 packages, 36 classes and interfaces, *five* associations as well as *five* nested classes. In addition, the model comprises operations and structural features as well as 23 generalizations and one interface realization. The three classes, `MicrowaveOven`, `TemperatureConditioner` and `RollingShutters`, declare their behavior by the means of a state

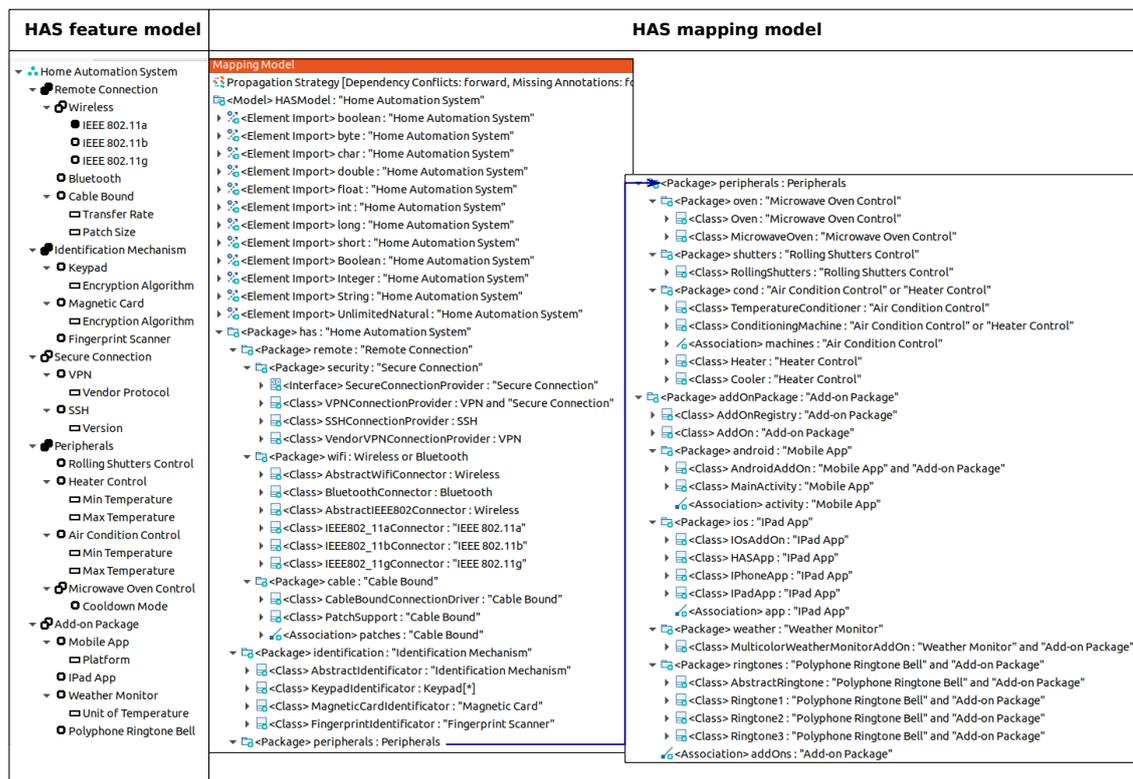


Figure 10.2.4: Home Automation System product line feature model and F2DMM model.

machine which will not be transformed by the UML2Java transformation (c.f., third part of Sec. 10.2.3). Further operations and properties present on the classifiers sum up to 221 model elements in total and a total number of 559 source mapping elements.

Annotations Many of the features are mapped 1:1 onto equally or similarly named packages and classes. Therefore, the remainder highlights annotations which deviate from this behavior:

The package `wifi` is present in variants derived from configurations which either select the feature `Wireless` or `Bluetooth`. Similarly, the package `cond` comprises implementations to realize the air conditioning control as well as the heating control. Therefore, the two respective features are combined in a disjunctive expression and mapped onto the respective packages. Furthermore, the interface `SecureConnectionProvider` realizes the *Singleton* design pattern. Thus, it comprises a static class field which requires the feature `VPN` to be selected, too. Consequently, the feature expression `VPN` or `"Secure Connection"` is mapped onto the corresponding property. The `BluetoothConnector` specializes the `AbstractWifiConnector`. Accordingly, the expression `Bluetooth` and `Wireless` is mapped onto the contained generalization.

Summary Altogether, the product line is medium-sized. Although the number of 16560 possible configurations is almost 100 times larger than the number of configurations in the Graph product line, it is still not considered of large size because it does not comprise hundreds of optional features resulting in millions of configurations that can be created. In contrast to the DBC and Graph product lines, the feature model incorporates *excludes*-dependencies as constraints. Thus, the design complements the selection of feature models to cover each type of constraint at least once in an examined product line. Additionally, as the last row of Table 10.8 enumerates, the domain model and mapping model establish a medium-sized *system control* product line, which is an experimental but real-life oriented subject system.

10.2.3 Model-To-Model Transformations

In order to further diversify the evaluation setup and to give evidence for language- and definition-independence of the propagation approaches, we examine three kinds of M2M transformations and vary the transformation languages.

Road Map On the one hand, we reuse and re-implemented the model transformation definition which is commonly used for demonstrating the capabilities and syntax of model transformation languages: It transforms *class models* into a *relational database schema* to establish object-relational mappings. The first part of this section describes the specifics of the **Ecore2SQL** transformation which results from harmonizing and employing an existing bidirectional version for this task.

On the other hand, the evaluation examines the behavior in an **Ecore2UML** class model transformation to refine the Ecore source models. The specific of this transformation are explained in the second part of this section. Finally, an already existing UML class model to Java MoDisco transformation turns the resulting UML class model into a Java model. The third part of this section describes details of this **UML2Java** transformation.

The three parts are organized in the same way: they first present details of the metamodel and describe thereafter how the rules generate target elements from the source elements. As we employ a trace-based propagation, next the information stored in the resulting traces is sketched. Furthermore, if language-specific behavior is relevant, the closing of the part will shed light on it.

I Ecore to Relational Database Transformations

Several transformation language documentations employ the *class model to relational database schema* transformation to demonstrate the syntax and capabilities of their language. To employ these transformations and allow for justified comparisons, we have streamlined an existing ATL transformation [INR05], as well as a similar QVT-O Eclipse example, which both transform simplified forms of class models into a relational database schema. The resulting harmonized transformation definitions, denoted as **Ecore2SQL** transformation in the sequel, transform an Ecore class model into a metamodel representing relational databases, which is described in the following paragraph. The subsequent paragraphs elaborate on the details of the reused BXTend and the re-implemented ATL/EMFTVM and QVT-O definitions, respectively.

Relational Database Schema Metamodel For representing relational database schemata, we employ an already existing metamodel [Wes15], denoted as *SQL* metamodel, which conforms to the Ecore meta-metamodel. The metamodel represents the target of a bidirectional incremental transformation specified in QVT-R and was used to specify the same transformation behavior in a BXTend transformation in a student project.

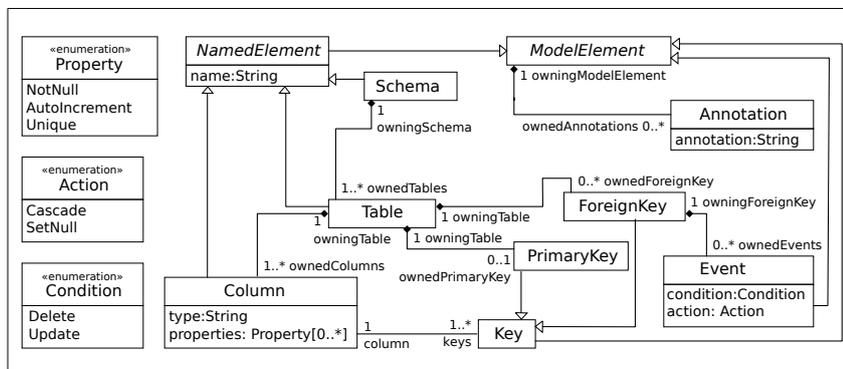


Figure 10.2.5: Relational database schema metamodel.

Fig. 10.2.5 represents the main elements of the metamodel. Each model element may carry up to several annotations which help to uniquely transform one of its instances into a Ecore model in

the backward direction. In our transformation definitions we assign these annotations but do not consider the backward transformation.

A **Schema** forms the root of each instance and contains several tables. Each table comprises columns, a unique primary key and up to several foreign keys. A column is of a certain type, stored as String literal, and owns properties. Furthermore, it may serve as the key for other tables. For instance, typically a column named ‘id’ serves as primary key of a regular table. While the primary key belongs uniquely to a table, the foreign key references the column of another table in addition and possesses several events which perform an action for a given condition. For instance, an event may be triggered upon a delete-action which may cause a cascading deletion of all corresponding columns and tables.

Common Transformation Rules The three transformation languages compose a different number of rules to achieve the same SQL target model. Therefore, the following paragraphs summarize the general behavior of the Ecore2SQL transformation definition [Wes15].

The transformation constructs the following mappings:

1. **EPackage: Schema**
2. The **Schema** maintains an *object table*, for managing the unique ID’s of each table
3. **EClass: Table** including a **PrimaryKey** which is a column named `id` of type `int`.
4. **Inheritance**:
 - the table of a *root* class (i.e., inheriting from `EObject`) possesses a *foreign key* into the object table
 - table of a *specific* class possesses a *foreign key* into the table representing the `superType`
5. **Single-Valued EAttribute: Column**
6. **Containment EReference: Column and ForeignKey** of the table representing the opposite contained class
7. **Single-valued, unidirectional, non-containment EReference: Column and ForeignKey**
8. Remaining types of **EReference: Table**. The rules must ensure to create only *one table* for a bidirectional reference.

Accordingly, the transformation assumes the following facts:

- *single, unique package* which serves as root of the class model
- *single inheritance*
- structural features are either single- or multi-valued; any specific value as upper bound (e.g., five first names) can *not* be fostered

As a consequence, additionally neither *enumerations*, *datatypes*, *operations*, nor attributes with an enumeration type are transformed.

The following explanations describe in detail how the Bxtend definition transforms an Ecore class model into a corresponding SQL model. While the first paragraphs illuminate the Bxtend transformation in detail, the subsequent paragraphs focus on implementation details of the ATL/EMFTVM and QVT-O specifications.

Bxtend The reused Bxtend transformation composes five Xtend classes which implement the rules to transform packages, classes, the inheritance relationships, attributes and references in the given order. It implements the same behavior as the QVT-R transformation [Wes15]. On the one hand, the rules explicitly implement the incremental behavior by checking whether elements, which should be created, exist already and only modify their properties in case they exist. On the other hand, the rules implement the backward direction which we do not regard in the sequel.

In addition, each of the classes representing a rule inherits from the class `Elem2Elem`. This abstract class offers functionality to maintain the source and target model. It generates and updates trace information generically and allows to retrieve source and target elements from a given trace element, denoted as `Corr(espondence)` object.

Transforming EPackage The rule implemented in the class `Package2Schema` creates a schema for each Ecore package and inserts a table, named `EObject`, which owns an `id` column of type `int` and the properties `NOT_NULL` and `AUTO_INCREMENT`. Furthermore, the schema receives the annotation `package` for representing the source object type from which it was created.

Transforming EClasses The rule executed in the second place transforms each class into a table. As an example, Listing 10.2.1 presents the implementation of the corresponding forward BXTend rule of the class `Class2Table`. Line 4 either retrieves the table from the trace or creates a new one whereas Line 9 adds the table to the schema which was created for the package containing the processed class. Furthermore, for each `EClass`, the rule adds a primary key as well as the annotation `class` in Lines 12 and 14, respectively. The method in Line 12 creates a primary key which is named `id`, of type `int` and possesses the property `NOT_NULL`. If the object table does not own a reference key of the class yet, it will be created in Lines 18-21 or its properties are updated in the following lines.

Listing 10.2.1: BXTend forward transformation rule creating a Table for a given EClass.

```

1 override sourceToTarget() {
2   sourceModel.allContents.filter(typeof(EClass)).forEach[ ec |
3     val corr = ec.getOrCreateCorrModelElement(ruleID)
4     val tbl = corr.getOrCreateTargetElem(targetPackage.table) as Table
5     tbl.name = ec.name;
6
7     val schema = (ec.EPackage.corrModelElem.targetElement as Schema)
8     if (!schema.ownedTables.contains(tbl))
9       schema.ownedTables += tbl
10
11    if (tbl.ownedPrimaryKey === null)
12      tbl.createPrimaryKeyAttr
13    if (tbl.ownedAnnotations.size == 0)
14      tbl.addAnnotations(Arrays.asList("class", ec.kind))
15
16    val key = tbl.referenceForeignKeys.findFirst[k |
17      k.owningTable == schema.eObjectTable]
18    if (key === null) {
19      val col = schema.eObjectTable.createForeignKeyAttr(tbl.name, tbl)
20      col.properties.clear
21      col.properties += Property.UNIQUE
22    } else {
23      key.column.name = tbl.name
24      key.column.properties += Property.UNIQUE
25      key.referencedTable = tbl
26    }
27  ]
28 }

```

Transforming Class Inheritance In the set of subsequently executed rules, for each class which inherits from another class (i.e., the size of the `eSuperTypes` is larger than zero), the rule in class `Generalization2Realization` creates a foreign key. This key references the table representing the super class and is stored in the table representing the sub class. The column `id` of the super table serves as column referenced by the foreign key. If a class does not inherit from another class, a similar foreign key will be created which references a corresponding column in the object table and carries the annotation `root`. Each of the foreign keys triggers a cascading deletion, represented as `Event`. Accordingly, whenever one of the following rules creates a foreign key, additionally an event is created.

Transforming EAttributes Next, the transformation processes the attributes: A column is created for a *single-valued attribute*. The rule assigns its name and determines the type which

transforms the Ecore primitive datatypes, `EInt` and `ELong` into the String `'int'`, the `EBoolean` into `'boolean'`, the `EDate` into `'data'`, the `EString` into `'varchar(30)'` and `EDouble` into `'double'`. In addition, the rule assigns the annotations `attribute` and `single`.

If the attribute is *multi-valued*, instead, the rule creates a table. The rule combines the name of the containing class with the name of the attribute and assigns it as name of the table as well as the annotations `attribute` and `multi`. Furthermore, the rule creates an id-column and assigns the table as owner as well as a foreign key which is contained in the created table and references the table, which was created for the class containing the processed attribute as well as the created id-column as `column`.

Transforming EReferences In the end, the transformation class `Reference2Relation` handles `EReferences`. It distinguishes containment from uni- and bidirectional crossreferences as well as the cardinality of the referenced elements.

For *containment references* the transformation creates a column. The assigned name depends on the directionality of the reference. It combines the name of the reference with the suffix `_inverse` in case of a unidirectional reference whereas the name of the reference name is concatenated with the literal `_inverse_` and the name of the opposite reference. The column is inserted into the table that represents the class of the referenced type. Furthermore, the rule inserts an additionally created foreign key which mentions the created column as column and references the table which represents the containing class of the transformed reference. The same rule is applied for single- and multi-valued references which yield only different annotations stored in the foreign key and the column, stating either `single` or `multi` with respect to the upper bound of the attribute. Ex. 10.2.2 illustrates the results of transforming different kinds of references.

Example 10.2.2: Created SQL Elements for Ecore References

On the left, Fig. 10.2.6 shows the element resulting from transforming the containment reference between the class `DatabaseContent` of our demonstrating example and the class `Book`. The table which was created for the class `Book` contains a column which states that an inverse relationship exists between the database and the book table as well as a foreign key which references the database table and stores the column. The main target element that is recorded in the trace is the created column in the table `Book`.

The transformation of a *single-valued unidirectional crossreference* behaves similarly as the transformation of a single-valued attribute. The rule creates a column which is owned by the table representing the class which contains the reference as well as a foreign key which references the table created for the referenced type.

Instead, a *multi-valued unidirectional crossreference* provokes the creation of a table. The table is called by the name of the containing class combined with the reference name and is inserted into the schema. The table receives a foreign key named, `id`, which references the table representing the containing class, as well as a second foreign key, named `reference` which references the table representing the referenced type of the processed `EReference`. As annotation, the table stores the strings `cross`, `multi` and `unidirectional`.

Finally, a *bidirectional crossreference* is transformed into a table, which carries a name composed of the opposite reference name and the reference name. The `BXtend` rule ensures that only one of both references creates a table by comparing the names of the processed and the opposite reference and only transforming the one with a smaller String comparison value. It creates a `source` and a `target` column as well as corresponding foreign keys. The source foreign key references the table representing the containing class of the processed reference whereas the target foreign key references the table representing the referenced type. Both columns must not be `null`. The table carries the annotations `cross` and `bidirectional` as well as details about the multiplicity of the two ends of the reference.

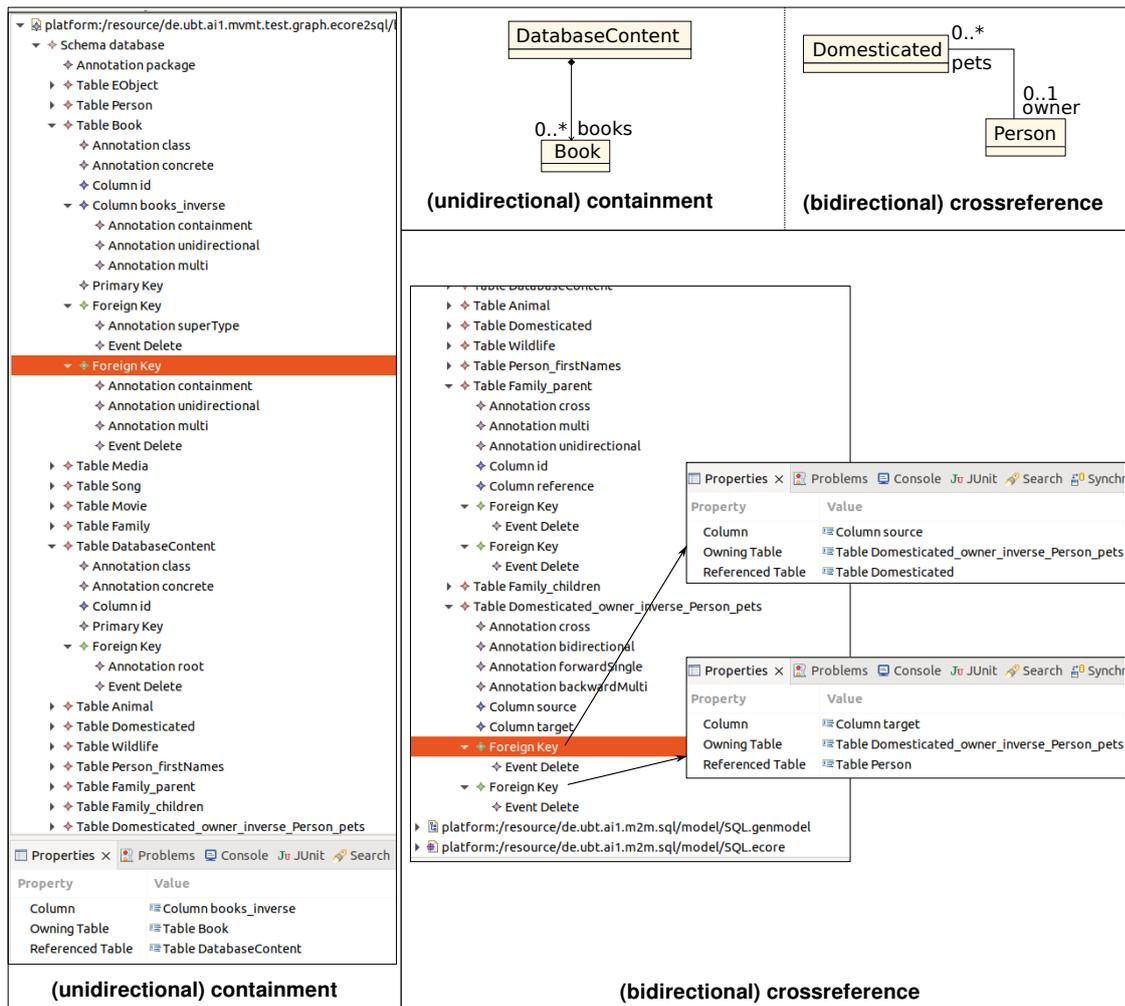


Figure 10.2.6: Transformed EReferences into columns, foreign keys, events, and tables.

Example 10.2.3: SQL-Table Created for the Bidirectional Crossreference in DBC Product Line

Fig. 10.2.6 further depicts an example of a transformed bidirectional crossreference on the right side. In the example the reference between the classes *Domesticated* and *Person* which declares the ownership of a pet is transformed.

The rule turns the crossreference into a table which possesses two columns and two foreign keys. While one foreign key references the table which was created for the class *Domesticated*, the second foreign key references the table *Person*. The corresponding columns are denoted as *source* and *target*, respectively.

The sql annotations declare the type of the transformed reference. In this case a *domesticated* pet belongs to exactly one person. Therefore, the sql annotations state *forwardSingle* for the reference to the class *Person* and *backwardMulti* for the reference to the class *Domesticated*.

Resulting Trace To this end, relevant information for propagating annotations resides in the incrementally maintained trace. Since the correspondence model (i.e., the trace information) created by BXTend contains only 1:1 correspondences, it does not record all elements created by a rule application. In concrete, the incomplete trace stores the following correspondences:

- EPackage ⇔ Schema

- **EClass** \leftrightarrow **Table**
- **EAttribute** \leftrightarrow
 - single-valued: **Column**
 - multi-valued: **Table**
- **EReference** \leftrightarrow
 - containment: **Column**
 - single-valued unidirectional: **Column**
 - multi-valued unidirectional: **Table**
 - any other: **Table**

Accordingly, the trace maps an **EPackage** and an **EClass** onto a **Schema** and a **Table**, respectively, whereas for references and attributes the mapping depends on their meta structural features. The trace records a **Column** as target element in case the **EAttribute** is single-valued and a **Table** if it is multi-valued. Similarly, while a containment reference or a single-valued unidirectional reference map onto a **Column**, any other kind of reference maps onto a **Table**.

Trace Table Table 10.2 summarizes these trace mappings in the first and second column, where the former states the source element and the latter the target element. The remaining objects which are created by the respective rules are not recorded in the **BXtend** trace but in the **ATL/EMFTVM** trace. The table enumerates these elements in the third column. The following paragraphs illuminate the **ATL/EMFTVM** and **QVT-O** transformations, particularly, the generated trace information.

ATL/EMFTVM We tried to reuse the **Class2Relational** transformation of the **ATL** zoo to employ an existing transformation definition. However, streamlining was hardly possible because the metamodels as well as the behavior deviate almost completely from the more fine-grained **BXtend** definition. Therefore, we (re-)implemented the transformation and used the **ATL/EMFTVM** compiler for translating and executing the transformation.

In order to create trace elements for each source element transformed by a rule, we did not employ called or lazy rules and refrained from storing parameters in helpers. Furthermore, the **ATL/EMFTVM** compiler only executes a subset of the functionality of the default **ATL** compilation engine. For instance, accessing elements stored in helpers and retrieving them in another rule is not supported. Consequently, we use only one helper without side effects to retrieve the **String** representation of an **sql** type corresponding with a given primitive **EType**. Moreover, we only accessed trace elements in **do**-blocks of the specification. This helps to assign, for instance, the table created for a bidirectional or multi-valued reference in the corresponding schema.

The generation-complete trace resulting from an **Ecore2SQL** implementation comprises mappings, which record each of the specified objects of the *to*-pattern. Table 10.2 enumerates the target elements which are mapped in the **ATL/EMFTVM** trace in the third column.

According to the rules, the transformation creates a **Schema** as well as an **EObject-Table**, a **Column** named **id**, a **PrimaryKey** and an annotation, stating the source element, as target element for a given source **EPackage**. Consequently, the **ATL/EMFTVM** trace records those elements as targets of a trace link as shown in the first row of Table 10.2. Similarly, the following rows of the table enumerate the target elements which are created for a source element of the type stated in the first column.

QVT Operational Mappings We have also re-implemented the transformation in **QVT Operational Mappings**. Similar as the proposed **ATL** class to relational database transformation example of the **ATL** zoo, the example provided in the **QVT** standard, which can be downloaded as an example **Eclipse** project, uses a simplified class and a different relational database meta-model. Therefore, we streamlined the transformation, too.

The implementation of the transformation definition can vary because different language constructs can be employed to achieve the same goal. In addition, the type of specified rules influences the contents of the trace strongly. Ex. 10.2.4 demonstrates how the rule specification can vary:

Table 10.2: Trace information for the Ecore2SQL transformation persisted by the execution engines for the languages BXtend, ATL/EMFTVM and QVT-O.

Source Element	BXtend	ATL/EMFTVM and QVT-O (created target elements)	QVT-O (context elements)
EPackage	Schema	Schema, object-Table, id-Column, PrimaryKey, package-Annotation	
EClass	Table	Table, id-Column, PrimaryKey, object-Column, ForeignKey(obj.Table), ForeignKey(loc. inheritance), two delete-Events, three Annotations	object-Table, Schema
EAttribute, single-valued	Column	Column, two Annotations	class-Table
EAttribute, multi-valued	Table	Table, id-Column, value-Column, ForeignKey, delete-Event, two Annotations	class-Table
EReference, containment	Column	Column, ForeignKey, six Annotations, delete-Event	–
EReference, single-valued, unidirectional	Column	Column, six Annotations, ForeignKey, delete-Event	–
EReference multi-valued, unidirectional	Table	Table, id-Column, reference-Column, three Annotations, two ForeignKeys, two delete-Events	–
EReference, bidirectional	Table	Table, source-Column, target-Column, four Annotations, two ForeignKeys, two delete-Events	–

Example 10.2.4: QVT Operational Mappings

Fig. 10.2.7 illustrates two QVT-O entry point rules as well as two kinds of specifying mappings in this language which are excerpts of our entire definition. On the top, in Lines 37-43 the `main` method controls the order of the execution. Since the presence of classifiers is indispensable for transforming references, the first set of rules transforms the root objects, the `EPackages`, into a `Schema` which invoke the transformations of classifiers and attributes. The second set of rules mentioned in the `main` method, transforms the specific types of references.

The mapping `mode12RDBModel`, opened in Line 51, creates a schema, an object table comprising an id-column, a primary key which uses the id-column, as well as an sql annotation which states that the schema was created for a `Ecore package`. While the version depicted on the left side mentions only the schema as the primarily created object by the rule in Line 51, the right side mentions all objects that are created by the rule in the header (Lines 52-56). Accordingly, the left version of the rule creates the objects aside the schema, as *variables* and integrates them in the corresponding container (e.g., the id-column in the created object table) by using corresponding references. Conversely, in the right version the created objects represent *OUT*-parameters. The rule assigns values to their structural features based on the parameter names.

The traces which are generated for both rule types diverge with respect to the listed target elements (represented as *OUT* parameters). The left version incorporates incomplete trace information, which maps a source object of type `EPackage` onto a target object of type `Schema`. The trace which is created for the right version records the objects created as `idColumn`, the `primKey` and the `annotation` as additional target elements.

Furthermore, the mapping-rules `class2Table` and `class2Table_inheritance` receive the object table and the schema as context parameters.

According to the example, a QVT-O trace represents context information as IN parameters and maintains a list of explicitly declared OUT parameters. The trace converter for QVT-O traces, `QVTOConverter` (c.f., Sec. 9.3.1), which is contributed as part of this thesis, records these IN

incomplete-trace-generating mapping	complete-trace-generating mapping
<pre> 34 /* execution order 35 * 36 */ 37 main() { 38 classDiag.rootObjects()[ecore::EPackage]->map model2RDBModel(); 39 classDiag.objectsOfType(ecore::EReference)->sortedBy(name) -> map containment 40 classDiag.objectsOfType(ecore::EReference)->sortedBy(name) -> map reference2 41 classDiag.objectsOfType(ecore::EReference)->sortedBy(name) -> map reference2col 42 classDiag.objectsOfType(ecore::EReference)->sortedBy(name) -> map reference2 43 } 44 45 /* 46 * mapping rules 47 */ 48 /** 49 * root: EPackage to Schema 50 */ 51 mapping ecore::EPackage::model2RDBModel() : sql::Schema 52 { 53 name := name; 54 55 var objTable := object sql::Table{ 56 name := 'EObject'; 57 owningSchema := result; 58 }; 59 var idColumn := object sql::Column{ 60 name := 'id'; 61 type := 'int'; 62 properties := Set(sql::Property::NotNull, sql::Property::AutoIncrement); 63 }; 64 var primaryKey := object sql::PrimaryKey{ 65 owningTable := objTable; 66 column := idColumn; 67 }; 68 var annotation := object Annotation{ 69 name := 'package'; 70 owningModelElement := result; 71 }; 72 73 // initiate transformation of classifiers 74 var remain := self.eClassifiers->sortedBy(name).map class2table(objTable, r 75 var remain2 := self.eClassifiers->sortedBy(name).map class2table_inheritanc 76 } </pre> <p style="text-align: center;">traced target elements</p> <p style="text-align: center;">non-traced target elements</p>	<pre> 34 /* execution order 35 * 36 */ 37 main() { 38 classDiag.rootObjects()[ecore::EPackage]->map model2RDBModel(); 39 classDiag.objectsOfType(ecore::EReference)->sortedBy(name) -> map containment() 40 classDiag.objectsOfType(ecore::EReference)->sortedBy(name) -> map reference2col 41 classDiag.objectsOfType(ecore::EReference)->sortedBy(name) -> map reference2col 42 classDiag.objectsOfType(ecore::EReference)->sortedBy(name) -> map reference2col 43 } 44 45 /* 46 * mapping rules 47 */ 48 /** 49 * root: EPackage to Schema 50 */ 51 mapping ecore::EPackage::model2RDBModel() : 52 s:sql::Schema, 53 objTable: sql::Table, 54 idColumn: sql::Column, 55 primaryKey: sql::PrimaryKey, 56 annotation: sql::Annotation 57 { 58 s.name := self.name; 59 objTable.name := 'EObject'; 60 objTable.owningSchema:=s; 61 62 idColumn.name := 'id'; 63 idColumn.type := 'int'; 64 idColumn.owningTable := objTable; 65 idColumn.properties := Set(sql::Property::NotNull, sql::Property::AutoIncrement); 66 primaryKey.owningTable := objTable; 67 primaryKey.column := idColumn; 68 69 annotation.annotation := 'package'; 70 annotation.owningModelElement := s; 71 72 // initiate transformation of classifiers 73 var remain := self.eClassifiers->sortedBy(name).map class2table(objTable, s); 74 var remain2 := self.eClassifiers->sortedBy(name).map class2table_inheritanc 75 } </pre> <p style="text-align: center;">traced context elements</p>

Figure 10.2.7: QVT-O mapping rule producing incomplete vs. (generation-) complete traces.

parameters as context elements of a trace step and the OUT parameters as target elements.

This example demonstrates that the granularity of a QVT-O trace depends on the way mappings are defined in the specification. If the definition comprises rules which only mention a single target element and create additional elements as variables, the trace granularity will be *incomplete*. For that reason, our transformation definition employs mapping-rules which state each element that is created explicitly as target element of the mapping.

Additionally, input parameters serve as context information in the mapping-rules creating tables for classes and for attributes. For instance, the object table serves to create foreign keys for an `EClass` whereas for an `EReference`, the transformation is initiated by the `main` method and, thus, cannot receive the containing class as well as the referenced class as context elements. The corresponding mapping rule retrieves the latter two elements by accessing the internally maintained trace. The persisted trace does not persist the information of additional trace lookup performed inside the mapping rule.

In summary, the QVT-O trace extends the ATL/EMFTVM trace information with the more complete information about context elements. Table 10.2 shows the additional elements stored in the QVT-O trace in its right-most column. Concretely, the QVT-O trace records the object table and the schema for each transformed class as context elements. In the same way, it lists the table which was created for the containing class as the context element of an `EAttribute`. Since the QVT-O target elements are the same as in the ATL/EMFTVM trace, these elements are not mentioned again in the table.

II Ecore to UML Class Models

Besides the Ecore2SQL transformations, we employ an Ecore to UML class model transformation [GSW17] to generate the source model for a consecutive reused UML to Java transformation. The Ecore to UML class model transformation is specified in the language ATL and compiled and executed using the ATL/EMFTVM. This allows to persist generation-complete trace information about corresponding elements. Before diving into the contents of the transformation rules and the resulting trace, the following paragraphs shed light on elements of the UML metamodel which are used to build class models (i.e., one out of the 14 UML model types).

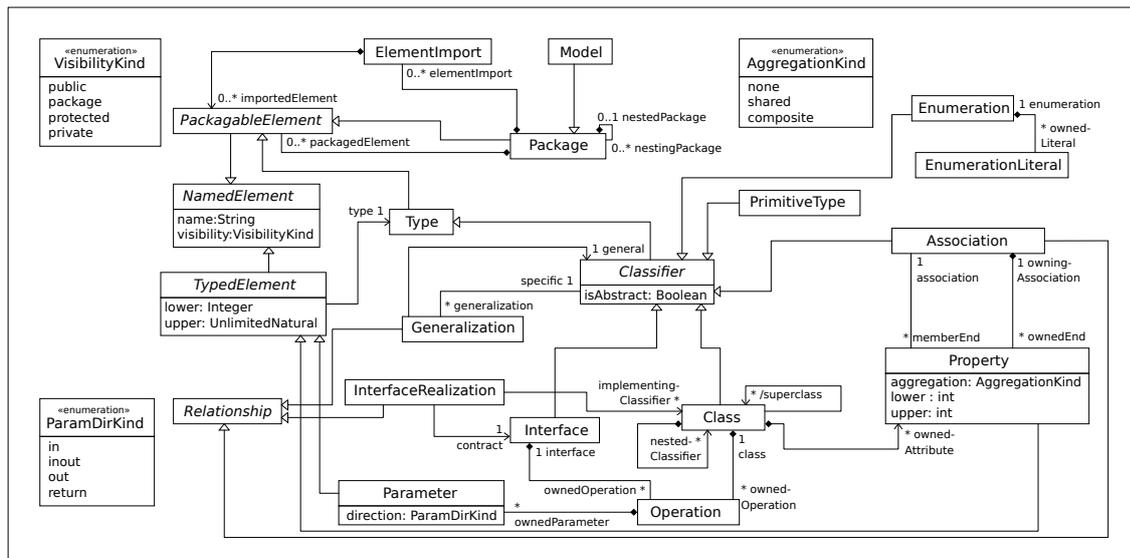


Figure 10.2.8: Simplified UML metamodel including to represent class models.

UML Class Metamodel Fig. 10.2.8 presents key elements of the UML 2.5 metamodel, which are required to represent a class model. The figure focuses on relevant elements which are created in the Ecore2UML transformation and similarly should be transformed into a Java representation. We refrain from depicting all abstract classes, such as `DirectedRelationship` or `EncapsulatedClassifier`. While those classes serve to organize and add (mostly derived) references between meta classes, they do not affect the transformation and hamper the readability of the figure. Furthermore, it must be noted that the metamodel assumes the specific characteristics to call multi-valued references with a name in singular form.

The `Model` forms the root of UML class models and is a special kind of a `Package`. Packages allow to further organize the contents of the models in different nesting levels. A package can import elements, which may be, for instance, primitive Java types but simultaneously can also comprise the primitive types of its own in form of `packageElements`. Besides primitive types, classifiers represent the central entities of a class model. The UML metamodel distinguishes – besides others, the following classifiers, which we employ in our transformations: `Class`, `Interface`, `Enumeration` and `Association`. `Relationships` express dependencies between two or multiple classifiers, such as inheritance or associations.

In more detail, in the first place, each element in the class metamodel is designated with a name and possesses a visibility which is either `public`, `package`, `protected` or `private`. Per default each element is publicly accessible. A `PackageableElement` forms either part of an element import (from another model) or of a `Package`. The latter can build different nesting levels and embraces mostly types in form of classifiers. A classifier can be abstract whereas the model distinguishes interfaces explicitly from classes.

Different kinds of inheritance relationships exist between classes and interfaces. Since UML supports multiple inheritance, a classifier can store several generalizations which record the `general` classifier from which the `specific` classifier inherits. The specific classifier in turn records the generalization object which summarizes this information. For example, the class `Domesticated` stores a generalization object which references the class `Animal` as its `general` class. Moreover, a `Class` may implement an `Interface`. `InterfaceRealizations` document this dependency by holding the `contract` (i.e., the interface) as well as the classifiers⁴. Finally, classes explicitly `derive` the list of super classes from the generalization objects and may comprise further classes, recorded as `nestedClassifier`.

Besides inheritance relationships, an `Association` describes a relationship between two or multiple

⁴ The figure simplifies the `implementingClassifier` reference of the `InterfaceRealization` which originally references the abstract class `BehavioredClassifier`. `UseCases`, `Collaborations` and `Actors` specialize the `BehavioredClassifier`, too, but are not required to represent a class model.

classifiers in terms of `Propert[y]ies`. The association stores each referenced property (at least two), which is typed by the classifier, as a `memberEnd`. A composite relationship (composition or aggregation) is modeled by setting a corresponding aggregation kind to the respective property. Furthermore, the UML metamodel allows to define the direction and the navigability, the owner of the association ends, which can be the classifier or the association itself, as well as the names and the multiplicity of the ends. If further information should be associated with an association, it may be modeled in form of an `AssociationClass` which combines the semantics of both, the UML `Association` and the UML `Class`.

