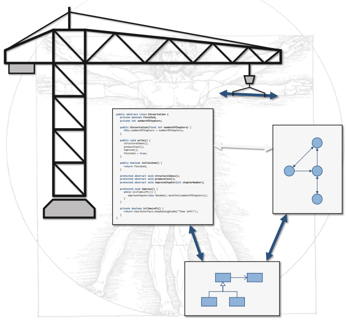
The Karlsruhe Series on
Software Design
and Quality

34

Building Transformation Networks for Consistent Evolution of Interrelated Models

Heiko Klare



KIT Scientific Publishing

Heiko Klare

Building Transformation Networks for Consistent Evolution of Interrelated Models

The Karlsruhe Series on Software Design and Quality
Volume 34

Dependability of Software-intensive Systems group
Faculty of Computer Science
Karlsruhe Institute of Technology

and

Software Engineering Division
Research Center for Information Technology (FZI), Karlsruhe

Editor: Prof. Dr. Ralf Reussner

Building Transformation Networks for Consistent Evolution of Interrelated Models

by
Heiko Klare

Scientific Publishing

Karlsruher Institut für Technologie
KASTEL – Institut für Informationssicherheit und Verlässlichkeit

Building Transformation Networks for Consistent
Evolution of Interrelated Models

Zur Erlangung des akademischen Grades eines Doktors der
Ingenieurwissenschaften von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT) genehmigte Dissertation

von Heiko Klare

Tag der mündlichen Prüfung: 19. Mai 2021
Erster Referent: Prof. Dr. Ralf H. Reussner
Zweiter Referent: Prof. Dr. Colin Atkinson (Universität Mannheim)

Impressum

KIT Scientific Publishing

Karlsruher Institut für Technologie (KIT)
KIT Scientific Publishing
Straße am Forum 2
D-76131 Karlsruhe

www.ksp.kit.edu

Print on Demand 2022 – Gedruckt auf FSC-zertifiziertem Papier

ISSN 1867-0067
ISBN 978-3-7315-1132-8
DOI: 10.5445/KSP/1000138566

Abstract

In this thesis, we formalize and analyze how to preserve consistency between multiple artifacts describing the same software system through the combination of transformations between themand support it with appropriate methods.

During the development of a software system, the developers and further stakeholders employ multiple languages or, in general, tools to describe different concerns. Code often represents the central artifact, which is, however, implicitly or explicitly complemented by specifications of the architecture, deployment, requirements and others. In addition to the programming language, further languages are used to specify these artifacts, such as the UML for object-oriented design or architecture models, the OpenAPI standard for interface definitions, or Docker for deployment specifications. To achieve a functional software system, all these artifacts must depict a uniform, non-contradicting specification of the whole system. Interfaces of services must, for example, be represented in all these artifacts uniformly. We say that the artifacts have to be *consistent*.

In model-driven software development, such artifacts are denoted as *models* and represent central units of the development process, from which also at least parts of the program code can be derived. This is, for example, already applied in automotive software development. A common means to preserve consistency between models are transformations, which adapt the other models after one of them was changed. Existing research is restricted to transformations that preserve consistency between pairs of models or to project-specific combinations of transformations to preserve consistency of multiple models. A systematic development process that enables the independent development of transformations and their modular reuse in different contexts is, however, not yet supported.

In this thesis, we research how developers can combine multiple transformations to a network that is able to execute these transformations in an order such that all resulting models are consistent. To this end, we assume that each transformation between two languages is developed independently and that the transformations cannot be aligned with each other. Our contributions are separated into those concerning the correctness of such a combination of transformations to a network and those concerning the optimization of quality properties of such a network.

We first derive and precisely define an appropriate notion of correctness for transformation networks. It induces three specific requirements, which are a *synchronization* property of the single transformations, a *compatibility* property of a network of transformations, and finding an appropriate *orchestration*, i.e., an execution order of the transformations. We propose a construction approach for transformations to fulfill the synchronization property with existing transformation specification languages on a formally proven property. For this approach, we show completeness and appropriateness with a case-study-based empirical evaluation in the domain of component-based software engineering. We formally define compatibility of transformations, for which we propose a formal analysis, which is proven correct, and derive a practical analysis, whose applicability we demonstrate with case studies. Finally, we define the orchestration problem of finding an orchestration that delivers consistent models whenever such an orchestration exists. We prove undecidability of that problem and discuss that restrictions to achieve its decidability will likely limit practical applicability. For that reason, we propose an algorithm that conservatively approaches the problem. It guarantees to deliver an orchestration under specific, well-defined conditions and otherwise indicates an error. We prove correctness of the algorithm and a property that supports finding the cause whenever the algorithm fails. Additionally, we categorize errors that can occur if a transformation network does not fulfill the defined correctness notion, from which we derive by means of the mentioned case studies that most potential errors can be avoided by construction with the approaches that we propose in this thesis.

The investigation of quality properties of transformation networks is based on a classification of relevant properties and of the effects of different types of network topologies on them. It reveals that especially correctness and reusability are contradictory, thus the selection of a network topology induces a trade-off between these properties. We derive a construction approach for transformation networks that mitigates the necessary trade-off decision and, under specific assumptions, guarantees correctness by construction. We support the development process for this approach with a specialized specification language. While trade-off mitigation is given by construction of the approach, we show achievability of the assumptions and benefits of the proposed language in an empirical evaluation using the case study from component-based software engineering.

The contributions of this thesis support researchers as well as transformation developers and users of transformations in analyzing and constructing networks of transformations. They depict systematic knowledge about correctness and further quality properties of transformation networks for researchers and transformation developers. In particular, they show precisely which parts of these properties can be achieved by construction, which can be validated by analysis, and which errors must inevitably be expected during execution. Along with these insights, we provide concrete, practically applicable approaches for the construction, analysis and execution of correct and modularly reusable