Transformation Assumptions Firstly, the definition assumes that the Ecore class model comprises only one package which serves as root of the model. Secondly, the rules assume only single inheritance between classes. In addition, since Ecore model elements do not possess a visibility, we consider the default UML visibility for all its elements, namely `public`, such that the transformation definition does not have to set it explicitly. As the Ecore metamodel does not allow for nesting classifiers and explicit interface realization, likewise the transformation does not construct nesting levels between classifiers and does not create interface realization objects, respectively.

Transformation Rules The transformation establishes the following mappings which add up to 15 matched ATL rules and one helper which returns a primitive types for parameters and attributes:

- `EPackage`: a UML `Model`, five `PrimitiveType` (`Boolean`, `Integer`, `String`, `Real`, `UnlimitedNatural`) and a root `Package`
- `EClass` (no interface, empty `eSuperTypes`) : `Class`
- `EClass` (no interface, with `eSuperTypes`): `Class`, `Generalization`
- `EClass` (interface, empty `eSuperTypes`): `Interface`
- `EEnum` : `Enumeration`
- `EEnumLiteral` : `EnumerationLiteral`

- `EAttribute` (primitive type): `Property`
- `EReference` (unidirectional): `Property` (owned by `Class`), `Association`, `Property` (owned by `Association`)
- `EReference` (bidirectional): two `Propert[y]ies` (owned by `Class`), `Association`, empty `Property` (owned by `Association`)
- `EOperation` (no return): `Operation`
- `EOperation` (with return): `Operation`, `return-Parameter`
- `EParameter`: `Parameter`

For the root `Package`, the transformation creates a UML `Model` as root and an equally named package which comprises the classifiers as well as the additionally created UML primitive types. As primitive types, the types `Boolean`, `String`, `Integer`, `Real` and `UnlimitedNatural` are present in the UML model per default. Accordingly, a helper mechanism maps the Ecore primitive types in the following way:

- `EBoolean` becomes `Boolean`
- `EString` and `EChar` become `String`,
- `EInt` and `EByte` become `Integer`,
- `EDouble` and `EFloat` become `Real` and
- `ELong` become `UnlimitedNatural`.

Transforming EClassifier The transformation definition differentiates `EClasses` which are *no* interfaces and those which are interfaces and `Enumeration` data types. It creates an interface for each class where the meta attribute `interface` is set `true`. Aside from interfaces, an `EClass` which is no interface is turned into a UML `Class` which may be `abstract` if the respective meta attribute is set `true` in the source class. If the `EClass` inherits from another class, the transformation creates an additional object of type `Generalization` by employing a different rule

which resolves the reference to the super class in its do-block. For **EEnums** a respective rule creates a corresponding **Enumeration** and literals. The rules add the classifiers to the package created for the root **EPackage** by employing the trace information maintained during the transformation.

Example 10.2.5: Ecore2UML Subclass Transformation Rule

Listing 10.2.2 presents the ATL matched rule which creates a UML Class and Generalization for a given EClass. This rule serves as example how the overall transformation proceeds, particularly, how the specification integrates the objects created for structural features into a classifier.

*In the first place, the guard in Line 3 of this rule ensures that only classes which inherit from another class match the rule. The rule does not only create a UML class as the primary target element, **tar**, but also the generalization object. The latter references the super class as **general** classifier and the created target class as the **specific** classifier. Line 10 inserts the generalization as the only one of the created target class. This action ensures also, that the derived reference **superClass** of the created UML class is set.*

*Finally, the do-block inserts the elements that are created for the **eStructuralFeatures** and **eOperations** of the input class. These two statements resolve trace information and return the corresponding elements. Line 17 accesses the trace element which is created for the transformation of **referecens**. By stating the name of the target element in the rule, here **trg**, it is possible to extract one specific target element out of potentially multiple ones created by a rule. The second line adds all operations which are created for the list of **eOperations** stored in the matched **EClass**.*

Please note: In case a rule creates multiple objects (e.g., such as the presented rule does), the operation **resolve()** returns the first created object in the matching rule. For instance, for the operations, the created UML operation which is stated as single or first object in the respective rule is integrated. In contrast, if multiple elements and another element than the first one that is created in the rule should be resolved by accessing the trace, the name of the target object in the rule, such as **gen** in the presented rule, must be stated in the operation **resolve()**. Since the rules which create the property which should be added to the list of **ownedAttributes** may state the association as first element, the resolution explicitly mentions the name of the created element that should be used. Consequently, it is indispensable that the target name given to the operation **resolve()** matches one object name defined in the to-pattern. If they do not match, a runtime error will occur.

Transforming EStructuralFeature To transform structural features, the definition differentiates attributes from references. It creates a **Property** for each **EAttribute**. Thereby, the corresponding rule retrieves the type for the created property either by employing the aforementioned helper, in case it is not an enumeration type, or by employing the trace to resolve the element that was created for the enumeration datatype. In addition, the rule copies the values of the meta attributes of the **EAttribute** to the corresponding meta attributes of the **Property**, such as the attributes **derived** to **isDerived** or **upperBound** to **upper**. Associations are not considered in this rule because they have to be created for references only:

The transformation definition employs distinct rules for unidirectional and bidirectional references. In contrast to the Ecore2SQL transformation, the multiplicity does not influence the distinction because it can be mapped onto equivalent meta attributes of the UML properties.

For a unidirectional reference, the transformation creates a **Property** which is stored in the containing class as well as an **Association** and a second **Property**. While the first property, which represents the values of the reference, is stored in the containing class, the second property is stored in the association as its **ownedMemberEnd**. Besides the owned member, the created association further records both properties in the separate meta reference **memberEnd**.

For bidirectional references, the specification employs two rules. One rule serves to create the association and the **Property** stored in the containing class. The second rule creates a property representing the opposite end. This allows to insert the created properties in the correct class

Listing 10.2.2: Ecore2UML ATL/EMFTVM rule for transforming subclasses.

```

1 rule EClass2Class_super {
2   from
3     src: Ecore!EClass
4     (src.interface = false and src.eSuperTypes.size() <> 0) -- subclass
5   to
6     tar: UML!Class (
7       name <- src.name,
8       isAbstract <- src.abstract,
9       package <- thisModule.resolveTemp(src.ePackage, 'pkg')
10      generalization <- Sequence{gen} -- set created generalization
11    ),
12    gen: UML!Generalization (
13      general <- src.eSuperTypes.first(),
14      specific <- tar
15    )
16  do {
17    tar.ownedAttribute <- src.eAttributes -> union(src.eReferences.resolve('tar'));
18    tar.ownedOperation <- src.eOperations;
19  }
20 }

```

inside the rule that creates the classes by resolving the created target elements by their name `tar` when using the trace information. Besides those two properties which are both stored in the respective containing classes, the rule, which creates the bidirectional association, creates another *empty* property (i.e., initialized with default values) and stores it as an `ownedMember` of the created association. This additional property is necessary due to the fact that the execution engine would not insert the created association in the package otherwise. This in turn would provoke incorrect trace information. Before propagating annotations to a Java model, we remove this additional unused property from the association in a postprocessing step

Transforming EOperation At last, the specification states two rules to transform operations. One rule creates only a UML `Operation` whereas the second rule creates a *return parameter* in addition in case the `eType` of the given `EOperation` is defined. This distinction helps to create two different types of trace elements and to insert the returned parameter as target element explicitly if it is present. Furthermore, another rule converts the operations' `EParameters` into UML parameters of the `DirectionKind in`.

Transformation Result Fig. 10.2.9 displays the resulting UML class model for the DBC product line in the Ecore tree editor. The figure highlights the following elements: An *inheritance relationship* (i.e., a `Generalization` in UML) exists between the classes `Domesticated` and `Animal`. The subclass stores the generalization object, in this example the class `Domesticated`. Similar elements are also present in the classes `Wildlife`, `Book`, `Song`, and `Movie` mentioning the respective super classes.

Furthermore, Fig. 10.2.9 stresses the elements created for uni- and bidirectional references. While the database contains elements in a unidirectional way (i.e., a unidirectional containment reference), as highlighted in the figure, a pet knows its owner and vice versa (bidirectional crossreference). The transformation creates one property for the unidirectional reference and stores it in the `Database` whereas the opposite end (which is not modeled in Ecore) resides as property in the additionally created `Association`. The association is named by the `EClass` owning the reference followed by the reference name and records both created properties as `memberEnds`. Conversely, the bidirectional reference between persons and domesticated animals is turned into an `Association` as well as two properties which are both stored in the respective class. The association is named by the name of the two references, in the example: `pets_To_owner`, and encompasses both properties as member ends.

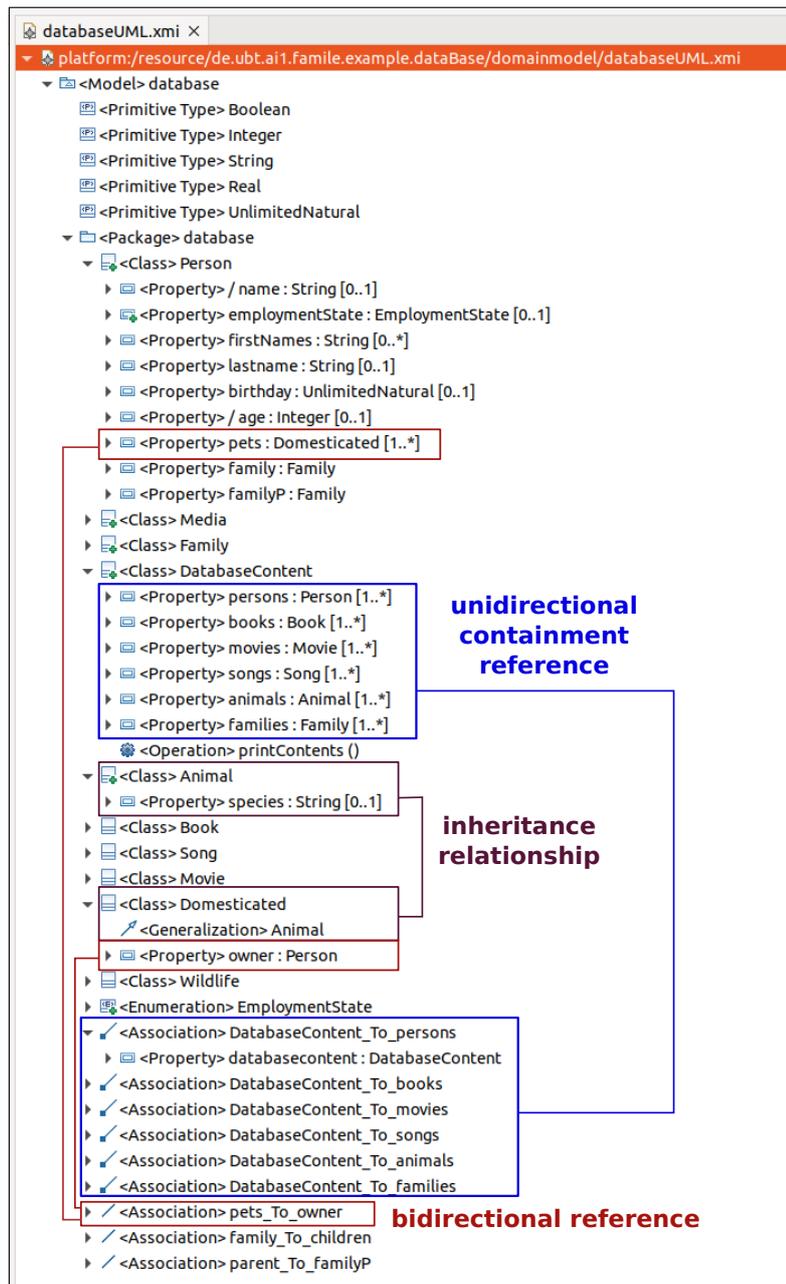


Figure 10.2.9: DBC UML class model resulting from Ecore2UML transformation.

III UML Class to Java Models

As a third kind of transformation, the UML class model to Java MoDisco transformation (denoted **UML2Java** in the sequel) serves to examine a real-world use case which is necessary to synchronize class models and derived Java source code. Besides its main purpose, the UML2Java transformation allows to assess the propagation in the large. The target of the transformation represents Java source code in form of a model. The MoDisco framework [Bru+10] builds on this model and offers a M2T generator which transforms the model into the corresponding Java source code, on the one hand. On the other hand, the framework offers *discoverers* which allow to parse the source code of a Java project and transform it into an instance of its Java metamodel by the means of a T2M transformation.

Notice that – like for the similar tool, JaMoPP [Hei+09] which allows to *parse* Java source code into a model and *print* Java source code based on such model – the active development of Java

MoDisco discontinued and supports Java source code conforming up to version 1.6 (JaMoPP represents Java 1.5). However, since we focus on developing and transforming the structure (not the behavior) of Java programs, the MoDisco framework suffices for the purpose of examining a real-world forward transformation.

After introducing the Java MoDisco metamodel, the following paragraphs disseminate two versions of the transformation: On the one hand, the base transformation rebuilds the class structure 1:1 as far as possible. On the other hand, we extended this transformation to reflect the EMF code generation with regards to separating the implementation from its interface and create implementation packages and an interface as well as class for each UML class.

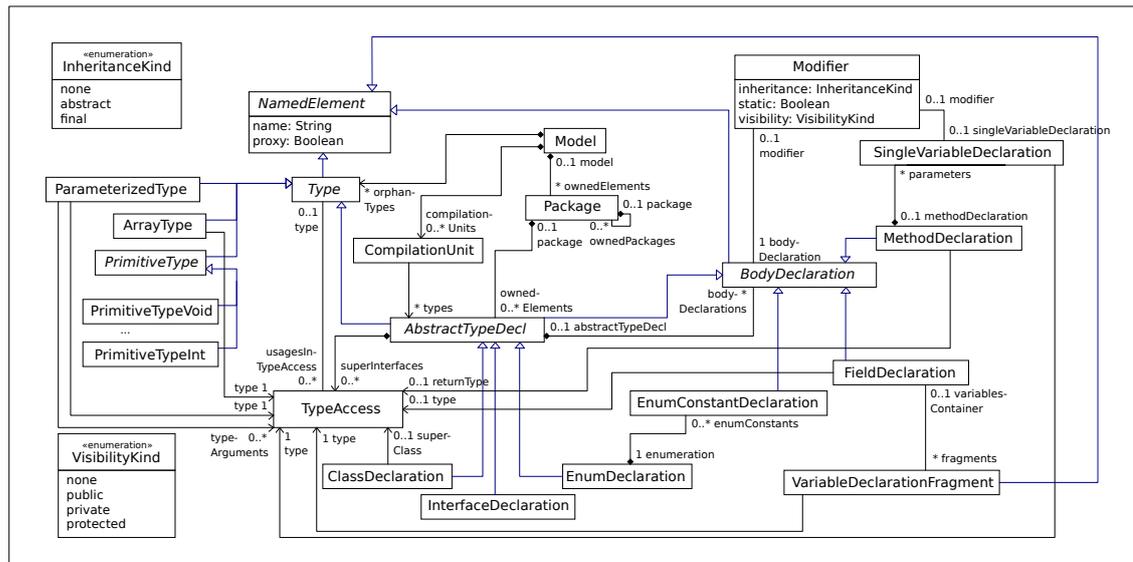


Figure 10.2.10: Simplified Java MoDisco metamodel.

Java MoDisco Metamodel The Java MoDisco metamodel serves to represent Java source code complying up to Java 1.6 in all its facets. Accordingly, it does not only represent the static structure of the source code completely but also the behavioral parts, namely method bodies and value assignments.

Fig. 10.2.10 summarizes elements of the Java MoDisco metamodel which are relevant to represent the static structure of Java source code on which we focus in the following. Similar to the UML metamodel, a `Model` forms the root of the Java MoDisco metamodel. In contrast to the UML metamodel, the `Model` meta class is no specialization but a class which neither inherits from another class nor is specialized by another class. The model comprises not only `Packages`, referenced as `ownedElements`, but also `orphanTypes` and `compilationUnits`. While the orphan types comprise any type, such as primitive types, type declarations or parameterized types, the compilation units represent the units which can contain one or several type declarations.

A `CompilationUnit` does not contain elements but references a *package* as well as the *types* which are integrated in that unit. Since the reference between the `Model` and the `CompilationUnit` is unidirectional, for storing a created compilation unit the Java model (as context element) needs to be present.

The `TypeAccess` represents another distinguishing design decision of the metamodel. Instead of referencing a specific type, such as the type of a field declaration, directly, an instance of the `TypeAccess` is referenced by the object to record the dependency explicitly in an object. Accordingly, the `TypeAccess` object references the actual `type`.

`Packages` may contain `nestedPackages` and either record its containing package or model. Furthermore, a package comprises `ownedElements`, too, which are of type `AbstractTypeDeclaration`. An `AbstractTypeDeclaration` is a specific `Type` which is contained in a package and encom-

passes `bodyDeclarations`. The abstract type declaration may realize several `superInterfaces` whereas a `ClassDeclaration` may encompass exactly one `superClass` which are accessed via respective instances of the class `TypeAccess`. Besides interfaces and classes an `EnumerationDeclaration` serves as third specialization of the abstract type declaration and comprises enumeration constant declarations.

Each abstract type declaration encompasses `bodyDeclarations` which can be another abstract type declaration (e.g., a nested class or an inline enumeration declaration) or a field or a method declaration. A `Modifier` refines the implementation information of the body declaration, stating its visibility, whether it is static and if it is inheritance state (i.e., whether it is final or abstract). A field declaration comprises `fragments` of type `VariableDeclarationFragment` which define the `name` of the field and store a type referenced via a `TypeAccess`. Similarly, a method declaration records a `returnType` and can comprise a number of parameters. In contrast to a field declaration the `name` of the method is part of the `MethodDeclaration` object and in contrast to UML, the return type is defined at each state. Due to explicitly modelling `void` as a primitive type, it can be used as return type in case the method does not return anything.

Transformation Rules We employ a transformation to convert a UML class model into a Java representation specified in BXtend [BG16b]. The specification allows to transform the static structure but does not add behavior in form of representing method bodies except for default return values. Furthermore, the specification employs a 1:1 mapping of classes, meaning that no additional interface declaration or any other package is created for a given class. Attributes are represented by accessor methods and associations are summarized in a class declaration which is able to mitigate the information loss in case a unidirectional association would have to be transformed.

Entry Point: Models and PrimitiveTypes In more detail, the BXtend specification encompasses *twelve* classes that allow to transform the main elements of UML class models. The transformation starts with transforming the model, primitive types and packages subsequently and continues to transform classifiers in form of enumerations, interfaces and classes and handles class inheritance thereafter. Additionally, after the classifiers are transformed associations are turned into Java class declarations. The remaining three classes comprise rules to transform properties, operations and the literals of an enumeration.

While the class and corresponding rules to transform models straightforwardly sets the names, primitive types can be either stored as packaged elements in the UML `Model` or as imported elements. Accordingly, the rule considers both possibilities. Furthermore, it creates an `ArrayType` for each primitive type in order to represent multi-valued primitive types in this way (there are no multiplicities in Java) as well as the primitive type `void` and the packages `java.lang` and `java.util` which comprise the `String` class and the `List` interface, respectively. The primitive type of the `ArrayType` is wrapped by the means of a `TypeAccess` which is also created by the rule. The created primitive and array types are added to the Java `Model` as `orphanTypes`. As a consequence, the correspondence element of the trace records only the source primitive type as source and the respective Java primitive type as target element. The trace records neither the additionally created type access and array type nor the primitive type `void` which does not reflect any specific UML type.

Transforming Package The transformation of packages consists of two steps. The first step creates a Java package for each UML package as well as a corresponding trace element and assigns the name. The second step constructs the package hierarchy by accessing the trace information which either adds the package as `ownedElement[s]` of the Java `Model` or as a `subPackage[s]` of another model. If the UML `Model` does not comprise a package, an artificial root package will be created in the target model.

Transforming Classifiers The transformation of each classifier, enumeration, interface and class, creates the respective type declaration as well as a parameterized type and a modifier for

the type declaration. On the one hand, again, the parameterized type serves to collect multiple instances of the type declaration. In this case, the parameterized type employs the interface `java.util.List` and the type declaration as its actual parameter to represent the container of the multiple values. Its realization requires to create two type accesses, one for the list interface and one for the type declaration, where the latter serves as sole `typeArgument[s]`. On the other hand, the modifier encapsulates the inheritance properties and visibility of the type declaration. The visibility can be mapped almost 1:1, except for the UML visibility `package`, which corresponds with the Java visibility `none`. Similarly, if the UML type declaration is an interface or a class, the fields `abstract` and `final` can be mapped accordingly onto the inheritance kind of the modifier. Additionally, a compilation unit is created for each type declaration which is referenced as `originalCompilationUnit` by the created type declaration. While the type declarations are stored in the corresponding package, the parameterized type and the compilation unit are stored in the Java model.

Moreover, the rule transforming classes regards the possibility to *nest classifiers* inside a type declaration. It employs a two-fold mechanism, similar as for the packages: First, it creates a class declaration for each UML class and thereafter, builds the nesting levels.

Inheritance relationships are handled after having created the type declarations. The transformation assumes only single inheritance and assigns the `superClass` via a `TypeAccess` accordingly. If a `Generalization` exists between two interfaces, `superInterfaces` are assigned instead.

Transforming Associations For transforming an association, different possibilities exist which may also regard the fact that unidirectional associations may provoke information loss if transformed in two directions. For that reason, the transformation creates a separate public class declaration and corresponding compilation unit for each association, regardless of its direction, navigability and multiplicity. The association class is stored in the package created for the UML package storing the association. The next step which transforms every kind of UML `Property` creates its field declarations.

Transforming Propert[y]ies The rule that transforms a UML property always creates a `FieldDeclaration` in the first place and sets the name by the means of an additionally created `VariableDeclarationFragment` in the second place. Setting the owner of the field declaration depends on the ownership of the UML property. If the end is owned by an association, the field declaration will be integrated as `ownedElement` of the corresponding association class created in the previous transformation step. In this case, further information about the owner, navigability and aggregation kind is added to the field declaration in form of additionally created Java-Doc comments. If the UML property does not correspond with an association (i.e., it is not recorded as a `memberEnd`), it will be integrated in the type declaration created for the corresponding UML container. Finally, types are set according to the type and multiplicity of the UML property. Due to the complexity of the transformation, we refrained from guarding read- and write-access to the field declarations by the means of get- and set-methods.

Transforming Operations The final part of the transformation definition converts UML operations into Java `MethodDeclarations`. First of all, this requires to assign the name but also to map the visibility in the same it is done for type declarations as well as the inheritance kind. Additionally, operations may be static which is set similarly. The transformation of the return type, either retrieves the corresponding primitive or complex type according to the respective multiplicity and references it via a type access object or the type access references the primitive type void in case no return value is set. In the original transformation an artificial parameter was included in the UML operation if no type was specified (i.e., it was `null`) in order to create a correspondence element in the trace. In this transformation we decided to not create a trace element in this case because it may provoke inconsistencies in incremental transformations. Instead, the rules handle the creation and deletion of type accesses explicitly for each method. As a consequence, however, the trace does not record a correspondence and thus, does not provide information which annotation to assign to the type accesses in the Java model.

Transforming Parameters The remaining *input* parameters are transformed into a `SingleVariableDeclaration`. In contrast to a field declaration, the name is set directly to the single variable declaration. Similar to field declaration, the type is referenced via a type access object which is created in the rule, additionally. The referenced type corresponds with the multiplicity and is accordingly either a primitive type or an array thereof or a type declaration or a list thereof.

Trace Mappings In summary, although several target elements are created in the majority of rules, the incomplete BXTend trace records the following 1:1 mappings only:

- `Model` \Leftrightarrow `Model`
- `PrimitiveType` \Leftrightarrow `PrimitiveType(...)`
- `Package` \Leftrightarrow `Package`
- `Enumeration` \Leftrightarrow `EnumDeclaration`
- `EnumerationLiteral` \Leftrightarrow `EnumConstantDeclaration`
- `Interface` \Leftrightarrow `InterfaceDeclaration`
- `Class` \Leftrightarrow `ClassDeclaration`
- `Association` \Leftrightarrow `ClassDeclaration`
- `Property` \Leftrightarrow `FieldDeclaration`
- (input) `Parameter` \Leftrightarrow `SingleVariableDeclaration`

As a consequence, the majority of elements in the target model are not recorded in the trace and will receive annotations by one of the completion strategies.

Advanced Version In order to examine the influence of modeling hierarchies in completing annotations due to the presence of incomplete traces, we modified the UML2Java forward transformation. The *advanced* version creates an implementation package in packages corresponding with an UML package which stores a UML class as well as corresponding interface declarations for each class declaration. The advanced transformation, **UML2JavaAdv**, behaves as follows:

For each UML class the transformation does not only create a class declaration, compilation unit and parameterized type are created but also an interface declaration which the class declaration realizes and a corresponding compilation unit. The rule further only creates only a single parameterized type which employs the interface as parameter of the Java list.

While the interface is stored in the package, which was created for the UML package containing the class to be transformed, the generated class declaration is stored in an implementation package, named 'impl'. The package `impl` is created once by the rule that transforms UML packages whenever a UML class exists in the package. Furthermore, the rule adds both created compilation units (one for the interface and one for the class declaration each) and the parameterized type to the model as `compilationUnits` and `orphanTypes`, respectively. Moreover, the rule adds the interface to the set of `superInterfaces` implemented by the created class declaration but refrains from adding the method heads present in the class declarations.

Apart from that behavior, the rule which transforms UML classes creates parameterized types and compilations units similar as in the base version. Creating the interfaces and the implementation package constitutes the main difference to the basic UML2Java transformation.

10.2.4 Model-To-Text Transformations

For evaluating the correctness of the aspect-oriented propagation, we further employ two M2T transformations specified in Xpand: While the Ecore to Java transformation reflects the EMF code generation inasmuch as it also creates interfaces and classes for an `EClass`, the MoDisco to Java transformation re-implements the basic behavior of the MoDisco source code generator. The original transformations both resulted from student projects supervised by the author of this thesis. The following two parts first explain the text generation for Ecore models followed by the generation for Java MoDisco models.

I Ecore to Java Source Code

At first, we consider an M2T transformation definition which converts an instance of the Ecore metamodel into Java source code. This **Ecore2Java** definition partially rebuilds basic concepts of the Ecore code generation. In particular, for each **EClass** which is no interface and not abstract, the transformation creates not only a Java class but also an interface which the created class implements.

Entry Point: EPackage The Ecore package matches the entry point rule of the transformation and the corresponding expand-directive initiates text productions for each contained classifier. The rule begins with the transformation of **EClasses** followed by the creation of **EEnums**. While the creation of an enumeration focuses on stating all literals literally (no further functionality), the transformation of classes assumes a higher degree of complexity:

Transforming EClass In the first place, the block handling **EClasses** creates a file which represents the interface declaration. This is created in any case regardless whether the meta attribute **interface** of the **EClass** is set true. Before opening the interface declaration with the text fragment `public interface <name>` (the name is dynamically replaced), the package declaration is stated as first line, which is derived from the container of the **EClass**. In the following lines, type import statements are printed which are computed from iterating the types used in the structural features and operations of the **EClass**. Even though the resulting text production is initiated by an expand-directive and, therefore, the annotations of the source element will be mapped onto it, it is a non-local rule because the necessity of the import may depend on the presence of an operation or association which includes the typed element for which the import is created.

In case the meta attribute **interface** of the given **EClass** is set **false**, the rule initiates the creation of a second file which stores the class declaration stating the same import-directives as the interface. Listing 8.3.1 summarizes its behavior: The transformation adds the created file to an package `impl` stored in the directory created for the containing package. Besides stating the package declaration and the import statements, the file opens the declaration of the class by stating its name combined with the suffix `Impl` (i.e., `<name>Impl`). Furthermore, the class implements the interface, the declaration of which was created before, by writing the word **implements** after the class name. Additionally, if a super type is present, the word **extends** is written right after the name of the class declaration (before the implements statement).

Before invoking the rule for transforming structural features and operations, the rule, creating the class declaration, opens a protected region, to allow for integrating and preserving additional source code. Then, it handles structural features followed by generating text for its operations.

Transforming EStructuralFeature To transform the structural features of the **EClass**, a rule creates access-methods (**get-** and **set-**). Furthermore, if an attribute is not derived, it will integrate a protected field declaration in the created class. While the created interface states only the heads of the methods, the rule which creates the class declaration invokes another rule that creates the entire body of the access methods.

Transforming EOperation Similarly, the transformation of **EOperations** creates only the head of the methods in the interface. Conversely, in the class declaration not only the head of the method is generated but also its body which may comprise a default return type (if the return type of the **EOperation** is defined). In addition, the corresponding rule creates another protected region as content of the method body. This region also encompasses the **return** statement in order to be able to replace the default with a custom return value.

II Java MoDisco Model to Java Source Code

As a second transformation, we employ a code generation prototype which creates Java source code from a given Java MoDisco model. Again, the focus of the rules lies in transforming the structural representation of the software but not its behavior. Furthermore, the purpose of using

this transformation lies in demonstrating that the same advice is able to transfer annotations between instances of different metamodels and not to fully rebuild the MoDisco M2T generation.

Entry Point: CompilationUnit As a consequence of the design of the Java MoDisco meta-model, the entry point rule does not iterate the contents of packages but of compilation units. These model elements represent exactly the contents which should be created in form of a Java file. While the Ecore2Java transformation needs to collect the element types which should be imported before opening the type declaration, the compilation units store `importedElements`. However, since this is not a derived reference, it depends on the creation mechanism of the Java model whether the compilation unit states imported elements. The UML2Java specification (Sec. 10.2.3), for instance, does not create these elements. Thus, if the compilation unit does not enumerate imported elements, no import-statements will be included in the created type declaration.

Transforming FieldDeclaration Then, the text-generating rule either opens an interface declaration or a class declaration. Enumerations and inheritance among classifiers is not supported in the rules. Inside the class declaration, MoDisco `FieldDeclarations` are converted in a corresponding Java (static) field declaration. The transformation assumes that only one variable fragment is stated per field declaration. For converting a method declaration, the visibility and type are extracted and used in the header. Furthermore, the return type as declared in the Java method declaration is stated as return value. Instead of field declarations, the transformation writes the method headers in interface declarations.

UML associations are stored in the input MoDisco model as class declarations with corresponding compilation units. Therefore, they are treated in the same way, as any other class declaration.

10.3 Results

This section presents the results of evaluating the correctness of propagating the annotations of the subject product lines (c.f., Sec. 10.2.2) after and while executing one of the transformation definitions presented in Sec. 10.2.3 and Sec. 10.2.4. In particular, the included sections bring up the resulting F2DMM models and the multi-variant source code as well as the measured error values and discuss the implied accuracy of the annotations. Besides demonstrating the correctness of annotations propagated based on generation-complete and complete traces, which result in correctly annotated models, one focus lies in examining the effect of the completion strategies used in sight of incomplete traces. The latter do not affect accuracy differently in our setup which is discussed in Sec. 10.4. Finally, the aspect-oriented propagation results show that the Xpand-based propagation achieves high accuracy despite the language's more powerful capabilities that violate the computational model in general.

Preliminaries The sequel presents the evaluation of commutativity for each pair of transformation definition and appropriate input model of the subject product line for each valid configurations in order to gain an accurate result. Accordingly, no sampling is used and each valid configuration (i.e., 234, 180 and 16560 configurations in the DBC, Graph and HAS product lines, respectively) is regarded. Furthermore, each of the input source mapping models is annotated completely with respect to the object mappings. As effect, a (user-intended) annotation is mapped onto each of the source elements recorded in traces or processed by the aspect. The models and source code directories are distinguished and referred to as *filer-transform* and *transform-filter* variants. The filter-transform variant results from applying the model filter to the annotated multi-variant source model followed by executing the single-variant transformation whereas the transform-filter variant results from executing the multi-variant transformation followed by applying the model filter to the annotated multi-variant target model, respectively.

Road Map The following sections demonstrate the results of the M2M transformations scenarios first and the results of the M2T transformations second. Thus, Sec. 10.3.1 - 10.3.4 present the results of conducting the Ecore2UML, base and advanced UML2Java and the Ecore2SQL

Table 10.3: Resulting error measurements of the multi-variant Ecore2UML transformation.

Product Line	$\#map_t$	err_{abs} [%]	err_{sev} [%]	err_{act} [%]
DBC	310	0.00	0.00	0.00
Graph	380	0.00	0.00	0.00

transformation, respectively. Conversely, Sec. 10.3.5 summarizes the results of the two M2T transformation scenarios which transform an Ecore and a Java MoDisco model into Java source code, respectively.

The description of each transformation scenario starts with an overview of the comparison results. Then, it dives into the details which present the resulting target mapping model and demonstrate the measured values afterwards.

10.3.1 Ecore2UML

The transformation of Ecore models into UML class models and propagating the annotations based on the ATL/EMFTVM trace results in an absolute error of 0%. Thus, the implemented trace propagation computes and assigns all annotations completely correctly yielding a correctly annotated domain model. The trace-based propagation algorithm together with the executed reused single-variant transformation definition creates correctly annotated models for both, the Graph and the DBC product line, despite the missing context elements in the ATL/EMFTVM trace. As a consequence, the severity and the actual error are equally of 0%, as listed in Table 10.3. This accuracy of the annotations also justifies that the resulting mapping models can serve as input to the following UML2Java transformation without adaptations.

Details Fig. 10.3.1 presents the F2MM models resulting from transforming the multi-variant Ecore DBC and Graph Ecore domain models. The left side depicts the DBC and the right side the Graph F2DMM models, respectively, which rebuild the UML class models created by the reused single-variant model transformation. The figure highlights some of the created types of UML elements with colored solid rectangles. The respective assigned annotations are stressed with circles inside the rectangles and reside behind the model element in the tree editor.

Correct Annotations Firstly, the figure indicates that each of the elements is annotated and the comparison results (not depicted in the figure) computes an absolute error of 0%. Thus, the assigned annotations are correct and satisfy commutativity to full extent. Even though not all annotations are depicted in Fig. 10.3.1 for space reasons, the evaluation results state that correct annotations are mapped onto all objects.

Annotations of Unidirectional Associations Secondly, the figure exemplifies the annotations of the associations created for uni- and bidirectional references. The associations `Database_to_families` and `Graph_to_nodes` serve as examples of unidirectional associations in the DBC and the Graph F2DMM models, respectively. Accordingly, the associations store the end from which the transformed `EReference` originates as its `ownedEnd` and the referenced class is stored as property in the class from which the association originates. For instance, the association `Database_to_families` comprises the property `database` whereas the class `DatabaseContent` stores the property `families`. The propagation algorithm mapped the annotation `Family and Relation`, which is the annotation of the source element, the `EReference families`, onto each of these elements.

Annotations of Bidirectional Associations As examples of transformed bidirectional references, the figure highlights the associations `parent_to_familyP` and `graph_to_algorithm`, additionally. While the respective opposite class stores the corresponding member end as property, another property had to be inserted into the association so that the ATL/EMFTVM engine generated the association. We remove this empty property before executing subsequent transformations. Nonetheless, the annotation of the bidirectional source reference is mapped onto all

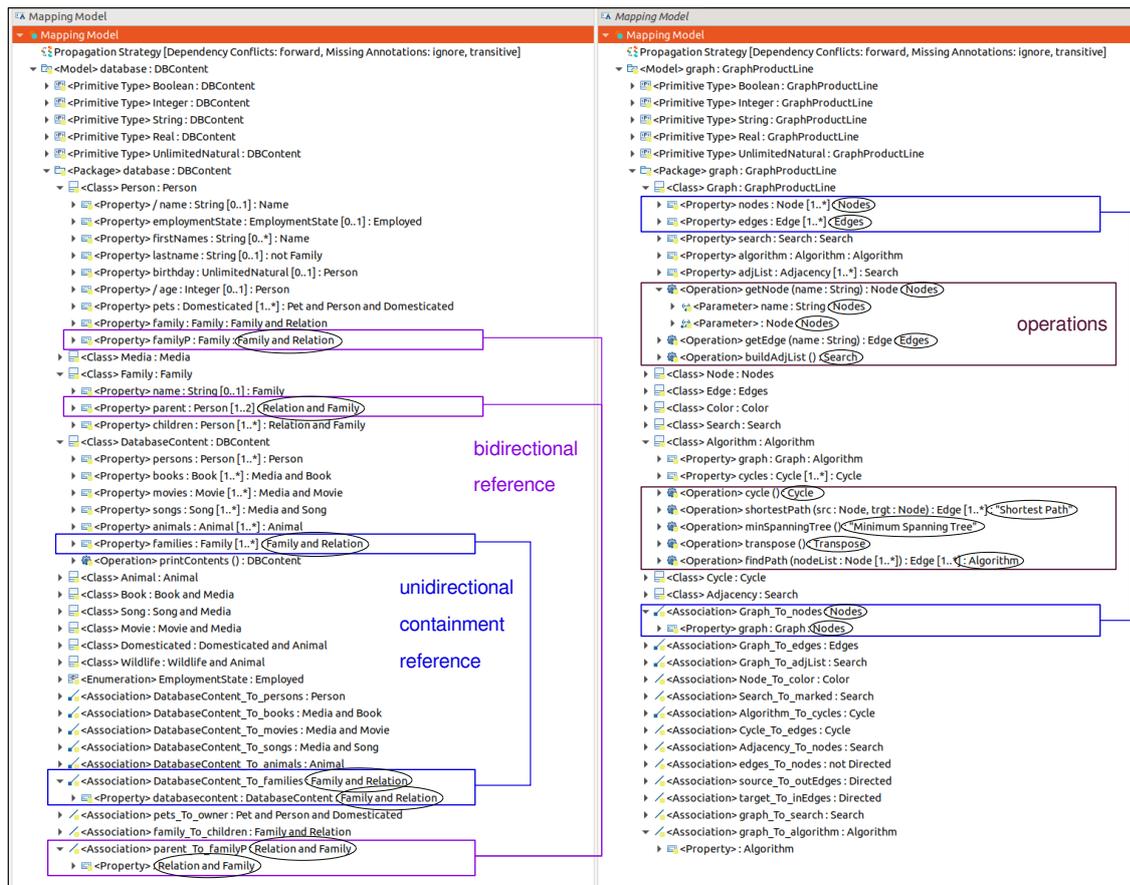


Figure 10.3.1: DBC and Graph UML F2DMM models resulting from Ecore2UML transformation.

elements created by the respective ATL rule which the trace records as target elements of this rule application.

Annotations of Further Elements Thirdly, the figure highlights further UML class model elements, such as operations and properties of primitive type. These elements carry correct annotations which correspond with the source element, the corresponding $EOperation$, too.

Correctness Despite Generation-Complete Trace All in all, the propagation based on the generation-complete ATL/EMFTVM trace produces correct annotations which result in commuting transformations. Although the ATL/EMFTVM trace is not complete, the presence of context elements would not refine the computed annotations in a way that influences the derived variants. This effect is caused by the source annotations which imply the selection of the annotations mapped onto context elements.

As an example, the creation of a class requires the package as context element and the annotations of the $EClasses$ imply the annotation of the single packages (i.e., **Graph** and **DBContent**, respectively) of both source models. The situation is the same for the remaining elements, which require the presence of context elements to be created, such as the UML operations, which require a classifier to be present. Still, in this transformation scenario and in the given source models, the annotation of the depending elements implies the annotation of the element they depend on, such that either both annotations are selected and both elements are present or both are missing.

Summary The comparison results give evidence that the annotation propagation performs 100%, correctly for the input product lines. Since context elements would not refine the assigned annotations, for the subject product line models and the transformation definitions a correct prop-

Table 10.4: Resulting mappings and error measurements of UML2Java transformations.

¹Calculated by subtracting the number of trace elements from all mappings.

UML2Java	Product Line	#target mappings	#incomplete mappings ¹	err_{abs} [%]	err_{sev} [%]	err_{act} [%]
base	DBC	688	625	100.00	20.22	4.22
	Graph	753	682	100.00	18.94	2.79
	HAS	1342	1192	100.00	14.52	6.11
advanced	DBC	788	725	100.00	23.64	5.96
	Graph	827	756	100.00	22.76	3.75
	HAS	1714	1564	100.00	14.99	7.70

agation is conducted based on the generation-complete ATL/EMFTVM trace. As a consequence, we can employ the resulting target domain and mapping models as input to the UML2Java transformation⁵.

10.3.2 Base UML2Java

The second scenario, which examines the effects of the completion strategies, transforms the created UML class models into Java MoDisco models. We conducted the UML2Java transformation on all three subject product line models. The DBC and Graph UML class models and mapping models result from executing the Ecore2UML transformation beforehand whereas the HAS domain and mapping model are replicated independently at the Chair of Software Engineering (of the University of Bayreuth) from literature [PBL05]. The UML2Java transformation is specified in BXTend which persists *incomplete trace information*. Therefore, we investigate the effects and correctness of the three completion strategies in this scenario.

Overview of Results Table 10.4 summarizes the results of executing the base and the advanced version of the UML2Java transformation. As the upper part of the table shows, transforming each of the subject product lines with the *base* transformation resulted in an *absolute error* of 100% while particularly the *actual* error remains below 10% in each subject product line. This allows for concluding that the propagation algorithm computes and assigns more than 90% of the annotations correctly. Furthermore, the results show that the three completion strategies exhibit the same accuracy in this transformation scenario. The measured error rates are equal for each strategy in each subject product line.

While the absolute error expresses that in each configuration at least one mismatch exists, the actual error indicates that the majority of annotations is correct with respect to satisfying commutativity. In fact, only the elements which are contained in the Java Model as `orphanTypes` or `compilationUnits` are annotated with a too broad annotation.

Details Fig. 10.3.2 presents, from left to right, the F2DMM models resulting from transforming the DBC product line and employing each of the three completion strategies, the *container*, the *contained*, and the *combined* strategy. The annotations of the F2DMM model resulting from applying the combined strategy are not simplified. This allows distinguishing the parent from children annotations based on which the combined annotation is computed.

Annotations Determined Based on Trace Information The figure indicates that the propagation algorithm maps a correct annotation onto each target element recorded in a 1:1 trace element. Blue boxes highlight these elements, which are referenced as target element of a correspondence element of the BXTend trace. For instance, the package `database` as well as each class

⁵ We remove the empty property, which is required to insert a bidirectional associations in the correct package in the Ecore2UML transformation, from the created association before executing the following UML2Java transformation.

container completion	contained completion	combined completion
<p>Mapping Model</p> <p>Propagation Strategy [Dependency Conflicts: Forward, Missing Annotations: Ignore]</p> <p>Model database: DBContent</p> <p>Package java: DBContent</p> <p>Package database: DBContent</p> <p>Enum Declaration EmploymentState: Employed</p> <p>Class Declaration Person: Person</p> <p>Modifier public: DBContent</p> <p>Field Declaration: Name</p> <p>Field Declaration: Employed</p> <p>Field Declaration: Name</p> <p>Field Declaration: not Family</p> <p>Field Declaration: Person</p> <p>Field Declaration: Person</p> <p>Class Declaration Media: Media</p> <p>Class Declaration Family: Family</p> <p>Class Declaration DatabaseContent: DBContent</p> <p>Class Declaration Animal: Animal</p> <p>Class Declaration Book: Book and Media</p> <p>Class Declaration Song: Song and Media</p> <p>Class Declaration Movie: Movie and Media</p> <p>Class Declaration Domesticated: Domesticated and Animal</p> <p>Class Declaration Wildlife: Wildlife and Animal</p> <p>Class Declaration DatabaseContent_To_persons: Person</p> <p>Class Declaration DatabaseContent_To_books: Media and Book</p> <p>Class Declaration DatabaseContent_To_movies: Media and Movie</p> <p>Class Declaration DatabaseContent_To_songs: Media and Song</p> <p>Class Declaration DatabaseContent_To_animals: Animal</p> <p>Class Declaration DatabaseContent_To_families: Family and Relation</p> <p>Class Declaration Pets_To_owner: Pet and Person and Domesticated</p> <p>Class Declaration Family_To_children: Family and Relation</p> <p>Class Declaration Parent_To_familyP: Relation and Family</p> <p>Array Type boolean[]: DBContent</p> <p>Array Type int[]: DBContent</p> <p>Array Type String[]: DBContent</p> <p>Array Type double[]: DBContent</p> <p>Array Type long[]: DBContent</p> <p>Primitive Type Void void: DBContent</p> <p>Parameterized Type java.util.List-EmploymentState: DBContent</p> <p>Parameterized Type java.util.List-Person: DBContent</p> <p>Type Access: DBContent</p> <p>Type Access: DBContent</p> <p>Parameterized Type java.util.List-Media: DBContent</p> <p>Parameterized Type java.util.List-Family: DBContent</p> <p>Parameterized Type java.util.List-DatabaseContent: DBContent</p> <p>Parameterized Type java.util.List-Animal: DBContent</p> <p>Parameterized Type java.util.List-Book: DBContent</p> <p>Parameterized Type java.util.List-Song: DBContent</p> <p>Parameterized Type java.util.List-Movie: DBContent</p> <p>Parameterized Type java.util.List-Domesticated: DBContent</p> <p>Parameterized Type java.util.List-Wildlife: DBContent</p> <p>Compilation Unit EmploymentState.java: DBContent</p> <p>Compilation Unit Person.java: DBContent</p> <p>Compilation Unit Media.java: DBContent</p> <p>Compilation Unit Family.java: DBContent</p> <p>Compilation Unit DatabaseContent.java: DBContent</p> <p>Compilation Unit Animal.java: DBContent</p> <p>Compilation Unit Book.java: DBContent</p> <p>Compilation Unit Song.java: DBContent</p> <p>Compilation Unit Movie.java: DBContent</p> <p>Compilation Unit Domesticated.java: DBContent</p> <p>Compilation Unit Wildlife.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_persons.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_books.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_movies.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_songs.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_animals.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_families.java: DBContent</p> <p>Compilation Unit Pets_To_owner.java: DBContent</p> <p>Compilation Unit Family_To_children.java: DBContent</p> <p>Compilation Unit Parent_To_familyP.java: DBContent</p>	<p>Mapping Model</p> <p>Propagation Strategy [Dependency Conflicts: Forward, Missing Annotations: Ignore]</p> <p>Model database: DBContent</p> <p>Package java: DBContent</p> <p>Package database: DBContent</p> <p>Enum Declaration EmploymentState: Employed</p> <p>Class Declaration Person: Person</p> <p>Modifier public: DBContent</p> <p>Field Declaration: Name</p> <p>Field Declaration: Employed</p> <p>Field Declaration: Name</p> <p>Field Declaration: not Family</p> <p>Field Declaration: Person</p> <p>Field Declaration: Person</p> <p>Class Declaration Media: Media</p> <p>Class Declaration Family: Family</p> <p>Class Declaration DatabaseContent: DBContent</p> <p>Class Declaration Animal: Animal</p> <p>Class Declaration Book: Book and Media</p> <p>Class Declaration Song: Song and Media</p> <p>Class Declaration Movie: Movie and Media</p> <p>Class Declaration Domesticated: Domesticated and Animal</p> <p>Class Declaration Wildlife: Wildlife and Animal</p> <p>Class Declaration DatabaseContent_To_persons: Person</p> <p>Class Declaration DatabaseContent_To_books: Media and Book</p> <p>Class Declaration DatabaseContent_To_movies: Media and Movie</p> <p>Class Declaration DatabaseContent_To_songs: Media and Song</p> <p>Class Declaration DatabaseContent_To_animals: Animal</p> <p>Class Declaration DatabaseContent_To_families: Family and Relation</p> <p>Class Declaration Pets_To_owner: Pet and Person and Domesticated</p> <p>Class Declaration Family_To_children: Family and Relation</p> <p>Class Declaration Parent_To_familyP: Relation and Family</p> <p>Array Type boolean[]: DBContent</p> <p>Array Type int[]: DBContent</p> <p>Array Type String[]: DBContent</p> <p>Array Type double[]: DBContent</p> <p>Array Type long[]: DBContent</p> <p>Primitive Type Void void: DBContent</p> <p>Parameterized Type java.util.List-EmploymentState: DBContent</p> <p>Parameterized Type java.util.List-Person: DBContent</p> <p>Type Access: DBContent</p> <p>Type Access: DBContent</p> <p>Parameterized Type java.util.List-Media: DBContent</p> <p>Parameterized Type java.util.List-Family: DBContent</p> <p>Parameterized Type java.util.List-DatabaseContent: DBContent</p> <p>Parameterized Type java.util.List-Animal: DBContent</p> <p>Parameterized Type java.util.List-Book: DBContent</p> <p>Parameterized Type java.util.List-Song: DBContent</p> <p>Parameterized Type java.util.List-Movie: DBContent</p> <p>Parameterized Type java.util.List-Domesticated: DBContent</p> <p>Parameterized Type java.util.List-Wildlife: DBContent</p> <p>Compilation Unit EmploymentState.java: DBContent</p> <p>Compilation Unit Person.java: DBContent</p> <p>Compilation Unit Media.java: DBContent</p> <p>Compilation Unit Family.java: DBContent</p> <p>Compilation Unit DatabaseContent.java: DBContent</p> <p>Compilation Unit Animal.java: DBContent</p> <p>Compilation Unit Book.java: DBContent</p> <p>Compilation Unit Song.java: DBContent</p> <p>Compilation Unit Movie.java: DBContent</p> <p>Compilation Unit Domesticated.java: DBContent</p> <p>Compilation Unit Wildlife.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_persons.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_books.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_movies.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_songs.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_animals.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_families.java: DBContent</p> <p>Compilation Unit Pets_To_owner.java: DBContent</p> <p>Compilation Unit Family_To_children.java: DBContent</p> <p>Compilation Unit Parent_To_familyP.java: DBContent</p>	<p>Mapping Model</p> <p>Propagation Strategy [Dependency Conflicts: Forward, Missing Annotations: Ignore, transitive]</p> <p>Model database: DBContent</p> <p>Package java: DBContent and (DBContent or DBContent)</p> <p>Package database: DBContent</p> <p>Enum Declaration EmploymentState: Employed</p> <p>Class Declaration Person: Person</p> <p>Modifier public: Person</p> <p>Field Declaration: Name</p> <p>Field Declaration: Employed</p> <p>Field Declaration: Name</p> <p>Field Declaration: not Family</p> <p>Field Declaration: Person</p> <p>Field Declaration: Person</p> <p>Class Declaration Media: Media</p> <p>Class Declaration Family: Family</p> <p>Class Declaration DatabaseContent: DBContent</p> <p>Class Declaration Animal: Animal</p> <p>Class Declaration Book: Book and Media</p> <p>Class Declaration Song: Song and Media</p> <p>Class Declaration Movie: Movie and Media</p> <p>Class Declaration Domesticated: Domesticated and Animal</p> <p>Class Declaration Wildlife: Wildlife and Animal</p> <p>Class Declaration DatabaseContent_To_persons: Person</p> <p>Class Declaration DatabaseContent_To_books: Media and Book</p> <p>Class Declaration DatabaseContent_To_movies: Media and Movie</p> <p>Class Declaration DatabaseContent_To_songs: Media and Song</p> <p>Class Declaration DatabaseContent_To_animals: Animal</p> <p>Class Declaration DatabaseContent_To_families: Family and Relation</p> <p>Class Declaration Pets_To_owner: Pet and Person and Domesticated</p> <p>Class Declaration Family_To_children: Family and Relation</p> <p>Class Declaration Parent_To_familyP: Relation and Family</p> <p>Array Type boolean[]: DBContent and (DBContent)</p> <p>Array Type int[]: DBContent and (DBContent)</p> <p>Array Type String[]: DBContent and (DBContent)</p> <p>Array Type double[]: DBContent and (DBContent)</p> <p>Array Type long[]: DBContent and (DBContent)</p> <p>Primitive Type Void void: DBContent</p> <p>Parameterized Type java.util.List-EmploymentState: DBContent and (DBContent or DBContent)</p> <p>Parameterized Type java.util.List-Person: DBContent and (DBContent or DBContent)</p> <p>Type Access: DBContent</p> <p>Type Access: DBContent</p> <p>Parameterized Type java.util.List-Media: DBContent and (DBContent or DBContent)</p> <p>Parameterized Type java.util.List-Family: DBContent and (DBContent or DBContent)</p> <p>Parameterized Type java.util.List-DatabaseContent: DBContent and (DBContent or DBContent)</p> <p>Parameterized Type java.util.List-Animal: DBContent and (DBContent or DBContent)</p> <p>Parameterized Type java.util.List-Book: DBContent and (DBContent or DBContent)</p> <p>Parameterized Type java.util.List-Song: DBContent and (DBContent or DBContent)</p> <p>Parameterized Type java.util.List-Movie: DBContent and (DBContent or DBContent)</p> <p>Parameterized Type java.util.List-Domesticated: DBContent and (DBContent or DBContent)</p> <p>Parameterized Type java.util.List-Wildlife: DBContent and (DBContent or DBContent)</p> <p>Compilation Unit EmploymentState.java: DBContent</p> <p>Compilation Unit Person.java: DBContent</p> <p>Compilation Unit Media.java: DBContent</p> <p>Compilation Unit Family.java: DBContent</p> <p>Compilation Unit DatabaseContent.java: DBContent</p> <p>Compilation Unit Animal.java: DBContent</p> <p>Compilation Unit Book.java: DBContent</p> <p>Compilation Unit Song.java: DBContent</p> <p>Compilation Unit Movie.java: DBContent</p> <p>Compilation Unit Domesticated.java: DBContent</p> <p>Compilation Unit Wildlife.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_persons.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_books.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_movies.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_songs.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_animals.java: DBContent</p> <p>Compilation Unit DatabaseContent_To_families.java: DBContent</p> <p>Compilation Unit Pets_To_owner.java: DBContent</p> <p>Compilation Unit Family_To_children.java: DBContent</p> <p>Compilation Unit Parent_To_familyP.java: DBContent</p>

Figure 10.3.2: DBC Java F2DMM models created by base UML2Java transformation.

declaration created for UML classes or associations carry the annotation which is mapped onto the respective source element. These annotations satisfy commutativity.

Completed Annotations of Elements Not Contained in Root of Model In contrast, the remaining mapping elements which are surrounded by red rectangles demonstrate annotations which the respective completion strategy computes. Each *Modifier*, such as the one of the class declaration *Person*, which is created additionally in a rule besides the pivot target element (which is its direct container) receives its annotation from the completion strategy. The *container* completion strategy, the result of which is shown on the left, maps the annotation of the parent onto each model element without annotation. For the modifiers contained in abstract type declarations, field declarations, or method declarations, this strategy suffices to achieve commutativity and none of the annotations has to be refined. The *contained* strategy, maps the root annotation, *DBContent*, onto the modifier instead because the modifier is a leaf in the model tree such that no children annotations are present. Due to the hierarchical filter of *Family* (*forward* strategy), however, this annotation does not influence commutativity: the hierarchical filter will remove the modifier if no proper container is present. In our example, the annotations for elements contained in field or method declarations, such as type accesses, are correct, too. This can always be assumed when the annotation of the field or method declaration implies the annotations of their types.

Completed Annotations of Elements Contained in Root of Target Model Conversely, the completion strategies map the root feature, *DBContent*, as annotation onto the orphanTypes

(i.e., the `ParameterizedTypes`, `ArrayType`, etc.) and the `compilationUnits`. The parent of these elements is the `Java Model` which is annotated with the (broad) root feature, `DBContent`. The compilation units do not comprise children objects but only cross references to the types which they store. Consequently, the completion strategies assign either the parent annotation of the Java model (i.e., `DBContent`) or the root feature (i.e., also `DBContent`) because no annotated children exist. Although the parameterized types contain two type accesses as children, none of the completion strategies can compute a more specific annotation. This is due to the fact that the parent of the (sub-)model tree is the Java model and that none of the leaf nodes, the type accesses, is annotated. Consequently, the *contained* strategy can only assign the annotation `DBContent`. As a consequence of the broad annotations computed by the container and the contained strategy, the combined strategy cannot compute a more accurate annotation for the parameterized types and the compilation units.

Fig. 10.3.2 highlights the parameterized type and the compilation unit which are created for the class `Person`. While the class declaration which is also created and stored as single (pivot) target element in the trace is annotated correctly, the remaining two elements are not. Consequently, all compilation units and parameterized types remain in transform-filter variants whereas they are not present filter-transform variants which violates commutativity.

Results of Graph and HAS Product Lines The same situation holds true for the Java models for the Graph product line and the HAS product line: The completion strategies compute an annotation which satisfies commutativity for the elements contained in classifiers and their body declarations given the hierarchical `Familie` filter. In contrast, the annotations computed for the parameterized types and compilations units and their contained elements are too broad.

Actual Error Despite the broad annotations and the high severity errors, it suffices to modify the annotations of the parameterized types and the compilation units in each of the chosen completion strategies. For that reason, in the DBC product line the actual error is of 4.22% because it requires to change (at minimum) 29 out of the 688 annotations. In the Graph product line only 21 out of 753 annotations have to be replaced (2.79%) and in the HAS product line the annotations of 41 parameterized types and of 41 compilation units (i.e., 82 mappings out of 1342 mapping elements in the target F2DMM) have to be repaired manually. Thus, the actual error is of 6.11% in the HAS product line and still means that more than 90% of all automatically assigned annotations are correct with respect to achieve commutativity.

Summary In sum, the results of conducting the base UML2Java transformation reveal that the assignment of annotations is complete but that the completion strategies are not able to compute annotations that are specific enough to achieve at least one pair of matching transform-filter and filter-transform variants. However, computing the severity error and manually determining the actual error by repairing a minimum set of annotations (only nodes contained in the model root), indicates that more than 90% of the annotations automatically computed by the 1:1 propagation together with a completion strategy suffice to satisfy commutativity.

10.3.3 Advanced UML2Java

The *advanced* UML2Java transformation extends the base version inasmuch as it adds an implementation package to each package which contains a class declaration in the base version. In addition, it does not only create a class declaration for each UML `Class` but also an interface declaration. While the implementation package stores the class declarations, which are recorded in the incomplete trace, the original package, which corresponds with the UML package holding the UML class, contains interface declarations.

Effect of Model Structure and Type of Completion Strategy Due to the modified structure of the target model, this transformation serves to examine whether the `combined` or the `contained` completion strategy positively affect the accuracy and, thus, the correctness of the computed annotations for these models. The same three models, the DBC, Graph, and HAS



Figure 10.3.3: DBC Java F2DMM model created by advanced UML2Java transformation.

UML domain and mapping model serve as input to the transformation definition. Accordingly, the same number of target elements is annotated and therefore the information used for completing the annotation is exactly the same. This fact allows to investigate the effect of the *model structure* on the accuracy of annotations, in addition.

Overview The lower part of Table 10.4 lists the computed error values. As in the base UML2Java version, the absolute error remains at 100% and the severity error higher than 10% whereas the actual error is at the size of about 5%. Furthermore, there is no difference in accuracy between the completion strategies. However, as Table 10.5 will show, less annotations have to be fixed in the HAS model than in the Graph and DBC models in relation to the number of annotations assigned based on trace information. The structure and annotations of the target model cause this effect which the end of this section explains in more detail.

I Database Content and Graph Product Line

Results DBC The trace-based propagation, executed after the reused advanced UML2Java Bxtend transformation, creates 788 target mapping elements out of which 63 elements are annotated based on trace information in the DBC product line. Fig. 10.3.3 depicts the mapping model for the DBC product line resulting from the propagation based on the *container* completion strategy. At the coarse-grained level, the figure highlights model elements, which are referenced by a trace element, and their annotation with blue colored rectangles, on the one hand. On the other

hand, red rectangles enclose the model elements which are not referenced by BXTend trace element and, thus, receive their annotation from one of the completion strategies. The figure refrains from demonstrating all created primitive and array types because their annotations do not differ from the base UML2Java transformation and therefore do not add new insights. The annotations of the additionally created package `impl`, the interfaces created for a class, the parameterized types, and compilation units are computed based on the parent annotation in this example.

Annotations of 1:1 Mappings The algorithm maps the annotation of the source model element onto the Java element which the incomplete trace records as target element. As examples, the package `database`, the enumeration `EmploymentState`, the classes created for associations (e.g., `Database_To_persons`), and the regular class declarations receive annotations in this way.

Annotations of Interface Declarations Due to the container completion strategy, each of the declared interfaces receives the annotation of its parent, the package `database`, which means the root feature `DBContent`. This annotation is too broad for the majority of interfaces in this package. For instance, the interfaces `IPerson` and `IMedia`, depicted below the enumeration declaration in Fig. 10.3.3, should be integrated in variants only which select the features `Person` and `Media`, respectively. Conversely, the annotation of the interface declaration `IDatabaseContent` is correct as it is the same as the corresponding source annotation.

Annotations of Elements Stored in Root of Model The same fact holds for the parameterized types and compilation units, onto which the annotation `DBContent` of the Java model is mapped. Besides the annotations of the additionally created interface declarations, the annotations assigned to the parameterized type and compilation unit created for the enumeration type are too broad, too. This unveils the same effect as in the base BXTend UML2Java transformation: The trace records none of the parameterized types or compilation units and the Java model, annotated with the root feature `DBContent`, constitutes the parent annotation.

Annotation of Package `impl` The annotation of the package `impl` deviates in the completion strategies: The contained strategy computes the annotation by combining the annotations of the contained class declarations, which receive the annotation of the corresponding source object, in a disjunctive expression. This annotation is mapped onto the package `impl`. Since in the DBC product line the class `DBContent`, which is annotated with the mandatory root feature, is contained in the package, the computed annotation does not affect the presence of the package in derived variant: the package is present in every variant due to the annotation of the class `DBContent`.

Further Created Target Mapping Models Even though the annotations resulting from the remaining completion strategies are not depicted in the figure, they are of the same accuracy for the parameterized types and compilation units in the DBC product line as well as in the Graph product line. Furthermore, since the interface declarations do not comprise method stubs, the completion algorithm needs to compute the annotation of a leaf node meaning that the contained and the combined strategy either assign the root feature or the annotation of the parent which is also the root feature.

II HAS Product Line

As opposed to the DBC and Graph product lines, the HAS product line constitutes more complex and contains, for example, package hierarchies which are *not* annotated with the root feature. As a consequence, compared to the DBC and Graph product line less annotations of interface declarations have to be repaired after the trace-based propagation.

In the DBC and Graph product line, all annotations of interface declarations which do not correspond with a mandatory source element, such as the class `Color` in the Graph product line, have to be repaired. Conversely, since some of the packages in the HAS product line carry the same optional annotation as their contained interface declarations, less annotations of interfaces have to be fixed. In total, the annotations of only 15 out of 35 additionally *created interface declarations*

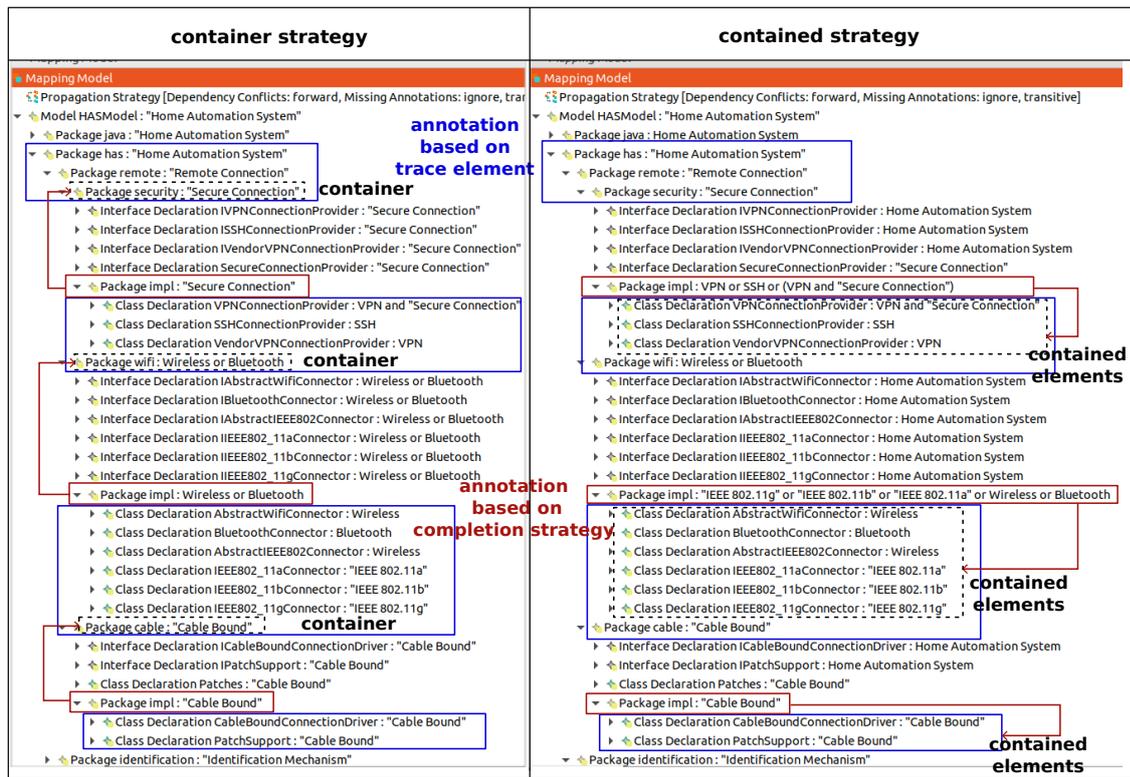


Figure 10.3.4: Effect of container and contained completion strategy in HAS Java F2DMM model.

(i.e., approximately 43%) have to be exchanged manually. In contrast, in the Graph product line 5 out of 8 (i.e., 62.5%) and in the DBC product line 9 out of 10 annotations of interface declarations have to be repaired.

The parameterized types and compilations units represent the majority of elements with wrong annotations mapped onto them similar as in the DBC and Graph product lines. The packages `impl` stored in each regular package which contains classifier declarations are not annotated with the root feature but either equally with the annotation of its parent package in the container strategy or with the annotations of the contained class declaration combined in a disjunctive Boolean expression in the contained strategy.

Effect of Container Strategy Fig. 10.3.4 presents the HAS mapping model which results from applying the container completion strategy. The figure highlights some annotations which are assigned to the implementation packages and which indicate that relatively⁶ less annotations have to be repaired than in the base version. On the left, the figure shows excerpts of the model resulting from applying the container completion strategy whereas the right side depicts excerpts of the model resulting from applying the contained strategy.

Again, annotations which were not computed by a completion strategy but are recorded in trace elements are surrounded by blue rectangles whereas the figure highlights each of the implementation packages, which are not referenced by a trace element and receive the annotation by the completion strategy, with red rectangles. The annotation "Secure Connection" is mapped onto the first (from top to bottom) highlighted package `security`. Accordingly, each of the interfaces and the contained package `impl` are assigned the same annotation. While the annotation is more specific than assigning the root feature, the contained class declarations carry annotations which realize distinct optional features that are contained in the feature group "Secure Connection" (i.e., `VPN` and `SSH`), of the respective feature model. As a consequence, while the annotation suf-

⁶ For determining the influence of the model structure on the accuracy of annotations, the ratio between the number of target elements annotated based on trace information and the number of target elements annotated based on a completion strategy must be considered.

fices for the package `impl`, the annotation of some of the interfaces is too broad. Except for the annotation of the `SecureConnectionProvider`, the annotations of the remaining interfaces in the package `security` have to be refined.

The same effect can be observed in the package `wifi`. The broad annotation `Wireless or Bluetooth` assigned to each of the interfaces contained in the package implies that each one is present whenever one of the two features is selected. This annotation correctly represents the visibility of the contained package `impl` because it contains class declarations with annotations stating only either one of the features. Conversely, the annotation is too broad for the interface declarations in this package: The different IEEE wireless standards are each represented with a distinct feature and constraints among them. Consequently, the annotation `Wireless` does not suffice to satisfy commutativity as, for instance, the interface `IEEE802_11bConnector` should only be present when the feature `"IEEE802.11b"` is selected and not when any of the features representing one of the standards is selected in a feature configuration.

In contrast, the specificity of the annotations of all elements contained in the package `cable` suffices to satisfy commutativity and does not require a manual modification. The corresponding source elements and target elements, the class declarations `CableBoundConnectionDriver` and `PatchSupport` are both annotated with the same expression `"Cable Bound"`. Accordingly, no manual modifications are required in this case. Similarly, the annotations mapped onto the elements in the packages `oven`, `shutters` and `addOnPackage` suffice to satisfy commutativity. The annotations of interfaces in the package `cond` (representing the logic for the two optional features air condition and temperature control) and `identification` (representing several distinct optional authorization mechanisms) require to manually modify the annotations of some of the contained interfaces.

Effect of Contained Strategy The right-hand side of Fig. 10.3.4 demonstrates the same excerpts of the resulting mapping model when applying the *contained* completion strategy which computes the annotation of elements missing one based on the annotation of their children elements. The completion algorithm maps the root feature, `"Home Automation System"`, onto each of the interfaces because they do not comprise any child element⁷. Conversely, the annotation mapped onto the package `impl` combines the annotation of each of its contained class declarations in a disjunctive form. As an example, the annotation of the package `impl` contained in the package `security` combines the annotation of each of the three contained classes in the disjunctive form: `VPN or SSH or (VPN and "Secure Connection")`. Consequently, it reflects the contained annotations accurately and satisfies the same set of configurations as all of its contained class declarations. The same effect holds for each of the implementation packages.

Effect of Combined Completion Strategy In the *combined* strategy, which combines the annotations computed by the container and contained strategy, no different effects can be observed. For that reason, the figure does not explicitly show the results. The annotations mapped onto the interfaces are the same as in the container strategy. Therefore, the annotations of the same interfaces have to be repaired. The annotations mapped onto the packages named `impl` combine the annotation of the container and the contained elements in the conjunction. Although, they are more complex, they are semantically equal as the one computed with the contained strategy which also does not affect the accuracy.

Equal Error Rates in Completion Strategies When comparing the effect of the strategies, the measured error values are equal in each completion strategy. This effect is mostly caused by the Famile filter together with the chosen subject product lines and boils down to two points: On the one hand, the hierarchical filter provokes this effect. For instance, deriving a single-variant model from the mapping model resulting from applying the *contained strategy* with a flat filter, would pertain each of the interfaces in the model regardless whether the container is present. The Famile filter, in contrast, will keep an element in a configuration only if its parent element is kept, too. Accordingly, a flat filter would derive an *invalid* product which could not be persisted

⁷ If the interfaces created for a class were containing methods, the pivot element recorded in the incomplete trace would be the method stored in the class declaration. Thus, the methods in the interface would miss an annotation, too.

Table 10.5: Effects on annotation correctness by the presence of annotated elements in the target model. ¹ Calculated by subtracting the number of trace elements from all mappings.

UML2Java	Product Line	#target mappings	#incomplete mappings ¹	#repaired annotations	$ratio_{repaired}$ [%]
base	DBC	688	625	29	0.47
	Graph	753	682	21	0.32
	HAS	1342	1192	82	0.87
advanced	DBC	788	725	47	0.56
	Graph	827	756	31	0.39
	HAS	1714	1564	132	0.81

due to pending references whereas the hierarchical filter prevents this situation by keeping only elements, the container of which is also selected. As consequence, while the *contained* strategy would provoke a higher error rate and potentially invalid products when using a *flat* filter, the *hierarchical* filter prevents this situation and gives rise to the same error rate.

On the other hand, annotated contained element may include the same annotation as the container of the element missing an annotation. Then, the container and contained strategy, and, consequently, the combined one have the same effect. In this scenario, the package `impl` exhibits this effect. A broad annotation is mapped onto its container which subsumes the annotation of each of the contained elements. As a consequence, combining the annotation of the contained element in a disjunctive form has the same effect on commutativity. Furthermore, the annotations mapped onto interface by the contained strategy would be too broad. Here the model filter eliminates the distinguishing factor as discussed as first point.

Influence of Model Structure This example demonstrates that the annotations assigned to containers, particularly the packages in this example, influence the size of the *actual error*. Given the number of elements without an annotation automatically assigned by the trace-propagation, in the advanced UML2Java transformation the ratio of annotations that have to be repaired decreases compared to the base version. This is in contrast to the DBC and the Graph product line where the number of annotations which have to be repaired increases compared to the number of incomplete annotations.

In concrete, from left to right, Table 10.5 gives an overview of the number of all mapping elements that are present in the created target models (i.e., *#target mappings*), the number of mappings for target elements not recorded in the trace (i.e., *#incomplete mappings*) and the number of annotations that have to be manually *repaired*. To consider the effect of the automatically propagated annotations, the $ratio_{repaired}$ measure takes them into account as follows:

$$ratio_{repaired} = \frac{\#repaired}{\#incomplete\ mappings} \times \frac{\#annotated\ by\ trace}{\#incomplete\ mappings}$$

Accordingly, the measure takes the ratio of annotation information provided by the trace into account in the second term. The number of annotations provided by the trace (*#annotated by trace*) results from subtracting the number of incomplete mappings from all mappings. This number is divided by the number of incomplete mappings. For example, in the base DBC product line the information ratio is 10.1% meaning that one annotation provided by the trace is used for approximately 9.9 elements on average. The ratio of information is multiplied by the ration of wrong annotations that have to be repaired out of all incomplete mappings. Thus the measure indicates how many annotations computed by a completion strategy have to be repaired with respect to the present trace information. This measure targets the fact that the actual error will increase if more elements remain without annotation but the present trace information stays the same.

In the DBC product line, it is necessary to fix the annotations of 29 out of 625 mapping elements which received the annotation by a completion strategy when conducting the base UML2Java transformation. 63 elements are annotated based on trace information such that the ratio of

Table 10.6: Results of conducting the Ecore2SQL transformations. ¹ in [%]

Product Line	#target mappings	BXtend			ATL/EMFTVM			QVT-O		
		err_{abs}^1	err_{sev}^1	err_{act}^1	err_{abs}^1	err_{sev}^1	err_{act}^1	err_{abs}^1	err_{sev}^1	err_{act}^1
DBC	623	100.00	1.47	2.89	0.00	100.00	30.99	0.00%		
Graph	605	86.67	1.18	1.82	0.00	100.00	42.38	0.00		

repaired annotation in sight of these elements remains at 0.47%. Similarly, the values of the remaining examined scenarios are calculated.

From the table the effect of the model structure and the annotations becomes obvious. In the advanced version, the interfaces and packages are added to the package mapped in the trace while the trace information stays the same. While the ratio of repairs increases for the DBC and Graph product line, it decreases for the HAS product line. As discussed above, this confirms that less annotations have to be repaired due to the structure and that more specific annotations assigned to those elements can positively affect accuracy. In the HAS product line, less annotations of interfaces have to be exchanged because their containing package is annotated more specifically as in the DBC and Graph product line.

Summary All in all, the examination of the two versions of the UML2Java transformation reveals that more than 90% of the annotations are computed correctly despite the absolute error of 100%. Furthermore, in the base and advanced version, the accuracy of the three completion strategies is exactly the same. This is, however, mainly an effect of the hierarchical filter used in Famile. If a flat filter was used, the contained strategy would provoke worse accuracy because it would retain several model elements (e.g., the modifier of type declarations or the interface declarations created for class declarations) in the derived variant despite a missing containing element. To this end, a variant derived in this way cannot be persisted as EMF resource and would therefore, render an entire mismatch between the filter-transform and the consequently missing transform-filter variant. Finally, the evaluation also indicates that the more specific the annotations are that serve for computing the missing annotations, as is the case in the advanced transformation of the HAS product line the less annotations have to be repaired.

10.3.4 Ecore2SQL

In contrast to the Ecore2UML and the UML2Java transformation, which are implemented in one language only, we implemented the Ecore2SQL transformation in BXtend, ATL/EMFTVM, and QVT-O to compare the propagation result based on traces of different completeness levels.

In this scenario, the propagation based on the generation-complete ATL/EMFTVM and the (generation-)complete QVT-O traces results in 100% correctness each. In contrast, the missing information in the BXtend trace causes an absolute error of 100% in the DBC product line and of about 86% in the Graph product line whereas the severity and, particularly, the actual error remain at less than 5% percent meaning that 95% of the annotations are computed correctly.

Details In more detail, Fig. 10.3.5 depicts the F2DMM models which result from performing the trace-based propagation based on the three traces of different granularity. The left side depicts the mapping model resulting from propagating annotations based on the information of the incomplete BXtend trace whereas the right side depicts the mapping model resulting from using the information of the complete QVT-O trace. The middle of the figure depicts the mapping model resulting from the propagation based on the generation-complete ATL/EMFTVM trace. Moreover, the red rectangles highlight the annotations mapped onto the object-table in all three models. Thereby, the figure illustrates the differences between the annotations computed by the completion strategies. Similarly, the blue rectangles highlight the annotations mapped onto the contents of the table **Person** as one representative of the remaining tables that are created.

ATL/EMFTVM Annotations Comparing the annotations mapped onto the columns of the table `Person` allows drawing conclusions on the effect of trace completeness. The annotations of the corresponding elements resulting from the propagation based on the `BXtend` and the `ATL/EMFTVM` trace do not deviate. This is due to the fact, that the `BXtend` trace records the `Table` and the columns apart from the `id`-column as pivot target elements. The additional elements created in the table (e.g., the `SQL` annotations) do not require a more specific annotation. Similarly, the table and the columns are target elements of trace elements in the `ATL/EMFTVM` trace and receive the same annotation as mapped onto the respective source element.

QVT-O Annotations In contrast, the elements of the mapping resulting from the propagation based on the *QVT-O trace* combine the source annotation with the annotation of the context elements. In the case of the table `Person`, the `Schema` (annotation: `DBContent`) is the context element of the `Class2Table` rule. The rule creates the `id`-column, annotations, a primary key, and one foreign key into the `object`-table. Accordingly, the annotation `Person` and `DBContent`, which combines the annotation of the source and the context element in a conjunction, is mapped onto these elements. The remaining *columns* stem from the rule which transforms `EAttributes`, where the table (annotation: `Person` and `DBContent`) serves as context element. Conversely, the second foreign key, stated as last contained element in the table `Person`, is created by the transformation rule which converts `containment`-references (i.e., no context elements in the trace).

As explained in Sec. 10.2.3, the `Ecore2SQL` transformation employs the `main` method (entrypoint) to prescribe the sequence of executing rules and requires the classifiers to be transformed before their references. For that reason, the mapping-rules which transform `EReferences` do not receive the containing classifier explicitly as parameter. Therefore, the classifiers are not recorded in the trace as context elements. As a result, the annotation of the column `persons_inverse` and the foreign key into the database are assigned the annotation `Person` which is mapped onto the source element, the `EReference`, but it is not combined with the annotation of its container.

BXtend Annotations Conversely, the annotations of the columns contained in the table `EObject` differ among each of the three mapping models. The annotations assigned by using the `BXtend` trace information, reside at the lowest level of specificity and result from the *parent* completion strategy in this example. Since neither the `object`-table nor any of its contained elements is the target element of a trace element, the propagation cannot compute their annotation based on trace information. Therefore, the completion strategy first determines the annotation of the tables' parent, which is the annotation `DBContent` mapped onto the `Schema`. Thereafter, the completion strategy assigns this annotations to the remaining elements contained in the table. Since none of the elements *contained* in the table is assigned an annotation and possesses children elements, the contained and combined completion strategies do not compute different annotations. In any completion strategy, the root annotation is assigned to the table `EObject` and its columns, causing the inaccuracy and violation of commutativity.

Results BXtend Despite the fact that each configuration encompasses at least one element onto which a wrong annotation is mapped in the `BXtend` trace-based propagation in the `DBC` product line, the actual number of differences remains low: The `DBC` product line exhibits a severity error of less than 2% for any of the completion strategies. Moreover, at minimum, 18 annotations of 623 mapping elements have to be repaired manually, resulting in an actual error of 2.89%. After these annotations are manually corrected and the transformation and comparison is executed again, an absolute error of 0% is measured meaning that annotations were assigned in the correct way to achieve commutativity. Similarly, in the `Graph` product line, the annotations of *five* columns and corresponding foreign keys have to be repaired manually in the table `EObject`. Additionally, the table `Nodes` comprises a foreign key into the table `Color` which receives the annotation `Nodes` or `GraphProductLine` (depending on the completion strategy). The annotation of this element has to be refined to `Color`, too, in order to achieve commutativity. In total, at minimum *eleven* annotations have to be repaired provoking an actual error of 1.82%.

Results ATL/EMFTVM and QVT-Os Finally, the annotations of the mapping model resulting from the propagation based on the ATL/EMFTVM trace and the QVT-O trace, differ only with respect to their specificity. Since the context element of creating the table `EObject` is the `Schema`, the annotation of the source element is combined in a conjunction with the root feature and assigned in that way to the columns contained in the table `EObject` created by the QVT-O transformation. In contrast, the mapping created based on the ATL/EMFTVM trace only maps the annotations of the respective source elements onto the column. However, the effect on commutativity is the same. Both mapping models created by exploiting the ATL/EMFTVM and the QVT-O traces, satisfy commutativity.

While no mismatching configuration and, thus, no wrong annotation can be found in the Graph product line for each of the two transformations, the comparison of the EMFCompare framework records differences between the filter-transform and the transform-filter variants in the DBC models created by the QVT-O transformation. Conducting a manual textual comparison of a sample of the xmi-files storing the filter-transform and the transform-filter variants, however, proves that there is not a single difference between both. Inspecting the comparison file reveals that, for instance, foreign keys are detected as deleted and added on both sides but are not recognized as equivalent. Manually overriding the `equals()` method of the class representing the metamodel element `ForeignKey` did not help to recognize matching foreign keys correctly. Therefore, we record the error but could not repair any annotation. Thus, the actual error is of 0.00%.

Summary In summary, in the Ecore2SQL transformation, the propagation based on the generation-complete ATL/EMFTVM and QVT-O traces results in correctly annotated target models in both transformation scenarios, the Graph and the DBC product line. Since in this example and with the given filter, the information of the generation-complete trace suffices to satisfy commutativity, the additional information about context elements in the QVT-O trace cannot improve the already correct result. Furthermore, the subject systems show that the completion strategies work as expected and generate mappings models which are completely annotated. However, the completion strategies make no difference so that it suffices to employ the container strategy which is less computational-expensive as accurate as possible.

10.3.5 Model-To-Text

After having examined commutativity in M2M transformation scenarios, this section presents the results of evaluating commutativity in the M2T multi-variant transformations using the aspect-oriented propagation of Xpand. In contrast to the model comparisons, the evaluation compares source code directories and files. Consequently, the first part of this section introduces adapted metrics which allow measuring commutativity. The second and third part illuminate details of the evaluation results in the Ecore2Java and the MoDisco2Java transformations, respectively.

I Error Computation

In contrast to comparing models and their elements, in text representations *several* files build a derived variant and may incorporate erroneous annotations. As a consequence, not only the text inside a file may be mismatching due to a wrong annotation but also an entire file may be missing.

Preliminaries For evaluating commutativity, we perform a String comparison which neglects differences in white spaces and focuses on the remaining text. The resulting String comparison value states the distance between the first two non-matching characters inside the String which does not allow to draw a conclusion about the quantity of the mismatches. Therefore, for computing the error we consider only a Boolean value at the level of files. Either the text contained in two files matches entirely or at least one textual difference exists. The latter also includes the possibility that the corresponding source code file is not present. For that reason, the error computation does not regard the concrete difference value resulting from the String comparison but only the number of files which contain an error. The severity error does not count how many lines of the generated source code or how many single characters in the compared files do not match but the number of files.

Table 10.7: Statistical data and error measurements of the Ecore2Java M2T transformation.

product line	#all derived files	#files with mismatches	#wrong annotations	err_{abs_mt}	err_{sev_mt}	err_{act_mt}
Graph	2196	0	0	0.00%	0.00%	0.00%
DBC	2885	216	4	7.49%	7.05%	1.83%

Adapted Metrics Firstly, the *textual absolute error* weighs the number of all erroneous files (including missing ones) in all derived directories against the total number of files that are derived:

$$err_{abs_mt} = \frac{\#files\ with\ mismatches}{\#all\ derived\ files} \quad (10.1)$$

Secondly, the *textual severity error* regards the fact that each variant may comprise a different number of files. The number of all configurations is n :

$$err_{sev_mt} = \frac{\sum_{i=0}^{n-1} \#files\ in\ i\ with\ mismatches}{\#all\ derived\ files\ in\ i} \quad (10.2)$$

err_{sev_mt} is more fine-tuned than err_{abs_mt} because it considers the fact that the number of created files can vary among the feature configurations. However, it is expected to be of same magnitude as the absolute error.

Finally, the *actual error* can be determined again manually by repairing and counting the number of wrong annotations. This number can be weighed against all lines that incorporate a directive in the multi-variant source code platform.

$$err_{act_mt} = \frac{\#wrong\ annotations}{\#all\ directives} \quad (10.3)$$

Based on these values, we can quantitatively evaluate and discuss the correctness of the generated source code annotations in the following sections.

II Ecore to Java Source Code

The transformation that generates Java source code from Ecore models is able to achieve high accuracy in the assigned annotations as summarized in Table 10.7. All annotations assigned to the source code representing the *Graph* product line are correct and satisfy commutativity. In contrast, the evaluation determines an absolute error of 7.49% when generating source code for the database content Ecore model implying that more than 90% of the files are annotated correctly. The following paragraphs explain the reasons for these results. Notice: As the HAS product line is modeled as UML model, it cannot serve as input to the transformation.

Graph Product Line The multi-variant M2T transformation generates 16 files for the *Graph* Ecore model, an interface as well as a class for each of the 8 **EClasses**. Each of the files starts with the package declaration followed by import-statements computed from the types that are used in the classifier. After declaring the name of the interface or class, the field declarations or corresponding methods are written followed by the method declarations for **EOperations**. Fig. 10.3.6 shows the generated directory and its contained files on the left side and parts of the text fragments created for the **EClass Graph** on the right side. The middle of the figure depicts the interface that is created and stored in the main directory `de.ubt.a11.famile.example.graph` whereas the right side presents the class stored in the contained directory `impl`. At the bottom, the figure highlights the method declaration `getEdge()` which is given a name and created from a corresponding **EOperation** stored in the **EClass**. The blue rectangles highlight the text fragment(s) that are created by the original (single-variant) transformation whereas the red rectangles, which surround the blue ones, stress the fragments which are added by the generic aspect.

directory	Graph interface	Graph class
	<pre> 1 package de.ubt.ail.famile.example.graph; 2 3 /** #IFDEF Nodes && GraphProductLine */ 4 import de.ubt.ail.famile.example.graph.Node; 5 6 /** #IFDEF GraphProductLine */ 7 8 /** #IFDEF Nodes && GraphProductLine */ 9 // @ID: obj_1989224723 10 public interface Graph { 11 12 /** #IFDEF Nodes && GraphProductLine */ 13 // @ID: obj_658781596 14 public void setNodes(java.util.ArrayList<Node> nodes); 15 /** #IFDEF */ 16 17 /** #IFDEF Edges && GraphProductLine */ 18 // @ID: obj_629321967 19 public void setEdges(java.util.ArrayList<Edge> edges); 20 /** #IFDEF */ 21 22 /** #IFDEF Search && GraphProductLine */ 23 // @ID: obj_1989924937 24 public void setSearch(Search search); 25 /** #IFDEF */ 26 27 /** #IFDEF Algorithm && GraphProductLine */ 28 // @ID: obj_333828675 29 public void setAlgorithm(Algorithm algorithm); 30 /** #IFDEF */ 31 32 /** #IFDEF Search && GraphProductLine */ 33 // @ID: obj_27294719 34 public java.util.ArrayList<Adjacency> getAdjList(); 35 /** #IFDEF */ 36 37 /** #IFDEF Nodes && GraphProductLine */ 38 // @ID: obj_244668763 39 Node getNode(java.lang.String name); 40 /** #IFDEF */ 41 42 /** #IFDEF Edges && GraphProductLine */ 43 // @ID: obj_1085941526 44 Edge getEdge(java.lang.String name); 45 /** #IFDEF */ 46 47 /** #IFDEF Search && GraphProductLine */ 48 // @ID: obj_1447678234 49 void buildAdjList(); 50 /** #IFDEF */ 51 } 52 53 /** #IFDEF */ 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 </pre>	<pre> 67 protected Search search; 68 69 public void setSearch(Search search) { 70 this.search = search; 71 } 72 public Search getSearch() { 73 return search; 74 } 75 /** #ENDIF */ 76 77 /** #IFDEF Algorithm && GraphProductLine */ 78 // @ID: obj_333828675 79 80 protected Algorithm algorithm; 81 82 public void setAlgorithm(Algorithm algorithm) { 83 this.algorithm = algorithm; 84 } 85 public Algorithm getAlgorithm() { 86 return algorithm; 87 } 88 /** #ENDIF */ 89 90 /** #IFDEF Search && GraphProductLine */ 91 // @ID: obj_27294719 92 93 protected java.util.ArrayList<Adjacency> adjList; 94 95 public void setAdjList(java.util.ArrayList<Adjacency> adjList) { 96 this.adjList = adjList; 97 } 98 public java.util.ArrayList<Adjacency> getAdjList() { 99 return adjList; 100 } 101 /** #ENDIF */ 102 103 /** #IFDEF Nodes && GraphProductLine */ 104 // @ID: obj_244668763 105 106 // @Override 107 public Node getNode(java.lang.String name) { 108 //PROTECTED REGION ID(obj_244668763) ENABLED START 109 return null; 110 //PROTECTED REGION END 111 } 112 /** #ENDIF */ 113 114 /** #IFDEF Edges && GraphProductLine */ 115 // @ID: obj_1085941526 116 117 // @Override 118 public Edge getEdge(java.lang.String name) { 119 //PROTECTED REGION ID(obj_1085941526) ENABLED START 120 return null; 121 //PROTECTED REGION END 122 } 123 /** #ENDIF */ 124 125 /** #IFDEF Search && GraphProductLine */ 126 // @ID: obj_1447678234 127 128 public void buildAdjList() { 129 //PROTECTED REGION ID(obj_1447678234) ENABLED START 130 //PROTECTED REGION END 131 } 132 /** #ENDIF */ 133 134 } 135 136 } 137 138 } 139 /** #ENDIF */ 140 </pre>

Figure 10.3.6: Excerpt of Graph Java source code created by the Ecore2Java transformation.

In both files, the generated annotations combine the annotation of the source element with the annotation of its container. In the highlighted method, the source element is the EOperation `getEdge()` and the container is the EClass `Graph`, onto which the annotations `Edges` and `GraphProductLine`, respectively, are mapped. Consequently, the opening directive (`#IFDEF`) stated as Java comment before the method declaration requires both features, `Edges` and `GraphProductLine`, to be selected in order to be present in a derived variant. Thus, they are combined in a conjunction. The remaining annotations of similar model elements are computed in the same way. If the parent annotation already occurs in the source annotation literally, it is not included in the conjunction. For example, the annotation of the class and interface declarations or of the method `printContents()` would require combining the annotation of the source element (annotation: `DBContent`) with exactly the same annotation of the parent element. Thus, the computed annotation states the same feature only once in a conjunction.

Lastly, the figure demonstrates the text generated as body of a method on the right side, for instance in the method `getEdge()`. The transformation rule creates a protected block in which the default value corresponding with the return type of the method is returned. Accordingly, the entire body can be customized and this modification is prevented from being overwritten in a consecutive incremental execution of the same transformation as long as the same source node remains in the source model.

In summary, as Table 10.8 mentions in its first row, the multi-variant transformation creates 16 files comprising 954 lines of code in total for the Graph Ecore model as input. Furthermore, the multi-variant transformation creates 244 lines of code which incorporate an opening or closing

Table 10.8: Different types of counted Lines of Code in the Ecore2Java M2T transformation.

product line	#files	#all	#directive	#other comment	#source code
Graph	16	954	244	166	327
DBC	21	843	218	131	317

directive (line including either the String "#IFDEF" or "#ENDIF") and 166 lines including another form of comment (i.e., "//", "/*", "/**/" or "*/"). 217 lines are counted as empty such that 327 lines remain that include implementation code. As a consequence, the generic aspect adds more lines of codes ($244 + 166 = 410$), which make up 42.98% of all created lines, than the original text production which creates the implementation source code, which makes up 34.28%.

As the text comparison does not detect mismatches between the filter-transform files and transform-filter files in any configuration, the absolute as well as the severity and the actual error remain at 0%. Table 10.7 summarizes the data from which the error measurements are created and lists the error values as well as the number of derived files and potentially wrong annotations. Its first row shows that no annotation is wrong in the text generated for the Graph product line such that we conclude that the created multi-variant files are annotated correctly.

DBC Product Line In the second place, the evaluation examines the correctness of the same transformation given the DBC Ecore model as input. The transformation creates 21 files for the database Ecore model. The source model comprises 10 `EClasses`, for each of which an interface and a class declaration are created, and one enumeration which is transformed into a corresponding declaration stored in a separate file.

Similar to Fig. 10.3.6, which demonstrates resulting annotated files of the Graph product line, Fig. 10.3.7 depicts the directory and files that are created by the transformation on the left side whereas the middle and the right side depict the multi-variant interface and class, respectively, created for the `EClass DBContent`. Moreover, similar as in the exemplary description of the outcome of transforming the Graph Ecore model, the annotations of the text fragments shown in the figure combine the annotation of the source node with the annotation of its container.

Furthermore, the figure spotlights the text produced for the containment reference `songs` stored in the class `DBContent`. The text production generates method headers in the Java interface and method bodies as well as a field declaration in the Java class. Preprocessor directives surround these source code fragments which result from applying the respective text-producing rule ϑ and, thus, represent the outcome of the original text production.

Cause of Wrong Annotation While in each configuration the majority of the created class and interface files does not exhibit textual differences, the four files `Person.java`, `PersonImpl.java`, `Domesticated.java` and `DomesticatedImpl.java` provoke a mismatch when the feature `Pet` is not selected⁸. Fig. 10.3.8 displays the relevant excerpts of the feature model and mapping model as well as the relevant source code fragments causing this error.

The mismatch in derived variants occurs in the `import` statements before opening the type declarations. The corresponding rule creates such statement for each non-primitive type that is used as type of property or of a parameter in the source `EClass`. The advice looks up the annotation of the source element (i.e., of the type that is imported) and of its container (i.e., the package in which the `EClass` or `EEnum` is stored) to compute the annotation. Consequently, the interface `Domesticated` is imported in the interface `Person`, depicted on the right side of the figure, whenever the features `Domesticated`, `Animal`, and `DBContent` are selected. However, the import is only required when the feature `Pet` is selected additionally because, only then, the interface is used as parameter of the methods declared in Lines 58 and 59 of the interface.

Although the import is not needed, this fact does not violate commutativity in isolation. While the import is present in the source code derived from the multi-variant source code platform regardless

⁸ We deduced this fact by manually inspecting a random sample of mismatching files in erroneous configurations.

The figure displays a multi-variant interface of a DBC product line. On the left, a 'Mapping Model' tree shows features like Person, Animal, Media, and Family. Below it, the 'Domesticated.java' file shows code for a 'Domesticated' interface. On the right, the 'Person.java' file shows code for a 'Person' interface. Red boxes highlight specific annotations in the code, and arrows point to a 'wrong annotation ('Pet' missing)' label.

Figure 10.3.8: Commutativity violation in multi-variant interfaces of DBC product line.

only present in the transformed source code when also the features `Domesticated` and `Pet` are selected.

Summary On the whole, the transformation of the database product line and the following commutativity evaluation computes an absolute error of 7.49% where 216 out of the 2885 generated files contain at least one syntactic mismatch. While the order of the severity error is of the same size, the actual error remains below *two%* and requires to change *four* annotations (i.e., *four* lines of the all non-empty lines) in total. Only the single import statements in each of the two interfaces `Person` and `Domesticated` and of their respective implementation has to be refined in order to satisfy commutativity.

III MoDisco to Java Source Code

As second example, we performed a transformation of the structural parts of the MoDisco model into Java source code. This transformation serves to demonstrate the feasibility of the generic advice. Without any adaptation the same generic advice can be injected in the original MWE2 workflow which triggers the mentioned transformation. Thus, it offers evidence that the advice is completely definition-independent. Table 10.9 offers an overview of the source code statistics created in this transformation scenario.

Generated Directories Fig. 10.3.9 presents the directory and files which are created for the input model on the left hand-side. On the right hand-side, it depicts the contents of two of the created files. The UML2Java transformation created a class declaration and a corresponding compilation unit for the UML association `database_to_persons`. The MoDisco2Java transformation transformed the compilation unit into the file depicted in the top right corner. First, the contents

Table 10.9: Different types of counted Lines of Code in the MoDisco2Java M2T transformation.

product line	#files	#all	#directive	#other comment	#source code
Graph	21	422	124	62	121
DBC	20	455	98	49	89

mention the package and open the class declaration which is only present when the features **Person** (source element) and **DBContent** (container) are both selected. Inside the class, it represents the two field declarations as two private object field declarations. The visibility is retrieved based on the modifier and the type based on the types stored in the type access. Since both, the visibility and the type, are not generated by the means of a self-contained rule, they are not annotated in a more fine-grained way than their container, the field declaration. The field declaration, on the other hand, receives the annotation **Person** only because the container is not annotated differently. As explained before, the advice includes the same feature in a single conjunction only ones.

The lower part of the figure depicts the text written to the generated file **Person.java**. Similar to the contents of the **Database_To_persons.java** file, only the field declaration in its entirety is embraced with the annotation of the source element combined with the parent annotation. In contrast to the second depicted class, annotations which are more fine-grained than the annotation of the container are mapped onto the source field declaration. Accordingly, the corresponding generated text fragments, such as the field **lastname**, combine the annotation of their source element with the annotation of its container. In this example, the annotation **!Family && Person** is mapped onto the text line `private String lastname;`.

Results Graph Product Line The same situation can be observed when transforming the MoDisco model representing the Graph product line contents. 21 files are created which contain the source code for representing (regular) class declarations holding field and method declarations as well as for representing association classes which only record two field declarations in this example. Annotations are computed and assigned in the same way such that the evaluation of commutativity does not detect any difference among the derived source code variants. Thus, the results allow for concluding that the advice annotates the source code correctly.

Summary The results of computing commutativity reveal that the propagation of annotations is correct because no mismatches are detected in any of the configurations of the DBC and the Graph product lines. This effect is caused by the nature of the transformation which only transforms core elements of the source model the annotations of which suffice to achieve commutativity.

10.4 Discussion

As concluding part of this chapter, this section summarizes and discusses the results of the evaluation. Accordingly, for the trace-based and the aspect-oriented propagation of annotations it shortly summarizes the main insights and discusses them with respect to satisfying commutativity and the implications on the computational model based on the measured metrics. Thereby it answers the evaluation questions.

Road Map In the first place, Sec. 10.4.1 answers the evaluation questions by summarizing the results of evaluating the *trace-based propagation*. In addition, the section discusses the properties postulated in the computational model and the effects of deviations from it in the examined transformation scenarios. In the second place, Sec. 10.4.2 similarly summarizes the results and discusses effects of deviating from the computational model of the *aspect-oriented propagation* followed by answering the evaluation questions based on the findings. The concluding section, Sec. 10.4.3, progresses to analyze the threats to validity of the evaluation and the drawn conclusions.

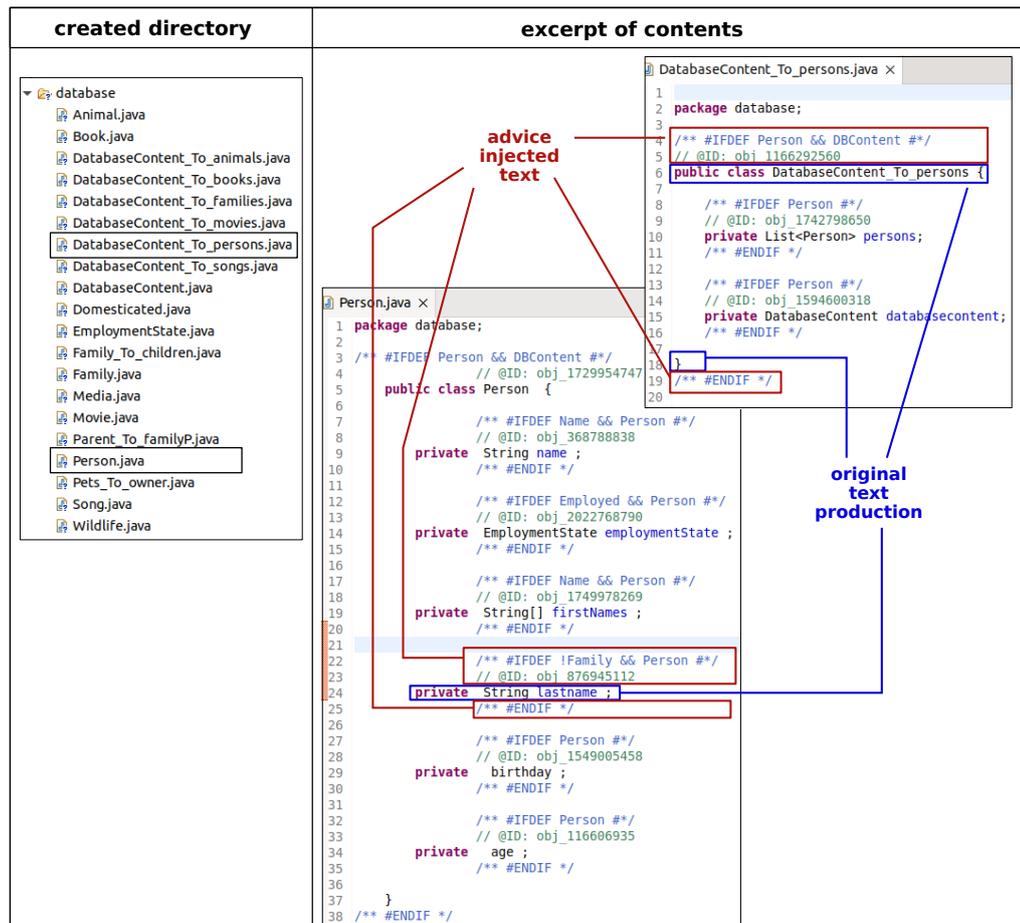


Figure 10.3.9: Excerpts of DBC source code created by MoDisco2Java transformation.

10.4.1 Trace-Based Propagation

From conducting the evaluation based on the different scenarios, we answer the evaluation question after giving an overview of the results. While the first part of this section discusses the conclusions we draw from applying trace-based propagation to the three subject product lines, the second part answers the evaluation questions based this discussion.

I Summary of Results

The following summary first gives an overview of the results, followed by a discussion of the details that may cause the results. The following parts

Overview The results of examining M2M transformation scenarios demonstrate that the trace-based a posteriori annotation propagation is able to construct a mapping model which is completely annotated despite violations to the computational model. Although the propagation based on a *generation-complete* trace violates the computational model, in general, the propagation based on this trace information in the examined scenarios annotated the target model correctly.

Furthermore, the evaluation gives evidence that the majority of annotations (more than 90%) is also correct when propagating annotations based on incomplete trace information and completing the annotations based on a completion strategy in each transformation scenario and subject product line. However, in our subject product lines, the results could not determine a difference in accuracy between the completion strategies which is an effect partly caused by the employed model filter. The sequel dives into the effects of (violating) the properties of the computational model.

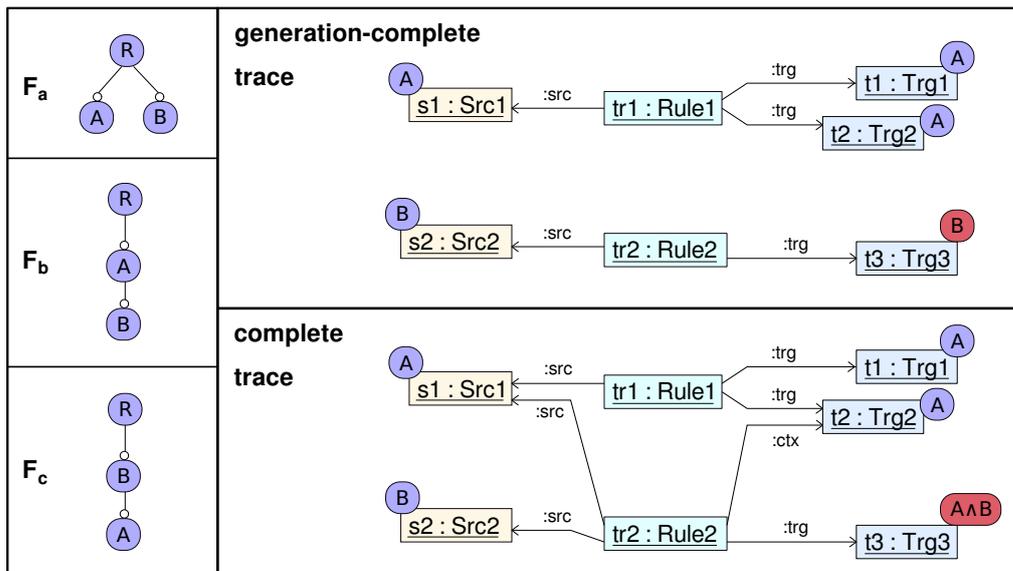


Figure 10.4.1: Influences of annotations, feature model, and trace completeness.

Trace Completeness As a first point, the computational model fosters *complete* traces to guarantee commutativity of trace-based propagation. Therefore, we would expect that the information in generation-complete and incomplete traces does not suffice to satisfy commutativity, such that some of the filter-transform and transform-filter variants deviate from each other.

In the subject systems and transformations which we executed, however, we found that the propagation based on a *generation-complete* trace was able to satisfy commutativity. On the one hand, this may be caused by the kind of annotations, which are mapped onto the source model: If the annotation mapped onto the recorded source elements *implies* the annotation of the non-recorded context information, the source annotation will encode the information that would be computed based on the context elements otherwise. Thus, it suffices to assign the source annotation. For instance, the annotation `Color` implies the annotation `Nodes` and the corresponding class `Node` needs to be present in order to add a reference to the class `Color` in the Graph product line. However, it suffices to map the annotation `Color` onto the reference because it is a child feature of the feature `Nodes` according to the feature model. The same holds for each created target element in our subject product lines which would be recorded as a context element: The annotation of primitive types (i.e., the root feature), for instance, is implied by any annotation that is mapped onto an attribute. Similarly, the annotation of general types is implied by the specific classes and method parameters which are more specific than the annotation of the method itself do not occur.

Example Trace Completeness Ex. 10.4.1 sketches and discusses the influence of the feature model and the trace completeness on the correctness in an abstract way:

Example 10.4.1: Trace Completeness and Annotation Interplay

On its left hand-side, Fig. 10.4.1 depicts three different feature models, F_a , F_b and F_c , comprising the same set of three features, R , A and B , and two equal STT model instances on its right hand-side. Two traces of different completeness levels connect the source with the target model elements. The scenarios on the right side show the application of two transformation rules, `Rule1` and `Rule2`. `Rule1` transforms source elements of type `Src1` into two target elements of types, `Trg1` and `Trg2`. `Rule2` requires the second target element of type `Trg2` to create a third target element of type `Trg3` for a source element of another type `Src2`. While the generation-complete trace, depicted in the upper part of the figure, records only the created target elements, the complete trace, depicted in the lower part, records the target element `t2` as context element of `Rule2` additionally.

The trace-based propagation maps the annotations A onto $t1$ and $t2$ regardless of the trace completeness. It will map the annotation B onto $t3$, if the trace is generation-complete because the propagation regards only the source element of `Rule2`. Since the complete trace records the element $t2$ (annotation A) as context element of `Rule2` in addition, the annotation which is mapped onto the target element $t3$ is the conjunction of the annotations of the source and context element: $A \wedge B$.

Depending on the feature model constraints, the annotations computed based on generation-complete and complete trace information affect the presence of the target element $t3$ in derived variants differently: If the feature model expresses constraints as in F_a or F_c , the annotation $A \wedge B$, computed based on the complete trace, will not convey the same visibility as the annotation B , computed based on the generation-complete trace.

If A and B are optional features without any other constraint as in F_a , a configuration may deselect A and select B . The transform-filter variant derived from the multi-variant model which was created based on the *generation-complete* trace will contain the target element $t3$. Although the derived source variant comprises $s2$, `Rule2` cannot be executed in this situation because it requires the presence of $s1$ and $t2$. Since the source element $s1$ will not be part of the derived source variant, the single-variant transformation cannot create the target element $t2$. Therefore, $t2$ will be missing and prevent the single-variant transformation to create $t3$. Consequently, the filter-transform and the transform-filter variant will not be equivalent and, thus, violate commutativity. Assigning the annotation based on the *complete* trace will prevent this situation because the computed annotation prohibits to include $t3$ in the transform-filter variant.

Similarly, it is possible to select B but not A in F_c . Then, each element the annotation B is mapped onto will be present in the derived variants but `Rule2` will not be executed in the single-variant transformation due to the missing source and target elements, $s1$ and $t2$. Thus, for the same reasons as with annotations conforming to F_a , the same set of annotations will violate commutativity if the feature model constitutes equally as F_c .

Finally, if the feature model constitutes as F_b , the annotations will suffice to satisfy commutativity regardless of the trace completeness. According to F_b , B implies A , such that either only A is selected or A and B , are both selected. As a consequence, either only $s1$ is present and transformed or $s2$ is present in addition. If $s2$ is present, the context element of `Rule2` will be present, too. Therefore, $t3$ is included in the filter-transform variant whenever it is present in the transform-filter variant. Thus, this scenario satisfies commutativity.

Consequences As the transformations, the annotations and the feature model in our subject product lines exhibit similar relationships and constraints as the mapped annotations and feature model F_b in Ex. 10.4.1, the generation-complete trace suffices to satisfy commutativity. Another reason why a generation-complete trace may suffice is that either no context elements are required in the transformation rules (such that the generation-complete and the complete trace persist exactly the same information) or because all relevant source elements are recorded in the trace: Firstly, to create target models which comprise differently typed elements which require the presence of another element to be created, the transformation definition needs access context elements. Thus, it is unlikely that a transformation definition can be executed *without* accessing context elements at all. Secondly, it may suffice to record *all* source elements, which are relevant for creating the target element. This should include source elements which were responsible for creating the context elements which are missing in the generation-complete trace. Then, the information which is redundantly present in a complete trace would suffice. In Ex. 10.4.1, it would suffice to add a link originating from the trace element `tr2` targeting $s1$ because it conveys the same annotation information as the context element $t2$ (referenced by the `ctx` edge). However, the traces that we examined either stored the entire information (all source and all context) elements or only the single source element and no context element.

Incomplete Traces As opposed to the sufficient information in generation-complete traces, *solely* the information persistent in *incomplete* traces does *not suffice* to satisfy commutativity. In neither subject system it is possible to satisfy commutativity based on this information. While we

do not evaluate the effect of an incomplete trace in isolation (i.e., the model filter may influence the result), we can draw the conclusion of its insufficiency from the fact that the annotations computed by the completion strategies based on the annotations assigned by the incomplete trace propagation are also not able to satisfy commutativity. Since we assume that the annotation completion does not worsen the commutativity result⁹, we conclude that the propagation without completion strategy (i.e., based on the incomplete trace only) cannot satisfy commutativity, too.

Trace Granularity As a second point, the trace granularity postulated by the computational model requires traces that record information at a level which is as fine-grained as the mapping and filter mechanism (c.f., Prop. 5.3.8). The mapping model as well as the filter of the tool Famile are able to define and process annotations which are mapped onto the structural features of objects. However, none of the traces which resulted from transformation engines we examined records corresponding elements at this trace granularity. *Consequently, we would expect that the coarse-grained trace causes a violation of commutativity (HG1).*

In contrast to this hypothesis, the results of the evaluation indicate that the missing fine-grained information does not affect the correctness of the assigned annotations negatively. On the one hand, none of the structural features of the source objects carries an annotation which is more specific than the annotation of the source object. Thus, the structural features are present in any derived variant where the source object is present and therefore, a more fine-grained annotation does not have to be propagated to the target mapping model.

On the other hand, if the mapping mechanism allows for fine-grained annotations and none are provided, the Famile model filter will consider them *incomplete* and assumes the annotation `true`. Thus, the structural features of a source object are present whenever the object is present in the source model. Alternatively, a completion strategy may compute the target annotations in their entirety assigning the annotation of the containing object to the structural features. As a consequence, in effect, insight of hierarchical filters the coarse-grained trace will affect the correctness negatively only if the source multi-variant model exploits the possibility to assign more specific annotations to structural features.

Properties of Transformation (Rules) Furthermore, we would expect a negative impact on the accuracy, if the transformation was performed *in-place, incrementally*, was not *rule-based* and if the transformation *rules* were either *not functional, monotonic* or *local*. To reduce these effects and isolate the influence of the trace completeness, we only consider rule-based out-place transformations composed of monotonic and functional rules in batch mode.

However, not all of the rules in the transformations that we employed are *local*. The Ecore2Java transformation, for instance, employs positive application conditions to create different elements for classes, where the structural feature `interface` is set `true` and `false`. Similarly, the Ecore2SQL transformation creates either a column or an entire table for a given `EReference` depending on whether the reference is a containment reference, single- or multi-valued. Similar to the Ecore2UML transformation definition, the UML2Java transformation definition creates different elements (e.g., different contents of comments attached to a field declaration depending on the type of association).

Although some of the rules are non-local, not all of them affect the correctness of the propagation. Particularly, if the presence or absence of an `EAttribute` of a source object varies the transformation, it will not have an effect on the examined transformations because we only map annotations onto the objects. However, if the presence or absence of a referenced object might create two distinct elements, the propagation cannot guarantee commutativity.

Because of the assigned annotations of the subject product lines the transformation does not execute different branches in sight of missing or additionally referenced objects. For instance, in the examined product lines whenever a class inherits from another class, the annotation of the

⁹ The annotation completion is based on already existing annotations which convey exactly the same information as present in the source model. Thus, it does not invent new behavior but propagates the same information of the source model to further dependent elements. The model filter further prevents that elements carrying a too broad annotation (e.g., the root feature) to remain in a variant without proper container and removes these elements. Thus, the completion strategy cannot exacerbate the commutativity result.

specific class implies the annotation of the general class. Therefore, it is not possible for the specific class to exist without the general. Similar annotations prevent non-local rules to provoke the execution of a different program path.

On the whole, the properties restricting the *execution* of the transformation definition do not violate the computational model but the transformation rules violate *locality*. However, either no annotation is mapped onto elements provoking non-locality or the annotations prevent the violation to happen.

Completion Strategies As a second point, we examined the effect of the completion strategy. Firstly, we would expect that the completion strategies *establish a multi-variant model which is completely annotated (HC1)*. Secondly, we would expect that the *container strategy is beneficial whenever a transformation rule creates a hierarchy of elements contained in the pivot model element which carries an annotation (HC2)*. Thirdly, we would expect that the *combined strategy achieves the highest accuracy whenever the target model forms a hierarchical structure (without crossreferences, relevant for the presence of an annotated element) (HC3)*.

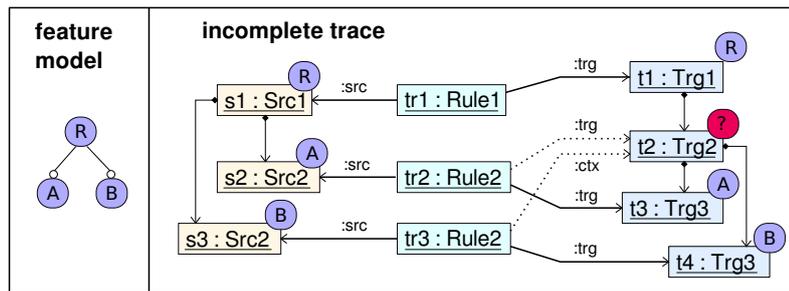
In the subject product lines, which were transformed with BXtend and corresponding trace information together with a completion strategy, the resulting mapping models are annotated *completely* regardless of the chosen completion strategy. Thus, the findings confirm expectation HC1. The result is a consequence of the design of the three completion algorithms: The algorithms iterate the elements without annotation either top-down or bottom-up to ensure the algorithm can retrieve the annotation for an element either from its container or from the contained elements. If the container and contained elements are not annotated, the algorithm will assign the root feature. Furthermore, we observe that the container strategy makes a difference when the model is structured mostly hierarchically whenever the container is the pivot element stored in the trace and the target elements not recorded in the trace are contained directly or transitively in this pivot element. For example, in the sql model most of the created pivot model elements (**Table**, **Column**) contain objects of type **Annotation**, which do not receive an annotation by the trace-based propagation due to the incomplete trace information. However, the annotation of the last annotated container is specific enough to satisfy commutativity. Therefore, compared to the UML2Java base version, about three times less (31.52% and 34.77% in the DBC and Graph product lines, respectively) annotations of all target annotations in the respective model have to be repaired in the Ecore2SQL transformation according to the actual error. In a similar way, the repair ratio which regards the trace information per incomplete mapping (c.f., Table 10.5) decreases when comparing the *base* with the *advanced* UML2Java transformation definition in the HAS model. While the model structure does not affect the actual error positively in the DBC and Graph product lines (because the relevant pivot container is annotated with the root feature), the hierarchical structure of the HAS with more specific annotations mapped onto the packages decreases the number of annotations that have to be repaired in relation to the number of annotations assigned by the trace propagation. In summary, while we would expect a positive effect of the container completion strategy in hierarchical models, the effect strongly depends on the kind of annotation that is mapped onto the container which is considered for computing the missing annotations. Thus, the evaluation cannot confirm expectation HC2 without restrictions.

Thirdly, we would expect that the combined strategy achieves the highest accuracy of all completion strategies. However, the findings show that in the subject product lines it does not achieve a better effect than the container strategy. As discussed above, this is mostly caused by one of the children annotations which is the same as the annotation of the container element. Therefore, the accuracy boils down to the accuracy of assigning the parent annotation.

Despite this effect in the chosen product lines, Fig. 10.4.2 sketches a situation where the combined strategy can pay off. Ex. 10.4.2 explains the situation in detail.

Example 10.4.2: Combined Strategy Effect

Fig. 10.4.2 demonstrates a situation where the feature model consists of a root feature R and two optional features A and B (as representative of a similar sub-group of features in a larger feature model). The source model comprises three elements: the source element

Figure 10.4.2: Example of *combined* completion strategy with positive effect.

$s1$ of type *Src1* (annotation A) contains the elements $s2$ and $s3$ of type *Src2* which are annotated with the features A and B , respectively. Two transformation rules are applied in sequence to create the four target elements depicted on the right side. Particularly, the second rule *Rule2* creates a target element of type *Trg2* whenever no appropriate container exists for the pivot target element of type *Trg3* it creates.

The incomplete trace records three trace elements for each rule application and references the source and target elements with solid arrows. The dotted references mark elements which are created either additionally in the rule application (i.e., $t2$ by *Rule2*) or serve as context element ($t2$ by *Rule3*). The element $t2$ misses an annotation because it is not referenced as target element by the incomplete trace. If the container strategy was applied, the annotation R would be mapped onto the target element. As a consequence, element $t2$ would be present in every derived variant although neither the target elements $t3$ and $t4$ were present nor the source element $s2$. Thus, the container strategy computes an annotation which would violate commutativity.

If instead the combined or contained strategy computed an annotation for $t2$, the resulting annotation would either be $R \wedge (A \vee B)$ or $(A \vee B)$, respectively. In contrast to the annotation computed by the container strategy, the annotation computed by both, the contained and the combined strategy is specific enough to satisfy commutativity in this scenario. The element $t2$ will only be present in a derived variant if either the element $t2$ or $t3$ are present in this variant, too.

Consequently, Ex. 10.4.2 emphasizes that the combined strategy is beneficial in a multi-variant model where the element missing an annotation contains annotations which are optional and which are more restrictive than the annotation of its container. Although the transformation of packages and classes in the advanced UML2Java transformation assumes the same pattern, the combined strategy cannot achieve a more accurate result than any of the other strategies due to the annotations assigned to the contained elements as discussed above. A similar model element constellation occurs, for instance, in the backward transformation of a benchmark for bidirectional incremental transformations [Anj+20] which converts a person register into a family register. We examined this transformation to show the benefits of the combined completion strategy based on a small theoretical product line example [GW18c; GW19c]. Thus, we state that – even if the examined subject product lines do not confirm our expectation HC3 – the combined strategy may increase the accuracy in situations equivalent to the one abstracted in Ex. 10.4.2.

II Answers to Evaluation Questions

Based on the explanations and discussion of the results of the trace-based propagation, we answer the evaluation questions as follows:

EQ1 a) To what extent is commutativity achieved? The results show that reused single-variant transformations which propagate annotations based on *generation-complete traces* are able to *satisfy commutativity*. Although this is partly caused by the structure of the domain model, the assigned annotations and the transformation rules, we do not trim any annotation on purpose to

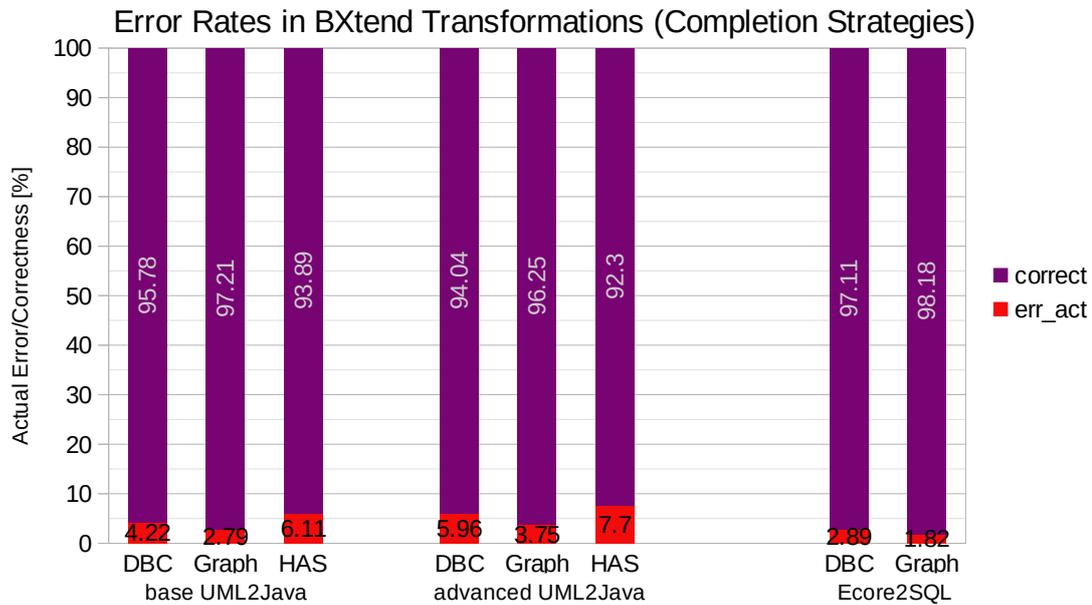


Figure 10.4.3: Statistics of correct vs. repaired annotations.

achieve the result but reused the Graph and HAS product lines as they has been designed before and created the DBC product line for demonstrating concepts in the course of this thesis. Thus, we draw the promising conclusion that a deviation from the computational model in form of using generation-complete traces, suffices in several transformation scenarios to achieve commutativity.

EQ1 b) How much manual effort has to be invested to repair wrong annotations?

The actual error states the minimum amount of annotations which have to be repaired manually. If generation-complete traces serve to compute annotations, *no manual* actions will be required. In contrast, Fig. 10.4.3 provides – from left to right – an overview of the computed actual error rates in the base and advanced UML2Java transformation as well as in the Ecore2SQL transformation where annotations are computed based on an incomplete trace and a completion strategy. The figure visualizes that in every subject product line less than 10% of all possible target annotations have to be repaired which corresponds with the manual effort that has to be invested. Without a propagation approach *all* target elements have to be annotated manually.

EQ2: How much manual effort is saved? In other words, answering EQ1 b allows to answer EQ2, how much manual effort is saved. As stated as answer to EQ1 b, less than 10% of the annotations have to be repaired. Consequently, more than 90% of the annotations are computed correctly and assigned correctly even if the trace is incomplete. Therefore, also 90% of the manual annotation effort is saved because potentially all target annotations have to be assigned manually without the proper tool support. In case of a generation-complete trace, the manual annotation effort is saved completely in the subject product lines which we examined.

EQ1 c) Which completion strategy achieves the highest accuracy? In the examined subject product lines executed with the three distinct transformation definitions, the three completion strategies achieve the same accuracy.

On the one hand, this effect is caused by the Famile model filter, which will only keep an element if its container is also present. The combined strategy assigns the root feature or the one of the container to elements which do not possess children elements.. While the annotation would provoke an invalid derived variant when a flat filter is used, the Famile filter prevents this situation to occur. However, as a consequence, the container and the contained strategy provoke the same effect if an element missing an annotation does not possess children. Similarly, the contained

strategy provokes the same effect if one of the children is annotated with an equally broad or even broader annotation than the parent of the model element missing the annotation. Since the combined strategy can only become as accurate as the more accurate of both the container and the contained strategy, it depends on their effect. Consequently, only if the children are annotated with a more specific annotation than the parent of the element missing the annotation and all three of them are annotated with an optional annotation, the combined strategy will achieve the highest accuracy.

10.4.2 Aspect-Oriented Propagation

Similar to Sec. 10.4.2, I, this section first summarizes the findings of conducting the aspect-oriented propagation with the language Xpand. The first part not only provides a short summary but also discusses the effects of the computational model whereas the second part answers the evaluation questions.

I Summary of Results

The M2T transformations which employ the aspect-oriented propagation approach, are able to satisfy commutativity if they conform to the properties postulated in Sec. 8.3.3. This is the case for the MoDisco2Java transformation which does not dynamically compute the set of elements matching an invoked rule. In contrast, the Ecore2Java transformation may violate commutativity since the transformation deviates from the computational model. The following paragraphs first offer an overview of the results and continue with analyzing details of how the transformation definitions and the subject systems influenced the correctness of the propagated annotations.

In more detail, the results of conducting the commutativity evaluation with the two Ecore models of the DBC and the Graph product lines, demonstrate that the transformation properties and the annotated multi-variant transformation influence the correctness of the propagation. Again, the Ecore2Java transformation deviates from the computational model inasmuch as one rule invokes another rule on linked (not necessarily contained) objects of the source node. Despite this violation of the computational model, given the Graph product line model as input, the transformation satisfies commutativity and the error in the DBC product line requires to change the annotation of four text fragments, in total. Furthermore, the MoDisco2Java transformation which does not violate the computational model satisfies commutativity to full extent by applying the same advice. Consequently, the results indicate that the properties of the computational model sufficiently reflect the implications on the correctness of a transformation scenario.

Computational Model Commutativity may be violated whenever the *transformation execution*

- applies the same rule more often than once to the same source node (Prop. 8.3.1),
- modifies the source model (Prop. 8.3.2),
- does not terminate due to infinitely iterating or recurring rule applications (Prop. 8.3.3 and Prop. 8.3.4).

Furthermore, the *rules* need to be

- functional (Prop. 8.3.5),
- local (Prop. 8.3.6),
- monotonic (Prop. 8.3.7) and
- need to preserve the order defined for the target elements (Prop. 8.3.8).

Similarly, the *aspect* needs

- to preserve the already existing order of the created fragments (Prop. 8.3.9) and
- to assign hierarchical annotations (Prop. 8.3.10).

Finally the computational model relies on the usage of coarse-grained annotations (Prop. 8.3.11) as well as a hierarchical filter which preserves the order of the text fragments (Prop. 8.3.12).

Termination With respect to the properties of the transformation engine, we employ Xpand transformations executed via MWE2 workflows to implement the aspect-oriented propagation. In

the first place, the language capabilities allow to define a transformation rule which can invoke itself and can be invoked on the input object. Thus, it violates the necessary property for the transformation rules to apply the same rule only once to the same source object (Prop. 8.3.1) in order to guarantee termination. The remaining property of preserving the source model and not modifying it by applying a rule is satisfied by the default properties of Xpand transformations. However, the possibility to execute almost arbitrary Java source code (only static methods), can provoke modifying the source model. In summary, although Xpand violates the three criteria, uniquely matching a source node with the same rule, preserving the source model and terminating each execution branch with a 'leaf' rule, the transformation definitions which we examined adhere to these properties and therefore, terminate the execution.

Transformation Rules In order to ensure that the generated single-variant source code can be embedded into the multi-variant source code, the rules further need to be functional, local, monotonic and preserve the order of the generated text. By design, Xpand rules are functional, monotonic and executed in the ordered defined in the specification. However, the rules are not local and injected advices can prevent text created by the original rule from being written to the target file. As a consequence, we expect that correctness is violated when rules are defined in a non-local way and when advices prevent the inclusion of original text in the multi-variant target files.

The examined transformation definitions do not involve advices which would prevent text generated by the original text production from being included in the created file. Therefore, correctness is not threatened and the specifications conform to Prop. 8.3.9, which fosters that the advice preserves the execution and the target tree. In contrast, the Ecore2Java specification comprises rules which collect types used in parameters and structural features of a given `EClass` to invoke another rule on them for creating their import statements. Furthermore, several define-blocks access information of structural features (e.g., inheritance of class when `eSuperTypes` comprises elements) of the matched input element and vary the transformation based on their values. However, varying the text-production based on the structural features of the matched source object does not influence the correctness in the examined scenarios because we postulated only coarse-grained annotations at the level of objects and the input multi-variant models adhere to this property. Consequently, only the deviation based on executing other rules which get annotations assigned threatens commutativity in this context. While this is the case in the DBC product line for only four annotations that are too broad, the situation does not occur in the Graph product line so that the latter created multi-variant models is annotated correctly.

Furthermore, we would expect a threat to the correctness when the computed annotation for the target element is not combined with the annotation mapped onto the containing block. Since the annotation is computed by the advice which only receives the source object as input, information about the outer block is per se not available to the advice. Therefore, our implementation accesses the parent object of the input object and looks up its annotation to combine it in a conjunction with the annotation of the input object. Although this does not have to be necessarily the same annotation mapped onto the surrounding text, the computed annotation do not harm correctness. Last but not the least, as discussed above, we employ only mapping models with coarse-grained annotations as input models. Furthermore, the Famile model filter served to derive model variants and a custom preprocessor to derive text variants. Since both filters work hierarchically, we did not expect a threat to correctness introduced by them and the results corroborated this expectation.

II Answers to Evaluation Questions

The results of the examined M2T transformations allow to answer the evaluation questions as follows:

EQ1 a) To what extent is commutativity achieved? The results demonstrate that the reused single-variant transformation which transforms core elements of a Java MoDisco model into Java source code satisfies commutativity to full extent. On the contrary, the Ecore2Java definition satisfies commutativity to full extent only for the Graph product line whereas four

annotations were assigned erroneously by the advice when generating source code for the DBC product line. Thus, although wrong annotations were introduced, less than 2% of all generated annotations had to be repaired to satisfy commutativity.

EQ1 b) How much manual effort has to be invested to repair wrong annotations? As explained in the answer of **EQ1 a**, it was necessary to repair less than 2% of the automatically created directives. In absolute numbers *four* annotations had to be refined in the text generated for the Graph product line in the Ecore2Java transformation only. Therefore, we argue that even if the transformation rules violate the computational model for the aspect-oriented propagation, the repair effort remains very low compared to the number of annotating directives that are automatically and correctly assigned.

EQ2: How much manual effort is saved? As a consequence of answering **EQ1 b**, more than 98% of the directives created in the Ecore2Java transformation for both input product lines are assigned correctly. Consequently, the effort of the product line developer is reduced significantly compared to stating each directives manually. In fact, more additional lines for directives and other comments are created automatically as are for the original text production as indicated in Table 10.8 and Table 10.9 ¹⁰.

EQ1 c) Which completion strategy achieves the highest accuracy? This question cannot be answered adequately for M2T transformation because they do not employ a completion strategy.

10.4.3 Threats to Validity

On the whole, the chosen subject product lines and the observed results demonstrate that the correctness of the annotations depends on the kind of the computed annotation as well as on the model filter, the model structure and the trace completeness. Therefore, the remaining factors as well as the chosen subject product lines may introduce threats to validity of the experiment results and the drawn conclusions. The following discussion in this section is based on the descriptions of threats to validity by Wohlin et al. [Woh+12], who distinguish four kinds of threats: *conclusion* and *construct* validity as well as *internal* and *external* validity.

The conclusion validity considers the relationship between the evaluated solution strategy and the outcome of the evaluation. It requires that there is a statistical (significant) dependency between both. As an example, *fishing* for a product line and transformation definition which satisfy commutativity completely despite violating the computational model would be an example of conclusion invalidity.

Similarly, construct validity regards the causality between the evaluated solution strategy and the observed outcome. If causality is given, construct validity is concerned with representing the cause in the input and the effect in the outcome with adequate constructs. For instance, measuring one type of value only or not defining the input at an adequate level may threaten the evaluation design.

While internal threats regard the selection of subject systems with respect to being adequate to indicate a causal relationship between the solution strategy and the result, external threats regard the fact to what extent the results transfer to input subjects outside the setting of the experiment. Again, both, the internal and external validity, can be threatened by the *selection* of the input subjects. Furthermore, for instance, grouping the input and the potential of it being influenced by maturation or its history may threaten internal validity whereas the interaction of the history or setting with the examined solution strategy may threaten external validity.

¹⁰The amount of automatically created lines of codes adds the numbers of the columns *#directives* and *#other comment* which is in each transformation scenario higher than the number of originally created source code (column: *#source code*).

I Conclusion Validity

As a first threat, this section discusses threats to conclusion validity [Woh+12] which can be introduced by

- (1) low statistical power,
- (2) violating the assumptions of the employed statistical tests,
- (3) “fishing” for adequate subjects and the error rates in statistical tests,
- (4) the reliability of the measured values and
- (5) the reliability of the solution implementation,
- (6) disturbing the experiment with random irrelevancies and
- (7) a random heterogeneity in the subject systems.

Replication of Results Since we do not conduct statistical tests, we cannot introduce threats to the statistical power (1) and violate the test assumptions(2). Furthermore, the measured values and the solution implementation are reliable with respect to producing a deterministic result (4). Conducting the experiments again will result exactly in the same measured values if the input product lines and transformations are not modified beforehand. Thus, also the error rates do not diverge in executing the tests repeatedly. Similarly, the implementation of the transformation definitions does not change unexpectedly so that the examined effects remain stable in iterating the experiment (5). In addition, disturbance provoked by outside factors such as noise cannot threaten the experiment since we evaluate correctness only (6).

Subject Product Lines Regarding fishing (3) and the heterogeneity of the subjects systems (7), the number of subject systems does not represent the entire universe of potential model-driven product lines and transformation definitions:

We picked a set of input product lines which covers a significant variety of modeling concepts. The domain models are complementary, which allows to cover several constellations in this exploratory study. The selection of product lines with three diverse feature models allows to discover three small- to medium-sized product lines where two impose a lower and the HAS a of valid configurations which challenges the commutativity evaluation. The feature models vary inasmuch as they incorporate requires- and excludes-dependencies, OR- and XOR-groups and mandatory as well as optional features. Thus, they cover each possible type of constraint that can be expressed with the (non-cardinality-based) feature models we presume in our approach.

Similarly, the kind of mappings that are applied range from annotating a model element with a single (optional) feature to including positive and negative selection conditions (e.g., `Directed` vs. `not Directed` edges in the Graph product line) and expression over the features, such as `Wireless` or `Bluetooth` or `Pet and Person and Domesticated`. As an example, the latter of which is mapped onto the bidirectional reference between the class `Domesticated` and the class `Person` in the DBC product line to realize the ownership of a pet for a person and the expression involves features of different feature groups.

However, it must be considered that the subject product lines cannot be more diverse than the transformation definitions: it is not possible to transform a state chart, persons model or finite automaton with a specification that expects class models as input. Therefore, we employed Ecore and UML models as input only because we executed transformation definitions which expected these types of models as input. Despite the fact that this selection may seem limited, we argue, on the one hand, that in this way we reduce the threat of heterogeneity and thereby reduce the potential of large variation. On the other hand, we expect that class models are commonly used to design a product line. Therefore, they require an automated propagation of annotations more necessarily than artificial product lines which pass a specific transformation benchmark (e.g., converting a person into a family register [ABW17]).

In summary, while the set of different product line and transformation definitions could have been larger, we expect that new insights on the correctness of the propagated annotations gained by executing further transformations would be marginal because the constellations that we examined are diverse. Furthermore, we shortly discussed that the correctness might have benefited from conducting the examination based on smaller product lines which would have fitted the design

of the completion strategies better, such as strict hierarchical models (no crossreferences) with annotations attached to children elements which are not as coarse-grained as the one of the parent element of an element missing an annotation.

II Construct Validity

As second threat, this section deals with the validity of constructs which is concerned with generalizing the results to the concept of the solution strategy. According to Cook and Campbell [], constructs are threatened by

- 1) inadequately explicating the constructs before the execution,
- 2) bias due to single operation,
- 3) bias due to a single method,
- 4) inherent levels of constructs and confounding constructs,
- 5) interacting solution strategies,
- 6) interacting of the experiment with the solution strategy and the
- 7) restricted possibility to generalize across the constructs.

In addition, when a group of persons is subject to the evaluation, social threats might be introduced which we do not consider.

Input Selection We reduced the threats to the inadequate explication of constructs before the execution (1) by scrutinizing them in Sec. 10.2.2 and Sec. 10.2.3. Furthermore, we do not employ only a single subject system but a diverse set of up to three product lines and up to four M2M transformations specifications and two M2T specifications to examine the effects on the correctness. Thus, we decrease the influence of the single-operation bias (2). Similarly, we do not employ a single measurement value in the M2M transformations but consider at least three error values (err_{abs} , err_{sev} and err_{act}) to assert the implication of the result. This decreases the influence of the single-method threat (3).

Influence of Model Filter While confounding constructs and levels of constructs do not exist in the subject systems that influence the results (4), there may be an interaction with the model filter which harms the expressiveness of our findings (5). We keep the influence of the filter as realistic as when employing the tool Famile to build a product line where the default filter would work hierarchically and include elements without an annotation whenever their parent element is present, too. Particularly, this filter is present on the source mapping models as well and ensures that the structural features of the objects do not have to be annotated in order to be present. For instance, therefore the name of each source class does not have to be annotated with the annotation mapped onto the class. Accordingly, we employed the same filter on the target side to keep this effect comparable.

However, the model filter directly influences the meaning of the completion strategies which were designed for general purposes where no such filter is available. On the one hand, it may 'repair' annotations that would be too broad and keep elements in the model when their parent does not exist. On the other hand, it prevents to detect differences between the propagation strategies.

III Internal Validity

Internal threats can be caused by single groups or multiple groups [Woh+12]. As the evaluation does not conduct experiments with persons, *social threats* do not play a role in our experiments as neither do multiple group threats because there is no maturation effect among them. However, we can detect some of the following single-group threats:

- (1) influence of history
- (2) influence of maturation
- (3) repeated testing
- (4) instrumentation
- (5) statistical regression
- (6) selection
- (7) mortality

(8) ambiguous direction of the causality.

While history (1), maturation (2), statistical regression (5), and mortality (7) do not apply to our test settings, repeating the tests does not change the outcome as there are no learning or outside noise involved (e.g., CPU power, threading, etc.) as it would be, for example, in measuring the execution *time*.

In contrast, the instrumentation (4) and selection (5) influence the results. As we discussed in Sec. 10.4.3, I, the selection of subject product lines may influence the results but we employed a diverse set of input models for realistic transformations. Regarding the selection of transformation definitions, the following points have to be respected, in addition:

Transformation Definition Selection Sec. 10.2.3 presents different M2M and two M2T transformation definitions which we use for evaluating the annotation propagation mechanisms. There is no denying the fact, that a plethora of further transformation scenarios converting instances of other metamodels exists. For instance, the ATL zoo [All22] offers a large collection of various ATL transformations which are compiled with the default ATL compiler (i.e., no persistent trace or execution model). Similarly, the Benchmarx framework [Anj+20] collects a multitude of challenging incremental bidirectional transformation scenarios.

Further transformations are not examined for two reasons: Firstly, existing transformation definitions frequently only allow to transform instances of simplified metamodels for which no product line models exist. As an example, the ATL zoo [All22] collects more than 100 specifications which, however, for instance, the class to relational database transformation definition is defined over simplistic source and target metamodels for which we are not aware of product line models. Consequently, it would be necessary to artificially create feature models and annotate domain models for executing multi-variant transformations. For the ATL transformations it would be further necessary to compile them with the EMFTVM compiler or to extend the transformation engine to persist the volatile trace information created during the transformation.

Secondly, the additional amount of knowledge gained from evaluating further transformations can be expected to be low. Our examined scenarios cover straight-forward transformations with mostly 1:1 or 1:n mappings, such as the Ecore2UML transformation, but also complex transformation scenarios which involve non-confluent rules, situations where (hierarchical) relationships cannot be mapped 1:1 or source and target elements do not correspond with an element in the respective opposite model.

As a consequence, the selection of transformation definitions allows to draw several conclusions on the accuracy of computed annotations. For rigorous testing, however, an automated testing tool which covers all dependencies in the feature model together with relationships between model elements of arbitrary metamodels could extract a more fine-grained result.

Instrumentation Influence Considering the evaluation results, it becomes obvious that the usage of the EMFCompare framework influences the measurement of the correctness strongly. While this framework produces the most accurate results when model elements are recognized with unique identifiers, we cannot rely on this comparison mechanism: The filter-transform and transform-filter variants need to be equivalent but they are not equal, meaning the identifiers would be different for equivalent elements. Therefore, we rely on a structural comparison which depends on the order of comparing model elements as it recognizes several mismatches among SQL models created with the QVT-O transformation but non in the same models created with the ATL/EMFTVM transformation. These models only differ in the order in which the tables and columns are integrated in the schema. Consequently, the error rates which rely on the differences counted by the EMFCompare framework, may be higher than when comparing the models by visual (i.e., manual) inspection. Therefore, we introduced the third measurement, the actual error, which however, requires to manually inspect and repair annotations. Accordingly, it required the author of the thesis to count the number of annotations which may introduce human error but on the whole reduced the error rate significantly. As the purpose of the automated propagation is to reduce manual effort and its benefit correlates with the number of annotations which have to be repaired thereafter, the actual error provides a realistic additional measurement.

As a second threat introduced by the instrumentation, the Famile model filter needs to be considered. Its effect is discussed in the part on construct validity.

Direction of Causality Finally, the direction of causality may be ambiguous. For example, it may be internal factors, such as implementation faults or the instrumentation, that cause an error or the error may be inherent to the problem.

On the one hand, as the trace-based propagation algorithm is formally defined and proven to be correct, the corresponding class which implements the algorithm in exactly the same way can be assumed to behave correctly. This is also confirmed by target models (Ecore2UML, Ecore2SQL based on generation-complete traces) which are annotated completely correctly. The commutativity evaluation confirms this fact as well as a manual inspection of the resulting annotations. On the other hand, as stated before, the model filter may influence the correctness. Without any annotation at hand, the filter pertains all elements in a derived variant. Therefore, it can only influence the derivation result in the presence of annotations. As its behavior is similar to the one of the container strategy, which completes annotations which are too broad, we can assume that commutativity is not caused by the filter but by the correctly computed annotations. Thirdly, the annotations assigned to the source model may affect the correctness. As explained in the first part, we selected realistic subject product lines without significant adaptations. We only ensured that they are annotated completely. However, it can be argued that developers map meaningful annotations onto the implemented artifacts [KDO14] and this is exactly the information that should be propagated by the contributions of this thesis.

In case of the aspect-oriented creation the model filter does not influence the transform-filter variant but a preprocessor instead. Similar as a compiler, if the preprocessor behaved incorrectly or annotations were assigned wrongly, no transform-filter variants would be created. Consequently, we can assume that commuting transformations result from correctly assigned annotations.

IV External Validity

External validity regards the experiment design and conclusions with respect to generalize them from the laboratory settings to apply them in practice. While field experiments, which we did not conduct, allow for greater generalizability in general, laboratory controlled experiments tend to be applicable in more restrictive scenarios [SF18].

External validity is threatened by the chosen subjects (1), and experimentation at the wrong time (2) or in the wrong environment (3) [Woh+12]. These represent the three major factors to interact with the solution strategy to generalize it.

While the history and the time (2) of conducting the experiments does not influence the validity of the experiments, the selection (1) of subject product lines and transformations was discussed before: The chosen subject product lines represent a realistic set of product lines which cover all constraints in the feature models and map diverse annotations on different model elements. Furthermore, the transformation definitions represent more complex but real-world-oriented scenarios such that the included patterns can serve to generalize the behavior for real-world applications. Thus, we argue that we decrease this threat of selecting adequate subjects (1) significantly.

Generalization of Propagation Concepts In contrast, the environment of the experimental setting might threaten the generalization possibilities (3). The employed M2M transformation engines are open source and developed actively but may not be the state-of-the-art in industry. In the same way, we cannot guarantee that the approaches work for every transformation language and every specification. Particularly, we cannot presume which trace-generating single-variant transformations are yet to be developed. However, we diversify the experimental evaluation and offer evidence of genericity by varying the transformation languages for the M2M transformations, the transformation definitions for different types of models and the product lines from small-sized and artificial to medium-sized and academic real-world simulations.

Additionally, we argue that the concepts of using a trace is general and can be employed even if no trace-generating transformation is available but, for instance, correspondences are expressed by the means of unique identifiers in another form of representation.

On the contrary, Xpand transformations are still supported in the Eclipse framework but are almost replaced by Xtend which however, removed the aspect-oriented capabilities. Despite this legacy technology, the concept of using an advice and propagating annotations by weaving them into the execution goes beyond the language and might be applicable in different settings in form of a replication.

V Further Remarks

Validity of Derived Variant In addition to the systematic analysis of the threats to validity, this section shortly discusses two further assumptions on the overall propagation approaches.

Validity of Derived Variant The evaluation focuses on creating commuting derived variants. On the one hand, there is no final validation of the derived source models to check whether they conform syntactically (and semantically) to their metamodel. However, it is ensured that the models can be saved as EMF `Resource` such that it is ensured that they do not expose dangling references. On the other hand, while we compare the source code variants syntactically, we do not examine whether they result in compilation and runtime errors. Different stages of the source code correctness in derived variants could be analyzed in addition, in a similarly staged way as suggested by Ratzenböck et al. [Rat+22].

10.5 Summary

All in all, this chapter presented the evaluation of the propagation strategies proposed in Chp. 6-8. Particularly, it examined to what extent the solutions satisfy the research objectives qualitatively and quantitatively. All solutions propagate annotations *automatically* and all, except for the propagation DSL, *reuse* existing model transformation technology. While the DSL approach is definition-specific, all concepts are language-independent¹¹ and the remaining ones definition-independent. Thus, they perform the annotation propagation *generically*.

For examining the correctness, the chapter introduced three diverse small- to medium-sized model-driven product lines and four types of M2M transformation definitions specified in varying trace-generating languages. The transformation served to investigate the effects of different trace completeness level on the accuracy of the propagated annotations. Moreover, it introduced two M2T Xpand transformations which served to evaluate correctness of the aspect-oriented propagation. The results give evidence that the aspect-oriented and the trace-based propagation both assign the large majority of annotations *correctly* and thereby reduce the manual annotation efforts significantly. Contrary to the postulations of the computational model, in all examined scenarios annotations propagated based on generation-complete trace were mapped onto elements completely correctly. In contrast, the propagation based on completion strategies resulted in some too broad annotation and they could not achieve a difference in accuracy. Particularly, in sight of completing annotations, however, the employed hierarchical model filter may influence the results. Similarly, if the computational model for aspect-oriented propagation is satisfied, the transformation will commute, too.

¹¹ The aspect-oriented approach was realized in Xpand and the bytecode instruction analysis based propagation employs the ATL/EMFTVM but the concepts can be replicated in other languages.

Part VI
Conclusion

Chapter 11 Resume

*One never notices what has been done;
one can only see what remains to be done*

Marie Skłodowska-Curie

~

The thesis contributes generic strategies to automatically propagate annotations by reusing existing model transformation technology without adaptations. The preceding chapters did not only derive them conceptually but also presented their implementation and an evaluation in diverse scenarios. To conclude this thesis, this final chapter summarizes the achievements of this work and reflects on them.

Firstly, Sec. 11.1 summarizes the contents by presenting a chapter-wise overview of the contribution and highlights the important insights. Furthermore, it reflects critically on the limitations implied by the design decisions. In contrast, Sec. 11.2 illuminates the benefits of the presented annotation propagation approaches, particularly, for research and industry. Finally, Sec. 11.3 sketches an outlook on future work which can build on and extend the propagation strategies, for instance to allow for synchronizing annotations.

11.1 Summary of Contribution

This concluding chapter reflects on the contributions of the thesis. Therefore, this section initiates with summarizing the contents chapter-wise, particularly the contributed novelties, of the thesis. Based on this recollection, Sec. 11.1.2 emphasizes the main strengths of the contributed propagation approaches whereas Sec. 11.1.3 discusses the impact of the design decisions.

11.1.1 Overview of Content

Background When recollecting the contents of this thesis, the introductory part served to motivate the necessity for propagating annotations in the model-driven development of a product line and gave an introduction to the problem solved in this thesis. Chp. 2 and Chp. 3 explained the disciplines, model-driven software engineering and software product line engineering, in more detail. Both chapters presented key techniques used in both disciplines which serves particularly readers who are not familiar with either discipline. Additionally, the first part of Chp. 4 summarized and classified existing techniques to realize model-driven product line engineering, the combination of both disciplines. In contrast, the second part of Chp. 4, Sec. 4.2 and Sec. 4.3, categorized the related work of propagating annotations in MDPLE approaches. The latter study forms part of the contribution, too, as it analyzes how to categorize the approaches and investigates the presence of multi-variant model transformations in existing MDPLE tools.

Summary of Contribution The remaining chapters presented the contributions of this thesis combined with delimiting them from closely related work on the specific approaches. This includes

- a *study of related work* on keeping annotations consistent across model-driven product line artifacts (i.e., mostly models) and on treating variability in model transformations in Sec. 4.2 and Sec. 4.3, respectively.
- an informal description and *formal definition of trace-based propagation* in Chp. 5 and Chp. 6, respectively.
- *alternative strategies* in case transformation scenarios violate the computational model of the trace-based propagation, including
 - a bytecode instruction analysis (Sec. 7.2),
 - annotation completion strategies (Sec. 7.3),
 - reconstructing information about corresponding elements (Sec. 7.4) and
 - preserving manual annotations from being overwritten in incremental transformation executions (Sec. 7.5)
- the informal and *formal description of employing an aspect-oriented approach* to generate target text representations (to derive source code variants) in Chp. 8
- a prototype of an *implementation framework* to realize multi-variant model transformations in Chp. 9
- an *evaluation of trace-based propagation* in academic product lines in situations which may violate the computational model in Sec. 10.3.1-Sec. 10.3.4. This includes
 - the design and modeling of a demonstrative model-driven product line (DBC)
 - an Ecore2UML transformation specified in ATL/EMFTVM
 - Ecore2SQL transformations specified in QVT-O and ATL/EMFTVM
- an evaluation of aspect-oriented propagation in academic product lines in Sec. 10.3.5. This includes:
 - an Ecore2Java transformation specified in Xpand
 - a MoDisco2Java transformation specified in Xpand

11.1.2 Consequences

The summary of our contribution entails the strengths of this thesis:

Pragmatic Solutions The thesis examines the field of propagating annotations in a *reuse-based* way and offers a set of solutions to diverse realistic scenarios. Particularly, the thesis offers *pragmatic and widely applicable solutions* to propagate annotations. These solutions are generic inasmuch as they do not foster to use a specific transformation language or mapping mechanism¹. We sketch how to automate the extraction of trace information in situations where this information is unavailable, for instance, by employing model matching techniques and thereafter the propagation DSL. Furthermore, the aspect-oriented approach demonstrates an alternative to propagate annotations while executing the reused model transformation. Although our derived computational model considers the creation of text (represented as trees), the design of the generic advice may also be applicable in generating models. In case an aspect-oriented approach is unavailable, we recommend to employ a trace-based propagation and to reconstruct the traceability information.

Holistic Automation of Propagating Annotations The summary further shows that the thesis does not only develop some tool for propagating annotations in one scenario but the problem is addressed **holistically**. For the trace-based propagation, the bytecode instruction analysis and the aspect-oriented propagation we present the theoretical aspects including closely related work, continue to formalize the solution, and demonstrate the implementations. The provided formal computational models define in which situations the trace-based and the aspect-oriented propagation may compute correct annotations. The subsequent evaluation further confirms most of the properties of the computational models. Surprisingly, in the examined scenarios, however, the generation-complete trace (together with a given hierarchical model filter) suffices to propagate annotations completely correctly. This fact indicates that the specificity of the annotations mapped onto the source model as well as the filter mechanism impact the correctness positively.

Reduction of Manual Efforts A further result of the evaluation affirms that our proposed strategies reduce the manual efforts significantly. The propagation based on generation-complete traces annotates the target model correctly in its entirety. Similarly, the completion strategies which compute annotations for target elements without corresponding trace elements together with the trace-based propagation assign more than 90% of the annotations correctly, thereby *reducing the manual efforts* of assigning annotations significantly. Furthermore, if the reused model transformation adheres to the computational model for aspect-oriented propagation, all annotations are computed correctly. In case of violations, in the examined transformations more than 95% of the annotations were mapped onto source code fragments correctly.

11.1.3 Design Decision

In developing the propagation mechanisms to address the research objectives, we made assumptions and design decisions which influence and potentially limit the applicability of the concepts but also represent a generic and unique approach.

To recapitulate, the main design decisions assume

- D1** an *annotative* model-driven product line development.
- D2** element-wise mapping representations.
- D3** constrained variability.
- D4** a proactive development.
- D5** approaches which neither extend the *syntax of existing transformation languages* nor manipulate existing *transformations engines* to extend them with modified execution semantics.
- D6** the *commutativity criterion* as correctness criterion.

¹ The aspect-oriented approach is a language-independent concept and the concept of analyzing bytecode instructions is language-independent, too.

Accordingly, questions may arise whether 1) it is a realistic setting to assume a proactive development of an annotative product line with element-wise mappings and 2) the research objectives are too restrictive to support a broader set of software product line engineering approaches.

Element-Wise Annotative MDPLE The answer to the first question centers around two aspects:

Firstly, annotative product lines with element-wise mappings are a straight-forward way to integrate variability in software [Ape+13]. Preprocessor directives are mapped onto single source code fragments and thereby realize an element-wise mapping. Similarly, in MDPLE applying annotations onto model elements can be accomplished by employing mapping models which offer the benefit to separate the concerns of modeling and annotating.

Instead, applying feature-oriented programming requires either a 1:1 mapping of one feature onto one module or to maintain crosscutting features with additional glue code or further non-trivial dependencies between the modules. Similarly, delta-oriented programming requires either the definition of delta operations bundled in modules or their derivation from edit operations as well as adequate composition sequences that generate a well-formed variant. Therefore, we decided to focus on the practical setting of annotative product lines.

Secondly, it may be true that frequently product line development initiates with one product which is enriched with further functionality. Thus, reactive or extractive approaches may be required by industry. However, once a product line is extracted, it can be maintained in a proactive way. Then, the maintenance of annotations will remain a manual task if no technique, such as the annotation propagation mechanisms contributed in this thesis is available. In addition, the *automated* propagation of annotations may mitigate the obstacles and doubts which hinder practitioners to develop model-driven product lines from scratch.

Research Objectives The research objectives postulate automation approaches which reuse existing model transformation technology without modifications and which are generically applicable and compute annotations correctly. In fact, correctness (or accuracy) may decrease the increase of genericity. Thus, it would have been possible to define a solution, such as the propagation DSL, which allows for transferring annotations from UML class models to created Java MoDisco models only, by declaring each correspondence at a fine-grained level. Similarly, the target mapping could also be created as additional model by the reused model transformation. As such solution would only work for one mapping representation and one transformation scenario, we refrained from developing a tool- and definition-specific solution. To offer a method which is widely applicable because it does not require a specific MDPLE tool or specific transformation engine, we contribute language- and definition-independent solutions.

11.2 Benefits and Take-Away

Although the design decisions may limit the development and applicability of the proposed propagation strategies, we identify significant benefits for research, practitioners, and industry in our contributed approaches. While Sec. 11.2.1 summarizes important benefits for research, Sec. 11.2.2 collects benefits for industry and, particularly, describes how to integrate our solutions.

11.2.1 For Research

For research, this thesis summarizes important insights and collects the techniques which *reduce the manual* maintenance efforts and *increase the consistency of variability information* in annotative model-driven product lines.

Holistically Contributed Propagation Strategies On the one hand, the thesis contributes the concepts how to *reuse* existing model transformation technology to generically propagate annotations from a source model to a target representation. For each of the contributed strategies,

the bytecode instruction analysis, the aspect-oriented propagation, and the trace-based propagation we illuminate if and in case how the solution has been approached before, can be approached, and disseminate our novel solution. Thereby we define the properties which have to be met for their applicability. The evaluation provides further insights to what extent the conceptual models can correctly propagate annotations in realistic scenarios where the transformation properties may violate the computational models.

Promising Evaluation Result On the one hand, the outcome of the thesis emphasizes that it is feasible to reuse the existing technique of model transformations and to utilize their capabilities to propagate annotations on top of them. By exploiting transformation traces and aspect-orientation, the thesis contributes propagation strategies which integrate the maintenance of variability in space *seamlessly* in existing technologies.

On the other hand, the evaluation demonstrates that the trace-based propagation and the aspect-oriented propagation deliver promising results and correctly annotate the target representations in most situations which even deviated from the computational model. In sight of a trend towards increasing traceability and refining the information stored in traces (e.g., [Mar+22; Bit+21; EPT18]), the limitations of propagating annotations based on incomplete or no traces diminish. Therefore, the thesis confirms that a reuse-based approach can work generically, particularly, independent of the source metamodel and target representation and does not assume a specific transformation language to be used, as long as trace information is present. Most importantly, in contrast to prior work on this topic, our contributions do not require changing the semantics or even the syntax of existing transformation languages.

11.2.2 For Practitioners and Industry

Although it might be argued that, still, industry and practitioners develop products in a clone-and-own manner and, in any case, based on source code only, this section first discusses the practicality of the chosen variability mechanism. Furthermore, it emphasizes the benefits for practitioners by explaining how to integrate our solutions.

Benefit of Annotative Product Line Engineering First of all, we support product line development processes building on the *annotative* variability mechanism. Compared to the remaining systematic variability mechanisms, the compositional and transformational, annotative variability can be claimed to be the most prevalent one in practice. Using preprocessor directives to employ conditional compilation is not only common to built-in preprocessor-based GPLs but also possible, for instance, in Java. Defining a base version and extending it with modules works to some extent only because, in reality, features typically interact which hinders a strict separation. Therefore, a 1:1 mapping of a feature onto a module is either impossible or causes several dependencies including potentially conflicts among the modules [Ape+13]. Similarly, delta-oriented programming either requires to declare the edit operations performed in one delta module explicitly. Likewise, it provokes an even higher potential for conflicts between applying a multitude of delta modules to one base version. Finally, clone-and-own approaches do not scale for many products and either remain on a two or three clone basis, neglecting the need for systematic product line engineering, or require an integration into a platform [Keh+21]. Thus, consistent variability management will be necessary, too, after this integration. On the whole, for these reasons we recommend employing annotative product line engineering and contribute an automated consistency management for the annotative variability mechanism.

Generic Reuse of Existing (Model-Driven) Technology Besides relying on the annotative variability mechanism, we provide strategies to automate annotating corresponding artifacts based on *reusing existing technology*. Thus, we discuss the requirements for a company to employ one of our contributed solutions. In any case it is necessary to respect the company's proprietary mapping representation (e.g., by the means of an adapter) to maintain it in the company's specific way.

Aspect-Oriented Propagation Assuming that a company employs an aspect-oriented mechanism to create source code from a given multi-variant model. Then, it could weave our generic advice into the execution during the creation. The advice needs to match each source element which can be annotated by the company's annotative mechanism. In the background, a custom realization needs to look up the company's mapping representation to retrieve the annotation mapped onto the matched source element. Based on this information, the advice can enclose the element created by the transformation rule with the target annotation. Even though we have implemented and examined the aspect-oriented propagation in M2T transformations with Xpand, the concepts are generic. Thus, they can be employed whenever the advice allows for 1) a wildcard which matches each annotated source element, 2) a method invocation to retrieve the annotation of the mapping representation, and 3) a way to represent the annotation mapped onto the created target element.

Usage of Trace Information Assuming a company collects traceability information among its artifacts, instead, this information can be used to propagate annotations. While with a classical traceability approach which links several different artifacts an annotation-wise mapping might be preferred to assign an annotation only once to all corresponding elements, with model transformation traces the trace-based propagation can be used almost out of the box. It is necessary that the annotations mapped onto source elements are available and the corresponding target elements. Furthermore, the way the target mapping is represented might be specialized. Based on that information the trace can be iterated. When no trace information is available and the target model was created manually, it might be possible to specify links between the source and target model manually based on a DSL or to perform an automated derivation of a trace.

Bytecode Analysis Finally to employ, the fine-grained annotation assignment based on analyzing rules, it is necessary to use ATL/EMFTVM specifications. If another bytecode representation is available, instead, Sec. 7.2.3 describes general assignment patterns of values and their mapping onto bytecode instructions. This knowledge may be employed to identify assignment patterns in different bytecode representations, such as the Java virtual machine.

Usage in Commercial MDPLE Tools Although we have built prototypes of our approaches in an academic tool, several commercial tools, such as BigLever gears [Big] and pure::variants [Beu13; Gmb22], support (at least) model-based engineering and sell the consistent management of variability information across artifacts. Therefore, our propagation strategies may be beneficial in these tools to automatically keep the variability information consistent.

Summary On the whole, we offer a toolbox of different strategies which are generic and may be accurate enough to be applied in industry. Potentially the highest amount of integration work has to be invested to adapt the mapping representation and to build on trace information.

11.2.3 Take Away

Before providing an outlook, this section summarizes important points to remember.

Independence of Variability Realization Firstly, one obstacle to defining a specific annotation propagation mechanism are missing standards for expressing variability. Although an ongoing community effort works on establishing a common variability model [Sun+21; Ach+20], so far no standardized representation is available. As a further consequence, without a standardized definition of expressing variability, a standardized representation for defining annotations and mapping them onto product line artifacts is available neither. Thus, we cannot offer a solution that propagates standardized variability expressions. In contrast, the concepts of our annotation propagation approaches per se are independent of the concrete variability representation. Consequently, in the case of an available standardized form of mapping variability information onto model elements, our annotation propagation solutions should still work.

Model Matching represents one strategy to extract trace information [Gra14] or, in case of metamodel matching [Kes+14], to synthesize model transformations [LF20]. Although model and metamodel matching forms an important direction to further establish or integrate model-driven development in practice, the accuracy of extracting out-place mappings still resides at a high level mainly for evolving models and metamodels. If a matching technique was used for propagating annotations, it would require improvements in matching corresponding elements which do not carry the same id's, constitute a similar model structure, and are not named similarly. Thus, to apply trace-based propagation we recommend to employ state-of-the-art trace-writing model transformations.

Interplay Annotation and Filter Capability We observed that the specificity of annotations and the granularity of the derivation mechanism (model filter) interplay and can influence the correctness of derived products. This observation confirms that it is essential to carefully map annotations onto (model) elements of the product line and to consider the specificity also when automating the annotation assignment differently (e.g., by using variation control systems).

11.3 Future Research

Last but not the least, this thesis can be considered as one step towards a holistic management of variability across multiple artifacts building a product line which is engineered in a model-driven way. This section sketches ways which subsequent research efforts can build on and how they can extend the contributed propagation strategies and complement the set of automated techniques to construct and maintain annotative model-driven product lines.

Multi-Variant Transformation Language

Based on the computational models derived in this thesis, it would be possible to design a custom multi-variant transformation language. Particularly, if the rules that can be declared conform to the computational model for trace-based propagation and the transformation records fine-grained trace information, it can be guaranteed that the multi-variant target model is annotated correctly.

(N-Way-)Synchronization of Multi-Variant Models Firstly, this thesis provides mechanisms to *propagate* annotations from one model to another representation (i.e., model or text). In practical workflows, particularly in incremental development, firstly, the source models may be subject to changes. Elements might be added or deleted and the same holds for their annotations. Furthermore, not only the source model may change but also the target model or source code which may require to reintegrate these changes (if possible) into a corresponding source model. Therefore, a bidirectional incremental exchange of annotations might be useful to fully support incremental and continuous development of a model-driven product line. Due to the possibility of manually editing annotations and due to a missing uniqueness of annotations, a synchronization of annotations is not a straightforward task. The easiest solution to this problem is to select one transformation direction and in a batch behavior overwrite the entire target side of the transformation direction. As this solution may delete manual modifications performed on the target side, an incremental approach may be necessary which particularly needs to compare each pair of annotations and check whether and in which way they have changed [GNS22]. Furthermore, when considering dependencies between a set of models, a synchronization may not be required only pairwise but n-wise. Consequently, since several matching constellations can occur and the complexity of the matched pairs increases with the complexity of the trace and the freedom of the product line developer, a sound solution may require further investigation.

Unconstrained Variability and Static Analysis Ideally, multi-variant models should not be limited by the single-variant syntax and semantics of state-of-the-art metamodels and modeling languages (i.e., they should support unconstrained variability). Thus, one step in this direction could involve defining the structure of a multi-variant model and the mechanisms how developers should interact with this representation such that the level of complexity does not increase

significantly (e.g., by editing projections [SBW21; BPB17]). Furthermore, program analyses may inspect, for instance, the control flow of the transformation specification or the facility that creates the target representation, to extract assignment patterns thereof. Based on this information, an unconstrained multi-variant target model could be built which may include elements and annotations resulting from executing different execution paths.

Chapter A Appendix

A.1 Classification of Annotation Maintenance in MDPLE Approaches

Table A.1: MDPLE approaches: The first third of the table lists approaches realizing positive variability, the second part annotative variability mechanisms including a projectional mechanism to switch between representations. The last two rows hold multi-view approaches.

Ref- erence	short de- scription	mapping notation	mapping placement	intra prop- agation	inter prop- agation	consis- tency
[GV09]	aspect- oriented MDPLE	element- wise	internal, reuse	manual	-	none
[Whi+09]	MATA	element- wise	proprietary	manual	manual	none
[Ape+09]	composing super- imposed models	module-wise	internal, reuse	manual	none ¹	
[SSA14b; Pie+15]	delta- oriented tools	module-wise	internal, proprietary	manual	manual	none
[Hau+08; Fon+15]	CVL	element- wise	internal, proprietary	manual	-	none
[Zsc+09]	VML*	element- wise	external	manual	manual	none
[CA05; CP06]	template- based	element- wise	internal, reuse	manual	-	none
[BCW11; Juo+19]	Clafer	element- wise	internal, proprietary	manual	-	none
[HKW08]	Feature- Mapper	element- wise	external	manual	manual	none
[BS12]	Famile	element- wise	external	automated	manual	none
[SW16]	SuperMod	element- wise	internal, ex- tending	automated	automated	none
[Reu+20]	projectional editing	element- wise	internal, ex- tending	automated	manual	none
[GS02]	multiple UML views	annotation- wise	external	-	-	by con- struction
[Ana+18]	VaVe	annotation- wise	external	-	-	by con- struction

A.2 ATL/EMFTVM Bytecode Instruction Opcodes

The excerpt in Fig. A.2.1 lists the ATL/EMFTVM bytecode instruction opcodes in the way they are enumerated in the ATL/EMFTVM execution metamodel.

Validation	Value	Name	Literal
	0	PUSH	PUSH
	1	PUSHT	PUSHT
	2	PUSHF	PUSHF
	3	POP	POP
	4	LOAD	LOAD
	5	STORE	STORE
	6	SET	SET
	7	GET	GET
	8	GET_TRANS	GET_TRANS
	9	SET_STATIC	SET_STATIC
	10	GET_STATIC	GET_STATIC
	11	FINDTYPE	FINDTYPE
	12	FINDTYPE_S	FINDTYPE_S
	13	NEW	NEW
	14	NEW_S	NEW_S
	15	DELETE	DELETE
	16	DUP	DUP
	17	DUP_X1	DUP_X1
	18	SWAP	SWAP
	19	SWAP_X1	SWAP_X1
	20	IF	IF
	21	IFN	IFN
	22	GOTO	GOTO
	23	ITERATE	ITERATE
	24	ENDITERATE	ENDITERATE
	25	INVOKE	INVOKE
	26	INVOKE_SUPER	INVOKE_SUPER
	27	INVOKE_STATIC	INVOKE_STATIC
	28	ALLINST	ALLINST
	29	ALLINST_IN	ALLINST_IN
	30	ISNULL	ISNULL
	31	GETENVTYPE	GETENVTYPE
	32	NOT	NOT
	33	AND	AND
	34	OR	OR
	35	XOR	XOR
	36	IMPLIES	IMPLIES
	37	IFTE	IFTE
	38	RETURN	RETURN
	39	GETCB	GETCB
	40	INVOKE_ALL_CBS	INVOKE_ALL_CBS
	41	INVOKE_CB	INVOKE_CB
	42	INVOKE_CB_S	INVOKE_CB_S
	43	MATCH	MATCH
	44	MATCH_S	MATCH_S
	45	ADD	ADD
	46	REMOVE	REMOVE
	47	INSERT	INSERT
	48	GET_SUPER	GET_SUPER
	49	GETENV	GETENV

Figure A.2.1: Complete list of ATL/EMFTVM bytecode instruction opcodes as defined in its execution metamodel (in Eclipse plugin project `org.eclipse.m2m.atl.emfsvm` v4.3.0.v202102071102).

A.3 Flexible Preprocessor

For deriving products from annotated source code we employ a preprocessor. Since we generate Java source code, which does not support a built-in preprocessor, we provided annotations in form of comments with a custom annotation style. Some Java preprocessors exist already but are mostly not compatible with current Eclipse projects (e.g., *Prebop*²) or require a manual adaptation of the build process, (e.g., when using the *Java Comment Preprocessor*³). For that reason, in a student project we developed a flexible (i.e., customizable) Xtext-based preprocessor.

The preprocessor is flexible inasmuch as it allows to configure

² <http://prebop.sourceforge.net/>

³ <https://github.com/raydac/java-comment-preprocessor>

- the file-ending of the source code files which should be preprocessed (e.g., 'java' or 'py'),
- the pair of opening and closing comment literals which define the opening and closing directive of the comment containing the annotation (e.g., '/*' and '*/' or '///' and '///')
- the keywords for opening, closing and possible branching an annotation (e.g., '#ifdef' and '#endif' or '#IFDEF' and '#END').

Based on a given flag-file, which provides feature names and their selection state, the preprocessor parses the given Java project. If the annotation stated in a directive is represented by the feature configuration (satisfiable), the source code stated inside the annotation will be included in the derived variant. If not, it will be excluded from the variant.

To employ this behavior for filtering source code based on a Famile feature configuration, we added a `FlagConverter` which transforms a Famile feature configuration into a flag file. Given that file, the `AspectEvaluator` can derive products from the multi-variant source code platform by executing the preprocessor API.

List of Figures

1.2.1	<i>Commutativity</i> of transformations noted informally.	6
1.4.1	Overview of contributions and their appearance in the thesis.	9
2.1.1	Automation and abstraction climax.	15
2.1.2	Classical MOF modeling hierarchy.	16
2.1.3	Modeling hierarchy for UML class models.	17
2.1.4	UML association in <i>concrete syntax</i> and in abstract syntax.	18
2.1.5	Classical software engineering directions.	20
2.2.1	Schematic overview of a model transformation.	21
2.2.2	Classification of model transformations.	22
2.2.3	Schematic overview of <i>M2T</i> transformations.	25
2.2.4	Parts of the MOF 2.0 Query View Transformation standard.	28
3.0.1	Overview of software product line engineering.	30
3.2.1	Four-clustered process to develop a software product line.	35
3.3.1	Feature model for <i>database contents</i>	37
3.4.1	<i>Compositional</i> implementation of database contents.	41
3.4.2	<i>Delta-oriented</i> implementation of database contents.	42
3.4.3	Annotative implementation of database contents.	43
4.1.1	Engineering directions in MDPLE.	47
4.1.2	Multi-variant (superimposed) UML class model	48
4.1.3	Annotated <i>multi-variant</i> UML class model	52
4.1.4	Mapping notations.	53
4.2.1	Formal concept lattice for mapping maintenance of MDPLE tools.	61
4.2.2	Feature-based classification of annotation maintenance in MDPLE.	62
4.3.1	Feature-based classification of multi-variant model transformations.	64
4.3.2	Formal concept lattice for multi-variant model transformation properties.	68
5.1.1	Fragments of database contents as UML class and Java MoDisco models.	74
5.1.2	Rule which transforms UML packages into Java packages.	75
5.1.3	Rule which transforms UML classes into respective Java elements.	76
5.1.4	Rule applications to multi-variant model in triple-graph representation.	77
5.2.1	Feature-based classification of M2M transformation <i>traces</i>	79
5.2.2	Trace completeness levels.	81
5.2.3	Common trace metamodel.	82
5.3.1	Schema of trace-based annotation propagation.	82
6.1.1	UML class model in graph notation.	88
6.2.1	Multi-variant model in graph notation.	92
6.2.2	Filtered variants in graph notation.	93
6.3.1	Rule application diagram for match in G and direct derivation of H	94
6.3.2	<i>Source-to-target</i> rule of P2P rule.	97
6.3.3	STT rule representing C2C rule.	98
6.3.4	Trace-generating STT rule of C2C rule.	99
6.4.1	Commutativity in graph formalism.	104
6.4.2	Monomorphism preservation by complete derivations.	104

6.4.3	Application of the monomorphism preservation to commutativity diagram. . . .	105
7.0.1	Overview and interplay of strategies to maintain missing trace information. . . .	110
7.1.1	Complete vs. generation-complete trace.	112
7.2.1	Commutativity violation due to fine-grained annotation mapping.	114
7.2.2	Schematic overview of bytecode instruction analysis-based propagation.	116
7.2.3	Mapping of ATL/EMFTVM rule onto bytecode instructions.	118
7.2.4	Classification of assignment patterns in model transformation languages.	120
7.2.5	Overview of bytecode instruction-based analysis.	126
7.2.6	Steps of bytecode instruction-based propagation.	127
7.3.1	Commutativity violation due to <i>incomplete</i> trace information.	136
7.3.2	Schematic overview of employing completion strategies.	140
7.3.3	Overview of steps to complete annotations.	141
7.3.4	Commutativity violation due to incomplete traces in extended example.	143
7.3.5	Computed annotations by <i>container</i> completion strategy (Alg. 4).	145
7.3.6	Computed annotations by <i>combined</i> completion strategy (Alg. 6).	148
7.3.7	Computed annotations for edges (Alg. 7).	149
7.4.1	Schematic overview of propagation DSL-based propagation.	157
7.4.2	Feature-based classification of designing propagation DSLs.	157
7.4.3	Commutativity without model transformations.	159
7.4.4	Model matching framework for generating trace links.	165
7.4.5	Schematic overview of <i>DSL-based</i> propagation by metamodel matching.	167
7.4.6	Schematic overview of <i>trace-based propagation</i> by model matching.	168
7.5.1	Manually repaired broad annotations.	170
8.1.1	Integration of method body in multi-variant vs. single-variant source code. . .	182
8.3.1	Schematic overview of aspect-oriented propagation.	191
8.3.2	Preprocessor directives enclosing text fragments in hierarchy of text blocks. . .	192
8.3.3	Representations of text generated by a M2T transformation.	193
8.3.4	STT-graph representation of Ecore2Java M2T transformation.	194
8.3.5	Commutativity of M2T transformations.	197
8.3.6	Execution tree building tp-STT rule (Def. 8.3.4).	205
9.1.1	Ecore metamodel.	216
9.1.2	Famile mapping model editor.	218
9.1.3	Database Content mappings with missing annotations in F2DMM editor.	219
9.2.1	Overview of architecture of MuVaTra framework.	221
9.2.2	Core of MuVaTra framework.	223
9.2.3	Adapter for Famile Mapping.	224
9.3.1	Trace metamodels.	225
9.3.2	ATL/EMFTVM trace between Ecore and UML database content model.	227
9.3.3	Simplified EMFTVM execution metamodel of its bytecode instructions.	229
9.3.4	AnalyzedRule as data structure.	230
9.3.5	Instance of AnalyzedRule representing ATL rule of Figure 7.2.3.	231
10.2.1	Schematic overview of commutativity evaluation framework.	245
10.2.2	EMFCCompare <i>difference model</i> of database content model variants	248
10.2.3	Graph product line feature model and F2DMM model.	251
10.2.4	Home Automation System product line feature model and F2DMM model. . . .	253
10.2.5	Relational database schema metamodel.	254
10.2.6	Transformed EReferences into columns, foreign keys, events, and tables.	258
10.2.7	QVT-O mapping rule producing incomplete vs. (generation-) complete traces. .	261
10.2.8	Simplified UML metamodel including to represent class models.	262
10.2.9	DBC UML class model resulting from Ecore2UML transformation.	266
10.2.10	Simplified Java MoDisco metamodel.	267
10.3.1	DBC and Graph UML F2DMM models created by Ecore2UML.	274

10.3.2	DBC Java MoDisco F2DMM models created by base UML2Java	276
10.3.3	DBC Java F2DMM model created by advanced UML2Java transformation.	278
10.3.4	Container vs. contained strategy in HAS Java MoDisco.	280
10.3.5	Relational database F2DMM models created by the Ecore2SQL transformation.	284
10.3.6	Excerpt of Graph Java source code created by the Ecore2Java transformation.	288
10.3.7	Excerpt of DBC Java source code created by the Ecore2Java transformation.	290
10.3.8	Commutativity violation in multi-variant interfaces of DBC product line.	291
10.3.9	Excerpts of DBC source code created by MoDisco2Java transformation.	293
10.4.1	Influences of annotations, feature model, and trace completeness.	294
10.4.2	Example of <i>combined</i> completion strategy with positive effect.	298
10.4.3	Statistics of correct vs. repaired annotations.	299
A.2.1	Complete list of ATL/EMFTVM bytecode instruction opcodes.	320

List of Tables

4.1	Terminology in MDSE and SPLE.	49
4.2	Classification of related multi-variant model transformations.	66
5.1	Categorization of an exemplary set of M2M transformation traces.	80
7.1	Overview of basic ATL/EMFTVM bytecode instructions.	117
7.2	Bytecode opcodes of direct assignment patterns of a single value.	121
7.3	Bytecode opcodes of direct assignment patterns of multiple values.	122
10.1	Statistics of subject product lines.	251
10.2	Traced elements in Ecore2SQL transformations.	260
10.3	Results of Ecore2UML transformations.	273
10.4	Results of UML2Java transformation.	275
10.5	Computed effect on correctness of existing annotations.	282
10.6	Results of Ecore2SQL transformations	283
10.7	Measured Error in Ecore2Java transformations.	287
10.8	Counted Lines of Code in Ecore2Java transformations.	289
10.9	Counted Lines of Code in MoDisco2Java transformations.	292
A.1	MDPLE approaches.	319

List of Listings

7.2.1	ATL/EFTVM rule with complex assignments and respective bytecode opcodes. . .	124
7.4.1	Excerpt of ModelSync grammar.	160
7.4.2	ModelSync correspondence definition of UML class and Java metamodel.	161
8.2.1	Example of an Acceleo module composed of three text production rules (template).	185
8.2.2	Example of Xpand text production rules for EClasses	187
8.2.3	Excerpt of MWE2 workflow for generating text from an Ecore model.	188
8.3.1	Xpand rules which generate a Java class declaration for a given EClass	195
8.3.2	Generic advice to enclose every text production with a preprocessor directive. . .	196
9.3.1	Simplified generic advice which embraces an original text generation.	234
9.3.2	Java field declaration annotated with JavaDoc comments.	234
9.3.3	Reused single-variant MWE2 workflow for an Ecore to Java transformation. . . .	235
9.3.4	Advice-injecting workflow for Ecore to Java transformation.	236
9.3.5	Protected text production of generating a method declaration for an EOperation . . .	237
9.3.6	MWE2 workflow for incremental Ecore to Java transformation.	238
10.2.1	BXtend forward transformation rule creating a Table for a given EClass	256
10.2.2	Ecore2UML ATL/EMFTVM rule for transforming subclasses.	265

Abbreviations

AGG	Attributed Graph Grammar
AOP	Aspect-Oriented Programming
ATL	Atlas Transformation Language
CVL	Common Variability Language
DBC	Database Content
DSL	Domain Specific Language
F2DMM	Feature To Domain Model Mapping
FCA	Formal Concept Analysis
FOP	Feature Oriented Programming
GQM	Goal Question Metric
HAS	Home Automation System
M2M	Model-to-Model
M2T	Model-to-Text
MBE	Model-Based Engineering
MDD	Model-Driven Development
MDPLE	Model-Driven Software Product Line Engineering
MDSE	Model-Driven Software Engineering
MVMT	Multi-Variant Model Transformation
oAW	open Architecture Ware
OCL	Object Constraint Language
OMG	Object Management Group
OVM	Orthogonal Variability Model
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QVT	Queries/Views/Transformations
QVT-O	Queries/Views/Transformations Operational Mappings
QVT-R	Queries/Views/Transformations Relational
SE	Software Engineering
SPLE	Software Product Line Engineering
SVMT	Single-Variant Model Transformation
SUMM	Single Underlying Metamodel
TGG	Triple Graph Grammar
TGTS	Triple Graph Transformation System
UML	Unified Modeling Language
UVM	Uniform Version Model
VASG	Variational Abstract Syntax Graph

Chapter B Bibliography

B.1 Third-Party Publications

- [200+] 2015 Obeo 2006 et al. *Comparison (EMF Compare API Specification)*. Modified: March 9, 2020, 09:54:59 GMT+1. URL: <https://www.eclipse.org/emf/compare/documentation/latest/developer/javadoc/org/eclipse/emf/compare/Comparison.html>.
- [201] 2019 Eclipse Foundation, Inc. *EMF Compare / Home*. Modified: July 12, 2021, 10:42:37 GMT+2. URL: <https://www.eclipse.org/emf/compare>.
- [Ach+20] Mathieu Acher, Philippe Collet, David Benavides, and Rick Rabiser. “Third International Workshop on Languages for Modelling Variability (MODEVAR@SPLC 2020)”. In: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. SPLC ’20. Montreal, Quebec, Canada: ACM, 2020. DOI: 10.1145/3382025.3414948.
- [Ach+12] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. “Feature Model Differences”. In: *Advanced Information Systems Engineering - 24th International Conference, CAiSE 2012. Proceedings*. Vol. 7328. Lecture Notes in Computer Science. Springer, 2012, pp. 629–645. DOI: 10.1007/978-3-642-31095-9_41.
- [Add+16] Lorenzo Addazi, Antonio Cicchetti, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. “Semantic-based Model Matching with EMFCompare”. In: *Proceedings of the 10th Workshop on Models and Evolution co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. Vol. 1706. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 40–49. URL: <http://ceur-ws.org/Vol-1706/paper6.pdf>.
- [Aho+13] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Pearson New International Edition: Principles, Techniques, and Tools. (2nd Edition)*. Pearson Education Limited, 2013. ISBN: 1292024348.
- [Aiz+06] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. “Model traceability”. In: *IBM Systems Journal* 45.3 (2006), pp. 515–526. DOI: 10.1147/sj.453.0515.
- [All22] Freddie Allilaire. *ATL Transformations | The Eclipse Foundation*. Visited: May 31, 2022, 06:01:55 GMT+2. 2022. URL: <https://www.eclipse.org/at1/at1Transformations/>.
- [Alt+08] Kerstin Altmanninger, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. “AMOR—towards adaptable model versioning”. In: *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS*. Vol. 8. Citeseer. 2008, pp. 4–50.
- [ASW09] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. “A survey on model versioning approaches”. In: *International Journal of Web Information Systems* 5.3 (2009), pp. 271–304. DOI: 10.1108/17440080910983556.

- [Ana+18] Sofia Ananieva, Heiko Klare, Erik Burger, and Ralf H. Reussner. “Variants and Versions Management for Models with Integrated Consistency Preservation”. In: *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018*. ACM, 2018, pp. 3–10. DOI: 10.1145/3168365.3168377.
- [ABW17] Anthony Anjorin, Thomas Buchmann, and Bernhard Westfechtel. “The Families to Persons Case”. In: *Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017)*, 2017, pp. 27–34. URL: <http://ceur-ws.org/Vol-2026/paper2.pdf>.
- [Anj+20] Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi, and Albert Zündorf. “Benchmarking bidirectional transformations: theory, implementation, application, and assessment”. In: *Software and Systems Modeling* 19.3 (2020), pp. 647–691. DOI: 10.1007/s10270-019-00752-x.
- [Anq+10] Nicolas Anquetil, Uirá Kulesza, Ralf Mitschke, Ana Moreira, Jean-Claude Royer, Andreas Rummeler, and André Sousa. “A model-driven traceability framework for software product lines”. In: *Software and Systems Modeling* 9.4 (2010), pp. 427–451. DOI: 10.1007/s10270-009-0120-9.
- [Ape+13] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013. ISBN: 978-3-642-37520-0. DOI: 10.1007/978-3-642-37521-7.
- [Ape+09] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. “Model Superimposition in Software Product Lines”. In: *Theory and Practice of Model Transformations, ICMT@Tools*. Vol. 5563. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 4–19. DOI: 10.1007/978-3-642-02408-5_2.
- [AK09] Sven Apel and Christian Kästner. “Virtual Separation of Concerns - A Second Chance for Preprocessors”. In: *Journal of Object Technology* 8.6 (2009), pp. 59–78. DOI: 10.5381/jot.2009.8.6.c5.
- [AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. “FEATUREHOUSE: Language-independent, automated software composition”. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 221–231. DOI: 10.1109/ICSE.2009.5070523.
- [AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. “Language-Independent and Automated Software Composition: The FeatureHouse Experience”. In: *IEEE Transactions on Software Engineering* 39.1 (2013), pp. 63–79. DOI: 10.1109/TSE.2011.120.
- [Ape+08] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. “An Algebra for Features and Feature Composition”. In: *Algebraic Methodology and Software Technology, AMAST*. Vol. 5140. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 36–50. DOI: 10.1007/978-3-540-79980-1_4.
- [Are+10a] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. “Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations”. In: *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Proceedings, Part I*. Ed. by Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen. Vol. 6394. Lecture Notes in Computer Science. Springer, 2010, pp. 121–135. URL: https://doi.org/10.1007/978-3-642-16145-2_9.
- [Are+10b] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. “Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations”. In: *Model Driven Engineering Languages and Systems, MODELS*. Vol. 6394. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 121–135. DOI: 10.1007/978-3-642-16145-2_9.

- [Atk98] David L. Atkins. “Version Sensitive Editing: Change History as a Programming Tool”. In: *System Configuration Management, ECOOP’98 SCM-8 Symposium*. Vol. 1439. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 146–157. DOI: 10.1007/BFb0053886.
- [ABB02] Colin Atkinson, Christian Bunse, and Joachim Bayer. *Component-based product line engineering with UML*. Pearson Education, 2002.
- [AK08] Colin Atkinson and Thomas Kühne. “Reducing accidental complexity in domain models”. In: *Software and Systems Modeling 7.3* (2008), pp. 345–359. DOI: 10.1007/s10270-007-0061-0.
- [ASB09] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. “Orthographic Software Modeling: A Practical Approach to View-Based Development”. In: *Evaluation of Novel Approaches to Software Engineering, ENASE*. Vol. 69. Communications in Computer and Information Science. Berlin, Heidelberg: Springer, 2009, pp. 206–219. DOI: 10.1007/978-3-642-14819-4_15.
- [Bak+16] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. “Clafer: unifying class and feature modeling”. In: *Software and Systems Modeling 15.3* (2016), pp. 811–845. DOI: 10.1007/s10270-014-0441-1.
- [BCW11] Kacper Bak, Krzysztof Czarnecki, and Andrzej Wasowski. “Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled”. In: *Software Language Engineering, SLE*. Vol. 6563. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 102–122. DOI: 10.1007/978-3-642-19440-5_7.
- [BBW21] Matthias Bank, Thomas Buchmann, and Bernhard Westfechtel. “Combining a Declarative Language and an Imperative Language for Bidirectional Incremental Model Transformations”. In: *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2021*. SCITEPRESS, 2021, pp. 15–27. DOI: 10.5220/0010188200150027.
- [Bar+10] Bruno Barroca, Levi Lucio, Vasco Amaral, Roberto Félix, and Vasco Sousa. “DSLTrans: A Turing Incomplete Transformation Language”. In: *Software Language Engineering, SLE*. Vol. 6563. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 296–305. DOI: 10.1007/978-3-642-19440-5_19.
- [Bat04] Don S. Batory. “Feature-Oriented Programming and the AHEAD Tool Suite”. In: *Proceedings of the 26th International Conference on Software Engineering, (ICSE)*. USA: IEEE Computer Society, 2004, pp. 702–703. DOI: 10.1109/ICSE.2004.1317496.
- [BSR04] Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. “Scaling Step-Wise Refinement”. In: *IEEE Transactions on Software Engineering 30.6* (2004), pp. 355–371. DOI: 10.1109/TSE.2004.23.
- [Bec+07] Simon M. Becker, Sebastian Herold, Sebastian Lohmann, and Bernhard Westfechtel. “A graph-based algorithm for consistency maintenance in incremental and interactive integration tools”. In: *Software and Systems Modeling 6.3* (2007), pp. 287–315. DOI: 10.1007/s10270-006-0045-5.
- [BPB17] Benjamin Behringer, Jochen Palz, and Thorsten Berger. “PEoPL: projectional editing of product lines”. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE*. IEEE, 2017, pp. 563–574. DOI: 10.1109/ICSE.2017.58.
- [Ben+19] David Benavides, Rick Rabiser, Don S. Batory, and Mathieu Acher. “First international workshop on languages for modelling variability (MODEVAR 2019)”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*. ACM, 2019, 46:1. DOI: 10.1145/3336294.3342364.

- [BBM05] Kathrin Berg, Judith Bishop, and Dirk Muthig. “Tracing Software Product Line Variability: From Problem to Solution Space”. In: White River, South Africa: South African Institute for Computer Scientists and Information Technologists, 2005, pp. 182–191. ISBN: 1595932585.
- [Ber84] Edward H. Bersoff. “Elements of Software Configuration Management”. In: *IEEE Transactions on Software Engineering* 10.1 (1984), pp. 79–87. DOI: 10.1109/TSE.1984.5010202.
- [Bet16] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.
- [Beu13] Danilo Beuche. “pure::variants”. In: *Systems and Software Variability Management: Concepts, Tools and Experiences*. Springer, 2013, pp. 173–182. DOI: 10.1007/978-3-642-36583-6_12.
- [Big] Inc. BigLever Software. *Gears Product Line Engineering Lifecycle Framework™*. Visited: 2022-07-15. URL: <https://biglever.com/wp-content/uploads/2018/11/BigLever-Enterprise-Gears-Data-Sheet.pdf>.
- [Bit+21] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. “Feature Trace Recording”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. Athens, Greece: ACM, 2021, pp. 1007–1020. DOI: 10.1145/3468264.3468531.
- [Boe88] Barry W. Boehm. “A Spiral Model of Software Development and Enhancement”. In: *IEEE Computer* 21.5 (1988), pp. 61–72. DOI: 10.1109/2.59.
- [BJR96] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language for Object-Oriented Development*. Santa Monica, California: Rational Software Cooperation, Sept. 1996.
- [BTG12] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. “A theory of software product line refinement”. In: *Theoretical Computer Science* 455 (2012), pp. 2–30. DOI: 10.1016/j.tcs.2012.01.031.
- [Bos99] Jan Bosch. “Superimposition: a component adaptation technique”. In: *Information and Software Technology* 41.5 (1999), pp. 257–273. DOI: 10.1016/S0950-5849(99)00007-5.
- [BOT07] Goetz Botterweck, Liam O’Brien, and Steffen Thiel. “Model-driven derivation of product architectures”. In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering ASE*. ACM, 2007, pp. 469–472. DOI: 10.1145/1321631.1321711.
- [BP14] Goetz Botterweck and Andreas Pleuss. “Evolution of Software Product Lines”. In: *Evolving Software Systems*. Berlin, Heidelberg: Springer, 2014, pp. 265–295. DOI: 10.1007/978-3-642-45398-4_9.
- [BS13] Julian Bradfield and Perdita Stevens. “Enforcing QVT-R with mu-Calculus and Games”. English. In: *Fundamental Approaches to Software Engineering*. Vol. 7793. Lecture Notes in Computer Science. Springer, 2013, pp. 282–296. DOI: 10.1007/978-3-642-37057-1_21.
- [BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012. DOI: 10.2200/S00441ED1V01Y201208SWE001.
- [BPV10] Mark van den Brand, Zvezdan Protić, and Tom Verhoeff. “Fine-Grained Metamodel-Assisted Model Comparison”. In: *Proceedings of the 1st International Workshop on Model Comparison in Practice, IWMCP*. Malaga, Spain: ACM, 2010, pp. 11–20. DOI: 10.1145/1826147.1826152.

- [Bro+12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. “An Introduction to Model Versioning”. In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM*. Vol. 7320. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 336–398. DOI: 10.1007/978-3-642-30982-3_10.
- [Bru+20] Jean-Michel Bruel, Benoît Combemale, Esther Guerra, Jean-Marc Jézéquel, Jörg Kienzle, Juan de Lara, Gunter Mussbacher, Eugene Syriani, and Hans Vangheluwe. “Comparing and classifying model transformation reuse approaches across meta-models”. In: *Software and Systems Modeling* 19.2 (2020), pp. 441–465. DOI: 10.1007/s10270-019-00762-9.
- [BP08] Cédric Brun and Alfonso Pierantonio. “Model Differences in the Eclipse Modelling Framework”. In: *UPGRADE IX.2* (Apr. 2008), pp. 29–34.
- [Bru+10] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. “MoDisco: a generic and extensible framework for model driven reverse engineering”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering. ASE '10*. Antwerp, Belgium: ACM, 2010, pp. 173–174. DOI: <http://doi.acm.org/10.1145/1858996.1859032>.
- [Buc18] Thomas Buchmann. “BXtend - A Framework for (Bidirectional) Incremental Model Transformations”. In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018*. 2018, pp. 336–345. DOI: 10.5220/0006563503360345.
- [Buc12] Thomas Buchmann. “Valkyrie: A UML-based Model-driven Environment for Model-driven Software Engineering”. In: *ICSOF 2012 - Proceedings of the 7th International Conference on Software Paradigm Trends*. SciTePress, 2012, pp. 147–157. DOI: 10.5220/0004027401470157.
- [BDW08] Thomas Buchmann, Alexander Dotor, and Bernhard Westfechtel. “Triple Graph Grammars or Triple Graph Transformation Systems?” In: *Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers*. Vol. 5421. Lecture Notes in Computer Science. Springer, 2008, pp. 138–150. DOI: 10.1007/978-3-642-01648-6_15.
- [BS16] Thomas Buchmann and Felix Schwägerl. “Breaking the Boundaries of Meta Models and Preventing Information Loss in Model-Driven Software Product Lines”. In: *ENASE 2016 - Proceedings of the 11th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress, 2016, pp. 73–83. DOI: 10.5220/0005789100730083.
- [BS12] Thomas Buchmann and Felix Schwägerl. “FAMILLE: Tool support for evolving model-driven product lines”. In: *Joint Proceedings of co-located Events at 8th European Conference on Modelling Foundations and Applications (ECMFA)*. CEUR WS. Lyngby, Denmark, 2012, pp. 59–62.
- [BS15] Thomas Buchmann and Felix Schwägerl. “On A-posteriori Integration of Ecore Models and Hand-written Java Code”. In: *ICSOF-PT 2015 - Proceedings of the 10th International Conference on Software Paradigm Trends*. SciTePress, 2015, pp. 95–102. DOI: 10.5220/0005552200950102.
- [BW14] Thomas Buchmann and Bernhard Westfechtel. “Mapping feature models onto domain models: ensuring consistency of configured domain models”. In: *Software and Systems Modeling* 13.4 (2014), pp. 1495–1527. DOI: 10.1007/s10270-012-0305-5.
- [Bür+16] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. “Reasoning about product-line evolution using complex feature model differences”. In: *Automated Software Engineering* 23.4 (2016), pp. 687–733. DOI: 10.1007/s10515-015-0185-3.

- [BCG19] Loli Burgueño, Jordi Cabot, and Sébastien Gérard. “An LSTM-Based Neural Network Architecture for Model Transformations”. In: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2019*. IEEE, 2019, pp. 294–299. DOI: 10.1109/MODELS.2019.00013.
- [Cal+19] Théo Le Calvar, Frédéric Jouault, Fabien Chhel, and Mickael Clavreul. “Efficient ATL Incremental Transformations”. In: *Journal of Object Technology* 18.3 (July 2019). The 12th International Conference on Model Transformations, 2:1–17. ISSN: 1660-1769. DOI: 10.5381/jot.2019.18.3.a2.
- [Che+16] Marsha Chechik, Michalis Famelis, Rick Salay, and Daniel Strüber. “Perspectives of Model Transformation Reuse”. In: *Integrated Formal Methods - 12th International Conference, IFM 2016*. Vol. 9681. Lecture Notes in Computer Science. Springer, 2016, pp. 28–44. DOI: 10.1007/978-3-319-33693-0_3.
- [CHS15] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. “Abstract delta modelling”. In: *Mathematical Structures in Computer Sciences* 25.3 (2015), pp. 482–527. DOI: 10.1017/S0960129512000941.
- [CHS08] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. “What’s in a Feature: A Requirements Engineering Perspective”. In: *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008*. Vol. 4961. Lecture Notes in Computer Science. Springer, 2008, pp. 16–30. DOI: 10.1007/978-3-540-78743-3_2.
- [CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. ISBN: 9780201703320.
- [Con91] Software Productivity Consortium. *Synthesis Guidebook*. Tech. rep. Technical report, SPC-91122-MC. Herndon, Virginia, 1991.
- [Cua+22] Jesús Sánchez Cuadrado, Loli Burgueño, Manuel Wimmer, and Antonio Vallecillo. “Efficient Execution of ATL Model Transformations Using Static Analysis and Parallelism”. In: *IEEE Transactions on Software Engineering* 48.4 (2022), pp. 1097–1114. DOI: 10.1109/TSE.2020.3011388.
- [CGL18] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. “AnATLyzer: an advanced IDE for ATL model transformations”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE 2018*. ACM, 2018, pp. 85–88. DOI: 10.1145/3183440.3183479.
- [CGL17] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. “Static Analysis of Model Transformations”. In: *IEEE Transactions on Software Engineering* 43.9 (2017), pp. 868–897. DOI: 10.1109/TSE.2016.2635137.
- [CM09] Jesús Sánchez Cuadrado and Jesús García Molina. “Modularization of model transformations through a phasing mechanism”. In: *Software and Systems Modeling* 8.3 (July 2009), pp. 325–345. ISSN: 1619-1374. DOI: 10.1007/s10270-008-0093-0.
- [CA05] Krzysztof Czarnecki and Michał Antkiewicz. “Mapping Features to Models: A Template Approach Based on Superimposed Variants”. In: *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005*. Vol. 3676. Lecture Notes in Computer Science. Tallinn, Estonia: Springer, 2005, pp. 422–437. DOI: 10.1007/11561347_28.
- [Cza+05] Krzysztof Czarnecki, Michał Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. “Model-driven software product lines”. In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005*. 2005, pp. 126–127. DOI: 10.1145/1094855.1094896.
- [CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming. Methods, Tools, and Applications*. Pearson Education, 2000. ISBN: 0201309777.

- [Cza+09] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. “Bidirectional Transformations: A Cross-Discipline Perspective”. In: *Theory and Practice of Model Transformations - 2nd International Conference, ICMT@TOOLS 2009*. Vol. 5563. Lecture Notes in Computer Science. Springer, 2009, pp. 260–283. DOI: 10.1007/978-3-642-02408-5_19.
- [Cza+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. “Cool features and tough decisions: a comparison of variability modeling approaches”. In: *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*. VaMoS. Leipzig, Germany: ACM, 2012, pp. 173–182. DOI: 10.1145/2110147.2110167.
- [CH03] Krzysztof Czarnecki and Simon Helsen. “Classification of model transformation approaches”. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Vol. 45. 3. USA. 2003, pp. 1–17.
- [CH06] Krzysztof Czarnecki and Simon Helsen. “Feature-based survey of model transformation approaches”. In: *IBM Systems Journal* 45.3 (2006), pp. 621–646. DOI: 10.1147/sj.453.0621.
- [CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. “Formalizing cardinality-based feature models and their specialization”. In: *Software Process: Improvement and Practice* 10.1 (2005), pp. 7–29. DOI: 10.1002/spip.213.
- [CP06] Krzysztof Czarnecki and Krzysztof Pietroszek. “Verifying feature-based model templates against well-formedness OCL constraints”. In: *Generative Programming and Component Engineering, 5th International Conference, GPCE 2006*. ACM, 2006, pp. 211–220. DOI: 10.1145/1173706.1173738.
- [DGR11] Deepak Dhungana, Paul Grünbacher, and Rick Rabiser. “The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study”. In: *Automated Software Engineering* 18.1 (2011), pp. 77–114. DOI: 10.1007/s10515-010-0076-6.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X. URL: <https://www.worldcat.org/oclc/01958445>.
- [Dij72] Edsger W. Dijkstra. “The Humble Programmer”. In: *Communications of the ACM* 15.10 (Oct. 1972), pp. 859–866. ISSN: 0001-0782. DOI: 10.1145/355604.361591.
- [Eff+04] Sven Efftinge, Peter Friese, Arno Hase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Markus Völter, et al. *Xpand documentation*. Tech. rep. Technical report, 2004-2010.(cited on page 64), 2004.
- [ES] Sven Efftinge and Miro Spoenemann. *Xtend - Modernized Java*. Modified: November 30, 2020 at 16:59:59 GMT+1. URL: <https://www.eclipse.org/xtend/index.html>.
- [EV06] Sven Efftinge and Markus Völter. “oAW xText: A framework for textual DSLs”. In: *Workshop on Modeling Symposium at Eclipse Summit*. Vol. 32. 118. 2006.
- [Ehr+08] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 3662569108. DOI: 10.1007/3-540-31188-2.
- [Ehr+15] Hartmut Ehrig, Claudia Ermel, Ulrike Golas, and Frank Hermann. *Graph and Model Transformation - General Framework and Applications*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2015. DOI: 10.1007/978-3-662-47980-3.
- [EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. “Fundamental Theory for Typed Attributed Graph Transformation”. In: *Graph Transformations*. Berlin, Heidelberg: Springer, 2004, pp. 161–177. ISBN: 978-3-540-30203-2.

- [EPT18] Romina Eramo, Alfonso Pierantonio, and Michele Tucci. “Improved traceability for bidirectional model transformations”. 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)*. Vol. 2245. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 306–315. URL: http://ceur-ws.org/Vol-2245/mdetools_paper_1.pdf.
- [EW11] Martin Erwig and Eric Walkingshaw. “The Choice Calculus: A Representation for Software Variation”. In: *ACM Transactions on Software Engineering and Methodology* 21.1 (Dec. 2011). ISSN: 1049-331X. DOI: 10.1145/2063239.2063245.
- [FLW11] Uli Fahrenberg, Axel Legay, and Andrzej Wasowski. “Vision Paper: Make a Difference! (Semantically)”. In: *Model Driven Engineering Languages and Systems. 14th International Conference, MODELS 2011, Wellington, New Zealand, October 2011. Proceedings*. Vol. 6981. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 490–500. DOI: 10.1007/978-3-642-24485-8_36.
- [Fam+15] Michalis Famelis, Levi Lucio, Gehan M. K. Selim, Alessio Di Sandro, Rick Salay, Marsha Chechik, James R. Cordy, Jürgen Dingel, Hans Vangheluwe, and S. Ramesh. “Migrating Automotive Product Lines: A Case Study”. In: *Theory and Practice of Model Transformations. 8th International Conference, ICMT 2015, Held as Part of STAF 2015, L’Aquila, Italy, July 20-21, 2015. Proceedings*. 2015, pp. 82–97. DOI: 10.1007/978-3-319-21155-8_7.
- [FL19] Shichao Fang and Kevin Lano. “Extracting Correspondences from Metamodels Using Metamodel Matching”. In: *STAF 2019 Co-Located Events Joint Proceedings: 1st Junior Researcher Community Event, 2nd International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems, and 1st Research Project Showcase Workshop co-located with Software Technologies: Applications and Foundations (STAF 2019)*. Vol. 2405. CEUR Workshop Proceedings. CEUR-WS.org, 2019, pp. 3–8. URL: http://ceur-ws.org/Vol-2405/02_paper.pdf.
- [Fei+13] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachsel, Maria Papendieck, Thomas Leich, and Gunter Saake. “Do background colors improve program comprehension in the #ifdef hell?”. In: *Empirical Software Engineering* 18.4 (2013), pp. 699–745. DOI: 10.1007/s10664-012-9208-x.
- [Fel91] Stuart I. Feldman. “Software Configuration Management: Past Uses and Future Challenges”. In: *ESEC ’91. 3rd European Software Engineering Conference, ESEC ’91, Milan, Italy, October 21-24, 1991, Proceedings*. Vol. 550. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1991, pp. 1–6. DOI: 10.1007/3540547428_39.
- [Fer+20] Fischer Ferreira, Markos Vigiato, Maurício Souza, and Eduardo Figueiredo. “Testing Configurable Software Systems: The Failure Observation Challenge”. In: *PLC ’20: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A. SPLC ’20*. Montreal, Quebec, Canada: ACM, 2020. DOI: 10.1145/3382025.3414725.
- [FF00] Robert E. Filman and Daniel P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*. Tech. rep. 2000.
- [Fin+92] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, L. Finkelstein, and Michael Goedicke. “Viewpoints: A Framework for Integrating Multiple Perspectives in System Development”. In: *International Journal of Software Engineering and Knowledge Engineering* 2.1 (1992), pp. 31–57. DOI: 10.1142/S0218194092000038.

- [Fon+15] Jaime Font, Manuel Ballarín, Øystein Haugen, and Carlos Cetina. “Automating the variability formalization of a model family by means of common variability language”. In: *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Volume 1*. New York, NY, USA: ACM, 2015, pp. 411–418. DOI: 10.1145/2791060.2793678.
- [Fow22] Martin Fowler. Visited: 2022-07-18. 2022. URL: <https://martinfowler.com/>.
- [Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.
- [Gam+97] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1997. ISBN: 978-0201633610.
- [GW99] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis. Mathematical Foundations*. Springer, 1999. ISBN: 978-3-540-62771-5. DOI: 10.1007/978-3-642-59830-2.
- [Ger+02] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. “Transformation: The Missing Link of MDA”. In: *Graph Transformation, First International Conference, ICGT 2002, Proceedings*. Vol. 2505. Lecture Notes in Computer Science. Barcelona, Spain: Springer, 2002, pp. 90–105. DOI: 10.1007/3-540-45832-8_9.
- [Gmb22] pure-systems GmbH. *pure-systems: pure::variants basic capabilities | pure-systems*. Modified: July 29, 2022. 2022. URL: <https://www.pure-systems.com/purevariants/purevariants-basic-capabilities>.
- [Gom06] Hassan Gomaa. “Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures”. In: *Proceedings. 10th International Software Product Line Conference, SPLC. 2006*, p. 218. DOI: 10.1109/SPLINE.2006.1691600.
- [Gom05] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Boston, MA, USA: Addison-Wesley, 2005.
- [GS02] Hassan Gomaa and Michael E. Shin. “Multiple-View Meta-Modeling of Software Product Lines”. In: *ICECCS 2002: 8th IEEE International Conference on Engineering of Complex Computer Systems, 2002. Proceedings*. IEEE Computer Society, 2002, pp. 238–246. DOI: 10.1109/ICECCS.2002.1181517.
- [Gra14] Birgit Grammel. “Automatic Generation of Trace Links in Model-driven Software Development”. PhD thesis. Dresden University of Technology, 2014. URL: <https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-155839>.
- [GKV12] Birgit Grammel, Stefan Kastenzholz, and Konrad Voigt. “Model Matching for Trace Link Generation in Model-Driven Software Development”. In: *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012*. Vol. 7590. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 609–625. DOI: 10.1007/978-3-642-33666-9_39.
- [Gre22] Sandra Greiner. *Reuse of Model Transformations for Propagating Variability Annotations in Annotative Software Product Lines - Evaluation Data*. Version v1.0.0. Zenodo, Dec. 2022. DOI: 10.5281/zenodo.7367986. URL: <https://doi.org/10.5281/zenodo.7367986>.
- [GV09] Iris Groher and Markus Voelter. “Aspect-Oriented Model-Driven Software Product Line Engineering”. In: *Transactions on Aspect-Oriented Software Development VI: Special Issue on Aspects and Model-Driven Engineering*. Berlin, Heidelberg: Springer, 2009, pp. 111–152. DOI: 10.1007/978-3-642-03764-1_4.
- [GV07] Iris Groher and Markus Voelter. “XWeave: Models and Aspects in Concert”. In: *Proceedings of the 10th International Workshop on Aspect-Oriented Modeling*. AOM ’07. Vancouver, Canada: ACM, 2007, pp. 35–40. DOI: 10.1145/1229375.1229381.
- [Hal17] Paul R. Halmos. *Naive set theory*. Courier Dover Publications, 2017.

- [Hau+08] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. “Adding Standardized Variability to Domain Specific Languages”. In: *2008 12th International Software Product Line Conference*. Limerick, Ireland, Sept. 2008, pp. 139–148. DOI: 10.1109/SPLC.2008.25.
- [Hei+09] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. “Closing the Gap between Modelling and Java”. In: *Software Language Engineering, Second International Conference, SLE, Revised Selected Papers*. Vol. 5969. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 374–383. DOI: 10.1007/978-3-642-12107-4_25.
- [HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. “FeatureMapper: Mapping Features to Models”. In: *Companion of the 30th International Conference on Software Engineering*. ICSE Companion ’08. Leipzig, Germany: ACM, 2008, pp. 943–944. DOI: 10.1145/1370175.1370199.
- [HJH06] Scott A. Hendrickson, Bryan Jett, and André van der Hoek. “Layered Class Diagrams: Supporting the Design Process”. In: *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*. Vol. 4199. Lecture Notes in Computer Science. Springer, 2006, pp. 722–736. DOI: 10.1007/11880240_50.
- [Her10] Markus Herrmannsdoerfer. “COPE - A Workbench for the Coupled Evolution of Metamodels and Models”. In: *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*. Vol. 6563. Lecture Notes in Computer Science. Springer, 2010, pp. 286–295. DOI: 10.1007/978-3-642-19440-5_18.
- [Hoe20] Fabian Hoellerich. “Graph-basiertes Matching für Instanzen unterschiedlicher Metamodelle”. Bachelor Thesis. University of Bayreuth, 2020.
- [Hof+10] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “Leviathan: SPL Support on Filesystem Level”. In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*. Vol. 6287. Lecture Notes in Computer Science. Springer, 2010, p. 491. DOI: 10.1007/978-3-642-15579-6_43.
- [HS77] J.W. Hunt and T.G. Szymanski. “A Fast Algorithm for Computing Longest Common Subsequences”. In: *Communications of the ACM* 20.5 (May 1977), pp. 350–353.
- [ikv18] ikv++ technologies. *medini QVT*. <http://projects.ikv.de/qvt>. ikv++ technologies. 2018.
- [INR05] INRIA. *ATL Transformation Example. Class to Relational*. November 23, 2022 at 22:21:59 GMT+1. 2005. URL: [https://www.eclipse.org/at1/at1Transformations/Class2Relational/ExampleClass2Relational\[v00.01\].pdf](https://www.eclipse.org/at1/at1Transformations/Class2Relational/ExampleClass2Relational[v00.01].pdf).
- [Ji+15] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. “Maintaining feature traceability with embedded annotations”. In: *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*. ACM, 2015, pp. 61–70. DOI: 10.1145/2791060.2791107.
- [Jou+08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. “ATL: A model transformation tool”. In: *Science of Computer Programming* 72.1-2 (2008), pp. 31–39. DOI: 10.1016/j.scico.2007.08.002.
- [Juo+19] Paulius Juodisius, Atrisha Sarkar, Raghava Rao Mukkamala, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. “Clafer: Lightweight Modeling of Structure, Behaviour, and Variability”. In: *The Art, Science, and Engineering of Programming* 3.1 (2019), p. 2. DOI: 10.22152/programming-journal.org/2019/3/2.
- [Kah+19] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Dániel Varró. “Survey and classification of model transformation tools”. In: *Software and Systems Modeling* 18.4 (2019), pp. 2361–2397. DOI: 10.1007/s10270-018-0665-6.

- [Kan+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. Carnegie-Mellon University, Software Engineering Institute, Nov. 1990.
- [Kar+03] Gabor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. “On the Use of Graph Transformation in the Formal Specification of Model Interpreters”. In: *Journal of Universal Computer Science* 9.11 (2003), pp. 1296–1321. URL: http://www.jucs.org/jucs_9_11/on_the_use_of.
- [Käs07] Christian Kästner. “CIDE: Decomposing Legacy Applications into Features”. In: *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007, pp. 149–150.
- [Käs12] Christian Kästner. “Virtual Separation of Concerns: Toward Preprocessors 2.0”. In: *it Inf. Technol.* 54.1 (2012), pp. 42–46. DOI: 10.1524/itit.2012.0662.
- [Käs10] Christian Kästner. “Virtual separation of concerns: toward preprocessors 2.0”. PhD thesis. Otto von Guericke University Magdeburg, 2010. ISBN: 978-3-8325-2527-9. URL: <http://edoc.bibliothek.uni-halle.de/servlets/DocumentServlet?id=8044>.
- [KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. “Granularity in software product lines”. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, 2008, pp. 311–320. DOI: 10.1145/1368088.1368131.
- [KDO14] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. “Variability Mining: Consistent Semi-automatic Detection of Product-Line Features”. In: *IEEE Transactions on Software Engineering* 40.1 (Jan. 2014), pp. 67–82. ISSN: 0098-5589. DOI: 10.1109/TSE.2013.45.
- [Kav+11] Amogh Kavimandan, Aniruddha S. Gokhale, Gabor Karsai, and Jeff Gray. “Managing the quality of software product line architectures through reusable model transformations”. In: *Proceedings of the 7th International ACM SIGSOFT Conference on the Quality of Software Architectures, QoSA 2011 and 2nd ACM SIGSOFT International Symposium on Architecting Critical Systems, ISARCS*. ACM, 2011, pp. 13–22. URL: <https://doi.org/10.1145/2000259.2000264>.
- [KKT13] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. “Consistency-preserving edit scripts in model versioning”. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Silicon Valley, CA, USA: IEEE, 2013, pp. 191–201. DOI: 10.1109/ASE.2013.6693079.
- [Keh+21] Timo Kehrer, Thomas Thüm, Alexander Schultheiß, and Paul Maximilian Bittner. “Bridging the Gap Between Clone-and-Own and Software Product Lines”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2021, pp. 21–25. DOI: 10.1109/ICSE-NIER52604.2021.00013.
- [KWN05] Udo Kelter, Jürgen Wehren, and Jörg Niere. “A Generic Difference Algorithm for UML Models”. In: *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen*. Vol. P-64. LNI. GI, 2005, pp. 105–116. URL: <https://dl.gi.de/20.500.12116/28304>.
- [Ker03] Harold Kerzner. *Project Management. A Systems Approach to Planning, Scheduling and Controlling*. 8th ed. Berea Ohio: John Wiley & Sons Inc., 2003. ISBN: 0201309777.
- [Kes+14] Marouane Kessentini, Ali Ouni, Philip Langer, Manuel Wimmer, and Slim Bechikh. “Search-based metamodel matching with structural and syntactic measures”. In: *Journal of Systems and Software* 97 (2014), pp. 1–14. DOI: 10.1016/j.jss.2014.06.040.

- [Kic+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. “Aspect-oriented programming”. In: *ECOOP’97—Object-oriented programming* (1997), pp. 220–242.
- [Kir+21] Jörg Christian Kirchhof, Michael Nieke, Ina Schaefer, David Schmalzing, and Michael Schulze. “Variant and Product Line Co-Evolution”. In: *Model-Based Engineering of Collaborative Embedded Systems: Extensions of the SPES Methodology*. Cham: Springer International Publishing, 2021, pp. 333–351. DOI: 10.1007/978-3-030-62136-0_18.
- [KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671.
- [Kla21] Heiko Klare. “Building Transformation Networks for Consistent Evolution of Interrelated Models”. PhD thesis. Karlsruhe Institute of Technology, Germany, 2021. URL: <https://nbn-resolving.org/urn:nbn:de:101:1-2021080405072192372510>.
- [Kla07] Benjamin Klatt. “Xpand: A closer look at the model2text transformation language”. In: *Language* 10.16 (2007), p. 2008.
- [KH10] Maximilian Koegel and Jonas Helming. “EMFStore: a model repository for EMF models”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010, pp. 307–308. DOI: 10.1145/1810295.1810364.
- [Kol+09] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. “Different models for model matching: An analysis of approaches to support model differencing”. In: *2009 ICSE Workshop on Comparison and Versioning of Software Models*. 2009, pp. 1–6. DOI: 10.1109/CVSM.2009.5071714.
- [KS04] Christian Koppen and Maximilian Störzer. “PCDiff: Attacking the fragile pointcut problem”. In: *European interactive workshop on aspects in software (EIWAS)*. Vol. 7. 2004.
- [KBL13] Max E. Kramer, Erik Burger, and Michael Langhammer. “View-Centric Engineering with Synchronized Heterogeneous Models”. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO ’13. Montpellier, France: ACM, 2013. DOI: 10.1145/2489861.2489864.
- [Kra] Christian Krause. *Henshin - Examples* / *The Eclipse Foundation*. Visited: June 15, 2022, 06:01:55 GMT+2. URL: <https://www.eclipse.org/henshin/examples.php?example=ecore2rdb>.
- [Kru92] Charles W. Krueger. “Software Reuse”. In: *ACM Computing Surveys* 24.2 (June 1992), pp. 131–183. ISSN: 0360-0300. DOI: 10.1145/130844.130856.
- [Küh06] Thomas Kühne. “Matters of (Meta-)Modeling”. In: *Software and Systems Modeling* 5.4 (2006), pp. 369–385. DOI: 10.1007/s10270-006-0017-9.
- [Kus+15] Angelika Kusel, Johannes Schönböck, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. “Reuse in model-to-model transformation languages: are we there yet?” In: *Software and Systems Modeling* 14.2 (2015), pp. 537–572. DOI: 10.1007/s10270-013-0343-7.
- [LF20] Kevin Lano and Shichao Fang. “Automated Synthesis of ATL Transformations from Metamodel Correspondences”. In: *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - MODELSWARD*. INSTICC. SciTePress, 2020, pp. 263–270. DOI: 10.5220/0008873702630270.
- [Lar+18] Juan de Lara, Esther Guerra, Marsha Chechik, and Rick Salay. “Model Transformation Product Lines”. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*. Copenhagen, Denmark, 2018, pp. 67–77. DOI: 10.1145/3239372.3239377.

- [LB03] C. Larman and V.R. Basili. “Iterative and incremental developments. a brief history”. In: *Computer* 36.6 (2003), pp. 47–56. DOI: 10.1109/MC.2003.1204375.
- [LS05] Michael Lawley and Jim Steel. “Practical Declarative Model Transformation with Tefkat”. In: *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*. Vol. 3844. Lecture Notes in Computer Science. Springer, 2005, pp. 139–150. DOI: 10.1007/11663430_15.
- [LAS14] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. “Developing eMoflon with eMoflon”. In: *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*. 2014, pp. 138–145. DOI: 10.1007/978-3-319-08789-4_10.
- [LAS15] Erhan Leblebici, Anthony Anjorin, and Andy Schürr. “Tool Support for Multi-amalgamated Triple Graph Grammars”. In: *Graph Transformation - 8th International Conference, ICGT 2015, Held as Part of STAF 2015, L’Aquila, Italy, July 21-23, 2015. Proceedings*. Vol. 9151. Lecture Notes in Computer Science. Springer, 2015, pp. 257–265. DOI: 10.1007/978-3-319-21145-9_16.
- [Lie+10] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. “An analysis of the variability in forty preprocessor-based software product lines”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010, pp. 105–114. DOI: 10.1145/1806799.1806819.
- [LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. “DSMDiff: a differentiation tool for domain-specific models”. In: *European Journal of Information Systems* 16.4 (2007), pp. 349–361. DOI: 10.1057/palgrave.ejis.3000685.
- [Lin02] Frank van der Linden. “Software Product Families in Europe: The Esaps & Café Projects”. In: *IEEE Software* 19.4 (2002), pp. 41–49. DOI: 10.1109/MS.2002.1020286.
- [Lin+15] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification - Java SE 8 Edition*. 2015-02-13. <https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf>. Oracle America, Inc. 2015.
- [Lin+21] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Virtual Machine Specification - Java SE 17 Edition*. 2021-08-09. <https://docs.oracle.com/javase/specs/jvms/se17/jvms17.pdf>. Oracle America, Inc. 2021.
- [LB01] Roberto E. Lopez-Herrejon and Don S. Batory. “A Standard Problem for Evaluating Product-Line Methodologies”. In: *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*. GCSE ’01. London, UK: Springer, 2001, pp. 10–24. ISBN: 3-540-42546-2.
- [MS99] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. 1st ed. Cambridge, Ma.; London, England: MIT Press, 1999. ISBN: 0-262-13360-1.
- [MRR11a] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. “ADDiff: semantic differencing for activity diagrams”. In: *SIGSOFT/FSE’11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC’11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 2011, pp. 179–189. DOI: 10.1145/2025113.2025140.
- [MRR11b] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. “CDDiff: Semantic Differencing for Class Diagrams”. In: *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*. Vol. 6813. Lecture Notes in Computer Science. Springer, 2011, pp. 230–254. DOI: 10.1007/978-3-642-22655-7_12.

- [Mar+22] Salome Maro, Jan-Philipp Steghöfer, Paolo Bozzelli, and Henry Muccini. “TracIMo: a traceability introduction methodology and its evaluation in an Agile development team”. In: *Requirements Engineering* 27.1 (2022), pp. 53–81. DOI: 10.1007/s00766-021-00361-5.
- [Mar+21] Johan Martinson, Herman Jansson, Mukelabai Mukelabai, Thorsten Berger, Alexandre Bergel, and Truong Ho-Quang. “HANs: IDE-based editing support for embedded feature annotations”. In: *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kindom, September 6-11, 2021, Volume B*. ACM, 2021, pp. 28–31. DOI: 10.1145/3461002.3473072.
- [MGR02] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. “Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching”. In: *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*. IEEE Computer Society, 2002, pp. 117–128. DOI: 10.1109/ICDE.2002.994702.
- [MG06] Tom Mens and Pieter Van Gorp. “A Taxonomy of Model Transformation”. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), pp. 125–142. DOI: 10.1016/j.entcs.2005.10.021.
- [ML97] Marc H. Meyer and Alvin P. Lehnerd. *The Power of Product Platforms*. Simon and Schuster, 1997.
- [NNZ00] Ulrich Nickel, Jörg Niere, and Albert Zündorf. “The FUJABA environment”. In: *Proceedings of the 22nd International Conference on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*. 2000, pp. 742–745. DOI: 10.1145/337180.337620.
- [NES17] Michael Nieke, Gil Engel, and Christoph Seidl. “DarwinSPL: an integrated tool suite for modeling evolving context-aware software product lines”. In: *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2017, Eindhoven, Netherlands, February 1-3, 2017*. ACM, 2017, pp. 92–99. DOI: 10.1145/3023956.3023962.
- [Obj17a] Object Management Group (OMG). *Action Language for Foundational UML (Alf). Concrete Syntax for a UML Action Language. Version 1.1*. formal/2017-07-04. <https://www.omg.org/spec/ALF/1.1/PDF>. Needham, MA, July 2017.
- [Obj19a] Object Management Group (OMG). *Business Process Model and Notation. Version 2.0.2*. formal/2013-12-09. Needham, MA, 2019. URL: <https://www.omg.org/spec/BPMN/2.0.2/PDF>.
- [Obj16] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Version 1.3*. formal/2016-06-03. <https://www.omg.org/spec/QVT/1.3/PDF>. Needham, MA, June 2016.
- [Obj19b] Object Management Group (OMG). *Meta Object Facility (MOF). Version 2.5.1*. formal/2019-10-01. <https://www.omg.org/spec/MOF/2.5.1>. Needham, MA, 2019.
- [Obj00] Object Management Group (OMG). *Model Driven Architecture. Version 3.2*. 00-11-05. <https://www.omg.org/cgi-bin/doc?omg/00-11-05>. Boston, MA, 2000.
- [Obj08] Object Management Group (OMG). *MOF Model to Text Transformation Language, v1.0*. 08-01-16. <https://www.omg.org/spec/MOFM2T/1.0/PDF>. Needham, MA, Jan. 2008.
- [Obj14] Object Management Group (OMG). *Object Constraint Language. Version 2.4*. formal/2014-02-03. <https://www.omg.org/spec/OCL/2.4/PDF>. Needham, MA, Feb. 2014.
- [Obj18] Object Management Group (OMG). *Semantics of a Foundational Subset for Executable UML Models (fUML). Version 1.4*. formal/2018-12-01. <https://www.omg.org/spec/FUML/1.4/PDF>. Needham, MA, Dec. 2018.
- [Obj17b] Object Management Group (OMG). *Unified Modeling Language. Version 2.5.1*. formal/2017-12-05. <https://www.omg.org/spec/UML/2.5.1/PDF>. Needham, MA, Dec. 2017.

- [OH07] Jon Oldevik and Øystein Haugen. “Higher-Order Transformations for Product Lines”. In: *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*. IEEE Computer Society, 2007, pp. 243–254. DOI: 10.1109/SPLINE.2007.11.
- [OO07] Gøran K. Olsen and Jon Oldevik. “Scenarios of Traceability in Model to Text Transformations”. In: *Model Driven Architecture - Foundations and Applications, 3rd European Conference, ECMDA-FA 2007, Haifa, Israel, June 11-15, 2007, Proceedings*. Vol. 4530. Lecture Notes in Computer Science. Springer, 2007, pp. 144–156. DOI: 10.1007/978-3-540-72901-3_11.
- [PMR16] Richard F. Paige, Nicholas Drivalos Matragkas, and Louis M. Rose. “Evolving models in Model-Driven Engineering: State-of-the-art and future challenges”. In: *Journal of Systems and Software* 111 (2016), pp. 272–280. DOI: 10.1016/j.jss.2015.08.047.
- [Par79] David Lorge Parnas. “Designing Software for Ease of Extension and Contraction”. In: *IEEE Transactions on Software Engineering* 5.2 (1979), pp. 128–138. DOI: 10.1109/TSE.1979.234169.
- [Par72] David Lorge Parnas. “On the Criteria To Be Used in Decomposing Systems into Modules”. In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058. DOI: 10.1145/361598.361623.
- [Pet+19] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. “Product sampling for product lines: the scalability challenge”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*. ACM, 2019, 14:1–14:6. DOI: 10.1145/3336294.3336322.
- [Pie+15] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. “SiPL - A Delta-Based Modeling Framework for Software Product Line Engineering”. In: *Proc. 30th ASE*. Lincoln, NE, USA, Nov. 2015, pp. 852–857. DOI: 10.1109/ASE.2015.106.
- [Pie+19] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. “Formal foundations for analyzing and refactoring delta-oriented model-based software product lines”. In: *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*. ACM, 2019, 30:1–30:11. DOI: 10.1145/3336294.3336299.
- [PBL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Germany: Springer, 2005.
- [Pra71] Terrence W. Pratt. “Pair Grammars, Graph Languages and String-to-Graph Translations”. In: *Journal of Computer and System Sciences* 5.6 (1971), pp. 560–595. DOI: 10.1016/S0022-0000(71)80016-8.
- [RJ01] Balasubramaniam Ramesh and Matthias Jarke. “Toward Reference Models of Requirements Traceability”. In: *IEEE Transactions on Software Engineering* 27.1 (2001), pp. 58–93. DOI: 10.1109/32.895989.
- [Rat+22] Michael Ratzenböck, Paul Grünbacher, Wesley K. G. Assunção, Alexander Egyed, and Lukas Linsbauer. “Refactoring Product Lines by Replaying Version Histories”. In: *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS ’22)*. New York, NY, USA: ACM, 2022. DOI: 10.1145/3510466.3510484.
- [RLK19] Dennis Reuling, Malte Lochau, and Udo Kelter. “From Imprecise N-Way Model Matching to Precise N-Way Model Merging”. In: *Journal of Object Technology* 18.2 (2019), 8:1–20. DOI: 10.5381/jot.2019.18.2.a8.

- [Reu+20] Dennis Reuling, Christopher Pietsch, Udo Kelter, and Timo Kehrer. “Towards Projectional Editing for Model-Based SPLs”. In: *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. VAMOS ’20. Magdeburg, Germany: ACM, 2020. DOI: 10.1145/3377024.3377030.
- [Rhe+15] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. “Presence-Condition Simplification in Highly Configurable Systems”. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 2015, pp. 178–188. DOI: 10.1109/ICSE.2015.39.
- [Rhe+18] Alexander Von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. “Variability-Aware Static Analysis at Scale: An Empirical Study”. In: *ACM Transactions on Software Engineering and Methodology* 27.4 (Nov. 2018). DOI: 10.1145/3280986.
- [RV08] José Eduardo Rivera and Antonio Vallecillo. “Representing and Operating with Model Differences”. In: *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*. Vol. 11. Lecture Notes in Business Information Processing. Springer, 2008, pp. 141–160. DOI: 10.1007/978-3-540-69824-1_9.
- [Roy87] W. W. Royce. “Managing the Development of Large Software Systems: Concepts and Techniques”. In: *Proceedings, 9th International Conference on Software Engineering, Monterey, California, USA, March 30 - April 2, 1987*. ACM Press, 1987, pp. 328–339. URL: <http://dl.acm.org/citation.cfm?id=41801>.
- [RET11] Olga Runge, Claudia Ermel, and Gabriele Taentzer. “AGG 2.0 - New Features for Specifying and Analyzing Algebraic Graph Transformations”. In: *Applications of Graph Transformations with Industrial Relevance - 4th International Symposium, AGTIVE 2011, Revised Selected and Invited Papers*. Vol. 7233. Lecture Notes in Computer Science. Budapest, Hungary: Springer, 2011, pp. 81–88. DOI: 10.1007/978-3-642-34176-2_8.
- [Sal+14] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. “Lifting model transformations to product lines”. In: *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*. 2014, pp. 117–128. DOI: 10.1145/2568225.2568267.
- [Sch+10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. “Delta-Oriented Programming of Software Product Lines”. In: *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010. Proceedings*. Vol. 6287. Lecture Notes in Computer Science. Jeju Island, South Korea: Springer, 2010, pp. 77–91. DOI: 10.1007/978-3-642-15579-6_6.
- [SBD11] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. “Compositional type-checking for delta-oriented programming”. In: *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD 2011*. Porto de Galinhas, Brazil: ACM, 2011, pp. 43–56. DOI: 10.1145/1960275.1960283.
- [SJ04] Klaus Schmid and Isabel John. “A customizable approach to full lifecycle variability management”. In: *Science of Computer Programming* 53.3 (2004), pp. 259–284. DOI: 10.1016/j.scico.2003.04.002.
- [SRG11] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. “A comparison of decision modeling approaches in product lines”. In: *Fifth International Workshop on Variability Modelling of Software-Intensive Systems, Namur, Belgium, January 27-29, 2011. Proceedings*. ACM International Conference Proceedings Series. ACM, 2011, pp. 119–126. DOI: 10.1145/1944892.1944907.
- [Sch06] Douglas C. Schmidt. “Guest Editor’s Introduction: Model-Driven Engineering”. In: *IEEE Computer* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58.

- [SG08] Maik Schmidt and Tilman Gloetzner. “Constructing difference tools for models using the SiDiff framework”. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*. ACM, 2008, pp. 947–948. DOI: 10.1145/1370175.1370201.
- [SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. “Feature Diagrams: A Survey and a Formal Semantics”. In: *14th IEEE International Conference on Requirements Engineering (RE 2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA*. IEEE Computer Society, 2006, pp. 136–145. DOI: 10.1109/RE.2006.23.
- [SBW21] Johannes Schröpfer, Thomas Buchmann, and Bernhard Westfechtel. “A Framework for Projectional Multi-variant Model Editors”. In: *Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development*. MODELSWARD. SCITEPRESS, 2021, pp. 294–305. DOI: 10.5220/0010310102940305.
- [Sch+16] Sandro Schulze, Michael Schulze, Uwe Ryssel, and Christoph Seidl. “Aligning Co-evolving Artifacts Between Software Product Lines and Products”. In: *Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '16. Salvador, Brazil: ACM, 2016, pp. 9–16. DOI: 10.1145/2866614.2866616.
- [Sch94] Andy Schürr. “Specification of Graph Translators with Triple Graph Grammars”. In: *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Herrsching, Germany, June 16-18, 1994, Proceedings*. 1994, pp. 151–163. DOI: 10.1007/3-540-59071-4_45.
- [Sch18] Felix Schwägerl. “Version Control and Product Lines in Model-Driven Software Engineering”. PhD thesis. University of Bayreuth, Germany, 2018. URL: <https://epub.uni-bayreuth.de/3554/>.
- [SBW16] Felix Schwägerl, Thomas Buchmann, and Bernhard Westfechtel. “Multi-variant Model Transformations - A Problem Statement”. In: *ENASE 2016 - Proceedings of the 11th International Conference on Evaluation of Novel Approaches to Software Engineering, Rome, Italy 27-28 April, 2016*. 2016, pp. 203–209. DOI: 10.5220/0005878702030209.
- [SW19] Felix Schwägerl and Bernhard Westfechtel. “Integrated revision and variation control for evolving model-driven software product lines”. In: *Software and Systems Modeling* 18.6 (2019), pp. 3373–3420. DOI: 10.1007/s10270-019-00722-3.
- [SW16] Felix Schwägerl and Bernhard Westfechtel. “SuperMod: tool support for collaborative filtered model-driven software product line engineering”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, 2016, pp. 822–827. DOI: 10.1145/2970276.2970288.
- [SHG12] Andreas Seibel, Regina Hebig, and Holger Giese. “Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance”. In: *Software and Systems Traceability*. Springer, 2012, pp. 215–240. DOI: 10.1007/978-1-4471-2239-5_10.
- [Sei17] Christoph Seidl. “Integrated Management of Variability in Space and Time in Software Families”. PhD thesis. Dresden University of Technology, Germany, 2017. URL: <https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-218036>.
- [SSA14a] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. “Capturing variability in space and time with hyper feature models”. In: *Proceedings of the 8th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'14), Sophia Antipolis*. ACM, 2014, 6:1–6:8.
- [SSA14b] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. “DeltaEcore - A Model-Based Delta Language Generation Framework”. In: *Modellierung 2014*. Vol. P-225. LNI. Wien, Österreich: GI, 2014, pp. 81–96. URL: <https://dl.gi.de/20.500.12116/17067>.

- [SSA14c] Christoph Seidl, Ina Schaefer, and Uwe Abmann. “Integrated management of variability in space and time in software families”. In: *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*. ACM, 2014, pp. 22–31. DOI: 10.1145/2648511.2648514.
- [SK03] Shane Sendall and Wojtek Kozaczynski. “Model Transformation: The Heart and Soul of Model-Driven Software Development”. In: *IEEE Software* 20.5 (2003), pp. 42–45. DOI: 10.1109/MS.2003.1231150.
- [Sij10] Marten Sijtema. “Introducing variability rules in ATL for managing variability in MDE-based product lines”. In: *Proc. of MtATL* 10 (2010), pp. 39–49.
- [SB02] Yannis Smaragdakis and Don Batory. “Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs”. In: *ACM Transactions on Software Engineering and Methodology* 11.2 (Apr. 2002), pp. 215–255. ISSN: 1049-331X. DOI: 10.1145/505145.505148.
- [SA20] Ferenc Attila Somogyi and Márk Asztalos. “Systematic review of matching techniques used in model-driven methodologies”. In: *Software and Systems Modeling* 19.3 (2020), pp. 693–720. DOI: 10.1007/s10270-019-00760-x.
- [Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Wien: Springer-Verlag, Dec. 5, 1973. URL: <https://archive.org/details/Stachowiak1973AllgemeineModelltheorie> (visited on 04/05/2017).
- [Sta+06] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development - technology, engineering, management*. Pitman, 2006. ISBN: 978-0-470-02570-3.
- [Ste+09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. 2nd. Addison-Wesley Professional, 2009. ISBN: 0321331885.
- [Ste10] Perdita Stevens. “Bidirectional model transformations in QVT: semantic issues and open questions”. English. In: *Software and Systems Modeling* 9.1 (2010), pp. 7–20. ISSN: 1619-1366. DOI: 10.1007/s10270-008-0109-9.
- [SF18] Klaas-Jan Stol and Brian Fitzgerald. “The ABC of Software Engineering Research”. In: *ACM Transactions on Software Engineering and Methodology* 27.3 (Sept. 2018). DOI: 10.1145/3241743.
- [SG05] Maximilian Störzer and Jürgen Graf. “Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software”. In: *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, 2005, pp. 653–656. DOI: 10.1109/ICSM.2005.99.
- [Str13] Bjarne Stroustrup. *The C++ programming language. 4th*. Addison-Wesley Professional, 2013.
- [Str+17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaella Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. “Henshin: A Usability-Focused Framework for EMF Model Transformation Development”. In: *Graph Transformation - 10th International Conference, ICGT 2017, Held as Part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings*. Vol. 10373. Lecture Notes in Computer Science. Springer, 2017, pp. 196–208. DOI: 10.1007/978-3-319-61470-0_12.
- [Str+18a] Daniel Strüber, Alexandru Burdusel, Stefan John, and Steffen Zschaler. “Henshin: A Model Transformation Language and its Use for Search-Based Model Optimisation in MDEOptimiser”. In: *Modellierung 2018*. Vol. P-280. LNI. Braunschweig, Germany: Gesellschaft für Informatik e.V., 2018, pp. 299–300. URL: <https://dl.gi.de/20.500.12116/14948>.
- [SPJ18] Daniel Strüber, Sven Peldszus, and Jan Jürjens. “Taming Multi-Variability of Software Product Line Transformations”. In: *Fundamental Approaches to Software Engineering, 21st International Conference, FASE 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Proceedings*. Thessaloniki, Greece, 2018, pp. 337–355. DOI: 10.1007/978-3-319-89363-1_19.

- [Str+18b] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. “Variability-based model transformation: formal foundation and application”. In: *Formal Aspects Comput.* 30.1 (2018), pp. 133–162. DOI: 10.1007/s00165-017-0441-3.
- [Str+15] Daniel Strüber, Julia Rubin, Marsha Chechik, and Gabriele Taentzer. “A Variability-Based Approach to Reusable and Efficient Model Transformations”. In: *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Vol. 9033. Lecture Notes in Computer Science. Springer, 2015, pp. 283–298. DOI: 10.1007/978-3-662-46675-9_19.
- [SS16] Daniel Strüber and Stefan Schulz. “A Tool Environment for Managing Families of Model Transformation Rules”. In: *Graph Transformation*. Cham: Springer International Publishing, 2016, pp. 89–101. ISBN: 978-3-319-40530-8.
- [Stü+20] Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. “Towards Multiple Model Synchronization with Comprehensive Systems”. In: *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Vol. 12076. Lecture Notes in Computer Science. Springer, 2020, pp. 335–356. DOI: 10.1007/978-3-030-45234-6_17.
- [Stü+21] Patrick Stünkel, Harald König, Adrian Rutle, and Yngve Lamo. “Multi-Model Evolution through Model Repair”. In: *Journal of Object Technology* 20.1 (2021), 1:1–25. DOI: 10.5381/jot.2021.20.1.a2.
- [Sun+21] Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. “Yet Another Textual Variability Language? A Community Effort towards a Unified Language”. In: *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Volume A*. New York, NY, USA: ACM, 2021, pp. 136–147. DOI: 10.1145/3461001.3471145.
- [SGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. “A taxonomy of variability realization techniques”. In: *Software: Practice and Experience* 35.8 (2005), pp. 705–754. DOI: 10.1002/spe.652.
- [SGM02] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software. Beyond Object-Oriented Programming*. 2nd ed. Addison-Wesley Component Software Series, 2002. ISBN: 0-201-74572-0.
- [Tae03] Gabriele Taentzer. “AGG: A Graph Transformation Environment for Modeling and Validation of Software”. In: *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003, Revised Selected and Invited Papers*. Vol. 3062. Lecture Notes in Computer Science. Springer, 2003, pp. 446–453. DOI: 10.1007/978-3-540-25959-6_35.
- [Tae99] Gabriele Taentzer. “AGG: A Tool Environment for Algebraic Graph Transformation”. In: *Applications of Graph Transformations with Industrial Relevance, International Workshop, AGTIVE'99*. Vol. 1779. Lecture Notes in Computer Science. Springer, 1999, pp. 481–488. DOI: 10.1007/3-540-45104-8_41.
- [Tae+17] Gabriele Taentzer, Rick Salay, Daniel Strüber, and Marsha Chechik. “Transformations of Software Product Lines: A Generalizing Framework Based on Category Theory”. In: *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017*. 2017, pp. 101–111. DOI: 10.1109/MODELS.2017.22.
- [Thü+14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. “A Classification and Survey of Analysis Strategies for Software Product Lines”. In: *ACM Computing Surveys* 47.1 (2014), 6:1–6:45. DOI: 10.1145/2580950.

- [TBK09] Thomas Thüm, Don S. Batory, and Christian Kästner. “Reasoning about edits to feature models”. In: *31st International Conference on Software Engineering, ICSE 2009*. IEEE, 2009, pp. 254–264. DOI: 10.1109/ICSE.2009.5070526.
- [Tis+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. “On the Use of Higher-Order Model Transformations”. In: *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*. Vol. 5562. Lecture Notes in Computer Science. Springer, 2009, pp. 18–33. DOI: 10.1007/978-3-642-02674-4_3.
- [TI06] Antoine Toulmé and I Inc. “Presentation of EMF compare utility”. In: *Eclipse Modeling Symposium*. Vol. 1. 2006, p. 2006.
- [Uhr11] Sabrina Uhrig. “Korrespondenzberechnung auf Klassendiagrammen”. PhD thesis. Universität Bayreuth, 2011.
- [Var+21] Zahra VaraminyBahnemiry, Jessie Galasso, Khalid Belharbi, and Houari Sahraoui. “Automated Patch Generation for Fixing Semantic Errors in ATL Transformation Rules”. In: *24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, Fukuoka, Japan, October 10-15, 2021*. IEEE, 2021, pp. 13–23. DOI: 10.1109/MODELS50736.2021.00011.
- [Var+18] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. “A classification of product sampling for software product lines”. In: *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*. ACM, 2018, pp. 1–13. DOI: 10.1145/3233027.3233035.
- [Voe10] Markus Voelter. “Implementing feature variability for models and code with projectional language workbenches”. In: *Proceedings of the Second International Workshop on Feature-Oriented Software Development, FOSD 2010, Eindhoven, Netherlands, October 10, 2010*. ACM, 2010, pp. 41–48. DOI: 10.1145/1868688.1868695.
- [Voe11] Markus Voelter. “Language and IDE Modularization and Composition with MPS”. In: *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*. Vol. 7680. Lecture Notes in Computer Science. Springer, 2011, pp. 383–430. DOI: 10.1007/978-3-642-35992-7_11.
- [Voe+13] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. “mbeddr: Instantiating a language workbench in the embedded software domain”. In: *Automated Software Engineering 20.3* (2013), pp. 339–390.
- [VIR10] Konrad Voigt, Petko Ivanov, and Andreas Rummler. “MatchBox: combined meta-model matching for semi-automatic mapping generation”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*. ACM, 2010, pp. 2281–2288. DOI: 10.1145/1774088.1774563.
- [VG07] Markus Völter and Iris Groher. “Product Line Implementation using Aspect-Oriented and Model-Driven Software Development”. In: *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*. IEEE Computer Society, 2007, pp. 233–242. DOI: 10.1109/SPLINE.2007.23.
- [Wac07] Guido Wachsmuth. “Metamodel Adaptation and Model Co-adaptation”. In: *ECOOP 2007 - Object-Oriented Programming*. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 600–624. DOI: 10.1007/978-3-540-73589-2_28.
- [Wag+12] Dennis Wagelaar, Ludovico Iovino, Davide Di Ruscio, and Alfonso Pierantonio. “Translational Semantics of a Co-evolution Specific Language with the EMF Transformation Virtual Machine”. In: *Theory and Practice of Model Transformations - 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings*. 2012, pp. 192–207. DOI: 10.1007/978-3-642-30476-7_13.

- [WO14] Eric Walkingshaw and Klaus Ostermann. “Projectional editing of variational software”. In: *Proc. 13th GPCE*. Västerås, Sweden, Sept. 2014, pp. 29–38.
- [WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product Line Engineering. A Family-Based Software Development Process*. Addison Wesley Professional, 1999.
- [Wes15] Bernhard Westfechtel. “A Case Study for a Bidirectional Transformation Between Heterogeneous Metamodels in QVT Relations”. In: *Evaluation of Novel Approaches to Software Engineering - 10th International Conference, ENASE 2015, Revised Selected Papers*. Vol. 599. Communications in Computer and Information Science. Springer, 2015, pp. 141–161. DOI: 10.1007/978-3-319-30243-0_8.
- [Wes14] Bernhard Westfechtel. “Merging of EMF models - Formal foundations”. In: *Software and Systems Modeling* 13.2 (2014), pp. 757–788. DOI: 10.1007/s10270-012-0279-3.
- [WMC01] Bernhard Westfechtel, Björn P. Munch, and Reidar Conradi. “A Layered Architecture for Uniform Version Management”. In: *IEEE Transactions on Software Engineering* 27.12 (2001), pp. 1111–1133. DOI: 10.1109/32.988710.
- [Whi+09] Jon Whittle, Praveen K. Jayaraman, Ahmed M. Elkhodary, Ana Moreira, and João Araújo. “MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation”. In: *Lecture Notes in Computer Science VI* (2009), pp. 191–237. DOI: 10.1007/978-3-642-03764-1_6.
- [Wil17] Edward D. Willink. “The Micromapping Model of Computation; The Foundation for Optimized Execution of Eclipse QVTc/QVTr/UMLX”. In: *Theory and Practice of Model Transformation - 10th International Conference, ICMT@STAF 2017*. Vol. 10374. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 51–65. DOI: 10.1007/978-3-319-61473-1_4.
- [Wim+11a] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger. “From the Heterogeneity Jungle to Systematic Benchmarking”. In: *Models in Software Engineering. Workshops and Symposia at MODELS 2010*. Vol. 6627. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 150–164. DOI: 10.1007/978-3-642-21210-9_15.
- [Wim+11b] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger, and Elizabeth Kapsammer. “A Survey on UML-based Aspect-oriented Design Modeling”. In: *ACM Computing Surveys* 43.4 (Oct. 2011), 28:1–28:33. DOI: 10.1145/1978802.1978807.
- [WP10] Stefan Winkler and Jens von Pilgrim. “A survey of traceability in requirements engineering and model-driven development”. In: *Software and Systems Modeling* 9.4 (2010), pp. 529–565. DOI: 10.1007/s10270-009-0145-0.
- [Woh+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012. DOI: 10.1007/978-3-642-29044-2.
- [ZHJ03] Tewfik Ziadi, Loïc Hérouët, and Jean-Marc Jézéquel. “Towards a UML Profile for Software Product Lines”. In: *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Revised Papers*. Vol. 3014. Lecture Notes in Computer Science. Springer, 2003, pp. 129–139. DOI: 10.1007/978-3-540-24667-1_10.
- [ZJ06] Tewfik Ziadi and Jean-Marc Jézéquel. “Software Product Line Engineering with the UML: Deriving Products”. In: *Software Product Lines - Research Issues in Engineering and Management*. Springer, 2006, pp. 557–588. DOI: 10.1007/978-3-540-33253-4_15.
- [Zsc+09] Steffen Zschaler, Pablo Sánchez, João Pedro Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. “VML* - A Family of Languages for Variability Management in Software Product Lines”. In: *Software Language Engineering, Second International Conference, SLE 2009 Revised Selected Papers*. Vol. 5969. Lecture Notes in Computer Science. Springer, 2009, pp. 82–102. DOI: 10.1007/978-3-642-12107-4_7.

B.2 (Co-)Authored Publications Related with Thesis

- [Gre19] Sandra Greiner. “On Extending Single-Variant Model Transformations for Reuse in Software Product Line Engineering”. In: *ESEC/FSE '19: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn, Estonia: ACM, 2019, pp. 1160–1163. DOI: 10.1145/3338906.3341467.
- [GNS22] Sandra Greiner, Michael Nieke, and Christoph Seidl. “Towards Trace-Based Synchronization of Variability Annotations in Evolving Model-Driven Product Lines”. In: *VaMoS '22: Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. Florence, Italy: ACM, 2022. DOI: 10.1145/3510466.3510470.
- [GSW17] Sandra Greiner, Felix Schwägerl, and Bernhard Westfechtel. “Realizing Multi-variant Model Transformations on Top of Reused ATL Specifications”. In: *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*. Porto, Portugal: SCITEPRESS Science and Technology Publications, Portugal, Feb. 2017, pp. 362–373. DOI: 10.5220/0006137803620373.
- [GW18a] Sandra Greiner and Bernhard Westfechtel. “Evaluating Multi-variant Model-To-Text Transformations Realized by Generic Aspects”. In: *Model-Driven Engineering and Software Development: 6th International Conference, MODELSWARD*. Vol. 991. Communications in Computer and Information Science. Cham: Springer International Publishing, 2018, pp. 82–105. DOI: 10.1007/978-3-030-11030-7_5.
- [GW19a] Sandra Greiner and Bernhard Westfechtel. “Evaluating the Multi-variant Model Transformation of UML Class Diagrams to Java Models”. In: *Model-Driven Engineering and Software Development: 7th International Conference, MODELSWARD 2019, Revised Selected Papers*. Vol. 1161. Communications in Computer and Information Science. Cham: Springer International Publishing, 2019, pp. 275–297. DOI: 10.1007/978-3-030-37873-8_12.
- [GW18b] Sandra Greiner and Bernhard Westfechtel. “Generating Multi-Variant Java Source Code Using Generic Aspects”. In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*. 2018, pp. 36–47. DOI: 10.5220/0006536700360047.
- [GW19b] Sandra Greiner and Bernhard Westfechtel. “Generic Framework for Evaluating Commutativity of Multi-Variant Model Transformations”. In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*. SciTePress, 2019, pp. 155–166. DOI: 10.5220/0007585701570168.
- [GW18c] Sandra Greiner and Bernhard Westfechtel. “Improving Trace-Based Propagation of Feature Annotations in Model Transformations”. In: *Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AM-MoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*. Vol. 2245. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 584–593. URL: http://ceur-ws.org/Vol-2245/me_paper_2.pdf.
- [GW19c] Sandra Greiner and Bernhard Westfechtel. “On Determining Variability Annotations In Partially Annotated Models”. In: *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS*. ACM, 2019, 17:1–17:10. DOI: 10.1145/3302333.3302341.

- [GW21] Sandra Greiner and Bernhard Westfechtel. “On Preserving Variability Consistency in Multiple Models”. In: *VaMoS '21: Proceedings of the 15th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, 2021, 7:1–7:10. DOI: 10.1145/3442391.3442399.
- [GW20] Sandra Greiner and Bernhard Westfechtel. “Towards iterative software product line engineering with incremental multi-variant model transformations”. In: *VaMos '20: Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. ACM, 2020, 22:1–22:9. DOI: 10.1145/3377024.3377032.
- [BG16a] Thomas Buchmann and Sandra Greiner. “Bidirectional Model Transformations Using a Handcrafted Triple Graph Transformation System”. In: *Software Technologies. ICSOFT 2016, Revised Selected Papers*. Vol. 743. Communications in Computer and Information Science. Springer, 2016, pp. 201–220. DOI: 10.1007/978-3-319-62569-0_10.
- [BG16b] Thomas Buchmann and Sandra Greiner. “Handcrafting a Triple Graph Transformation System to Realize Round-trip Engineering Between UML Class Models and Java Source Code”. In: *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - ICSOFT-PT*. SciTePress, 2016, pp. 27–38. DOI: 10.5220/0005957100270038.
- [BG18] Thomas Buchmann and Sandra Greiner. “Managing Variability in Models and Derived Artefacts in Model-driven Software Product Lines”. In: *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018*. 2018, pp. 326–335. DOI: 10.5220/0006563403260335.
- [WG20a] Bernhard Westfechtel and Sandra Greiner. “Extending single- to multi-variant model transformations by trace-based propagation of variability annotations”. In: *Software and Systems Modeling* 19.4 (2020), pp. 853–888. DOI: 10.1007/s10270-020-00791-9.
- [WG18] Bernhard Westfechtel and Sandra Greiner. “From Single- to Multi-Variant Model Transformations: Trace-Based Propagation of Variability Annotations”. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*. ACM, 2018, pp. 46–56. DOI: 10.1145/3239372.3239414.
- [WG20b] Bernhard Westfechtel and Sandra Greiner. “Trace-Based Propagation of Variability Annotations”. In: *Software Engineering 2020*. Bonn: Gesellschaft für Informatik e.V, 2020, pp. 59–60. DOI: 1018420/SE2020_16.

B.3 Further (Co-)Authored Publications

- [Ana+22a] Sofia Ananieva, Sandra Greiner, Timo Kehrer, Jacob Krüger, Thomas Kühn, Lukas Linsbauer, Sten Grüner, Anne Koziolk, Henrik Lönn, S. Ramesh, and Ralf H. Reussner. “A conceptual model for unifying variability in space and time: Rationale, validation, and illustrative applications”. In: *Empirical Software Engineering* 27.5 (2022), p. 101. DOI: 10.1007/s10664-021-10097-z.
- [Ana+22b] Sofia Ananieva, Sandra Greiner, Jacob Krueger, Lukas Linsbauer, Sten Gruener, Timo Kehrer, Thomas Kuehn, Christoph Seidl, and Ralf Reussner. “Unified Operations for Variability in Space and Time”. In: *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*. VaMoS '22. Florence, Italy: ACM, 2022. DOI: 10.1145/3510466.3510483.
- [Ana+20] Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolk, Henrik Lönn, Sebastian Krieter, Christoph Seidl, S. Ramesh, Ralf H. Reussner, and Bernhard Westfechtel. “A conceptual model for unifying variability in space and time”. In: *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Volume A*. ACM, 2020, 15:1–15:12. DOI: 10.1145/3382025.3414955.
- [Ger+21] Lea Gerling, Sandra Greiner, Kristof Meixner, and Gabriela Karoline Michelon. “Fourth International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2021)”. In: *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*. SPLC '21. Leicester, United Kingdom: ACM, 2021, p. 204. DOI: 10.1145/3461001.3473055.
- [GB16] Sandra Greiner and Thomas Buchmann. “Round-trip Engineering UML Class Models and Java Models: A Real-world Use Case for Bidirectional Transformations with QVT-R”. In: *International Journal on Information System Modeling and Design* 7.3 (2016), pp. 72–92. DOI: 10.4018/IJISMD.2016070104.
- [GBW16] Sandra Greiner, Thomas Buchmann, and Bernhard Westfechtel. “Bidirectional Transformations with QVT-R: A Case Study in Round-trip Engineering UML Class Models and Java Source Code”. In: *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development, MODELSWARD*. 2016, pp. 15–27. DOI: 10.5220/0005644700150027.

Acknowledgments

Writing this thesis in times of a pandemic and an uprising war was not an easy thing to go through. Therefore, I am deeply grateful to precious friends and family members (Tom and Sam, particularly) who lifted me up and helped me in various ways to stay focused. Most importantly, my parents encouraged me to study and without their almost unconditional support everything would have been significantly harder: Mama, Papa, ein riesiges Vergelt's Gott für alles!

From an organizational point of view: Monika Glaser, Bern Schlesier, and Simon Melzner, Danke dafür, dass Sie/Ihr den 'Betrieb so gut am Laufen haltet'! Particularly, Simon's technical almost Rund-Um-Die-Uhr support already during studying is amazing and was always a great help!

Chatting and discussing with a coffee or beer helped to refresh and collect valuable thoughts: Thanks to all colleagues and anyone at the university of Bayreuth who I had the pleasure to work with and/or spend beneficial times: Felix Schwägerl, Thomas Buchmann, Nikita Dümmel, Johannes Schröpfer, Sabrina Uhrig, Johannes Doleschal, Matthias Niewirth, Tina Trautner, Christian Knauer, Wim Martens, Matthias Stachowski, Johannes Seifert, Matthias Korch, Marvin Ferber, Tobias and Tim Werner, Oleg Lobachev, Lars Ackermann, Stefan Schönig, Christian Sturm, and Matthias Ehmann. Particularly, to Natalia Kallinik, who is simply an amazing person, thanks so much! Similarly, a huge thanks to all colleagues in my vicinity in the open-space office at the ITU Copenhagen for being so welcoming and integrating, here particularly, to Holger Stadel-Borum and Adrian Hoff.

Support in the every day working life came from a lot of great student assistants, whom I had the chance to guide but also to learn from: First and foremost, thanks to Philipp Dennerlein for all the feedback in improving the exercises and encouraging students, but not the least to: Erik Rill, Oliver Hacker, Jonas Fraas, Moritz Johlige, Sebastian Kaske, Matthias Bank, Ugur Can Özbay, Christopher Brunner, Christian Birzer, Benjamin Schlosser, Lukas Bachmeier, Niklas Bauriedl, Lisa Hartl, Heiko Bächmann, Sabine Fischer, and Michelle Reimann.

Turning to the thesis itself: First, several students lay foundations for some of the contributions in technical prototypes or (inclusive ;) fruitful discussions: Marco Dmitrow, Johannes Doleschal, Christoph Vogler, Sebastian Petter, Michael Bösch, Gerard Tehano, Lukas Schwarz, Sebastian Kaske, Valentin Strotzer, and Fabian Höllerich. Second, I am thankful to get feedback in various forms from several computer science academics I was able to present and discuss my work with: Sven Apel, Norbert Siegmund, Christian Kästner, Thorsten Berger, Jacob Krüger, Regina Hebig, Michael Nieke, Andrzej Wasowski, Mahsa Varshosaz, Raúl Pardo, Michael Siebers, Ute Schmid, Johannes Rabold, Johannes Meyer, Manuel Ohrndorf, Thomas Thüm, Chico Sundermann, Alexander Schultheiß, Paul Bittner, Christof Tinnes, Christian Kaltenecker, and Kristof Meixner. For having the pleasure to work with, I like to thank Philippe Collet, Lea Gerling, Gabriela Michelon, Thomas Kühn, Sofia Ananieva, Lukas Linsbauer, Sten Grüner, Ralf Reussner, Anne Koziolk, Henrik Lönn, and S. Ramesh. Particularly, I like to emphasize all the support and advice given by Christoph Seidl during my research visit in Copenhagen and by Timo Kehrer in our joint community research!

Keeping more than 350 pages without an exploding number of typos is only possible with several good proof readers. Thanks in these respects to Angela and Serena, my Mama, Anna W., Erik, Olli, Tobi, and Kalti for proofreading and commenting on parts of this thesis.

Preparing the 'defense' also weighs as an important part: I highly appreciate the challenging questions with which Oscar Nierstraz, Alexandre Bergle, Timo Kehrer, and Christos Tsigkanos prepared me as well as their mental uplifting, and Sebastian Krieter, Elias Kuitert, Alex Schultheiss, and Paul Bittner's tips and comments on the slides. Final fine-tuning was supported by Johannes Dorn, Stefan Mühlbauer, Sabine, Michelle, Tobi, Philipp, Christian B., Erik, Christian K, Christof, Paul, and Alex.

For serving in the committee, final questions, and comments, I like to thank Prof. Stefan Jablonski and Prof. Thomas Rauber. For additional valuable advice and always keeping an eye on the practical perspective of my work, I highly appreciate inspiring comments from my external reviewer, Rick Rabiser. And lastly, the most important thanks has to be granted to my supervisor, Bernhard Westfechtel. I am extremely grateful for the freedom he gave me to broaden my horizons in various aspects during my PhD. He certainly changed my way of approaching and solving software engineering problems. I highly appreciate and value all the comments and feedback that I received in this time.

Several more people positively affected developing necessary skills and knowledge during working on my PhD. To anyone I do not mention above I am equally thankful for contributing to where I am right now!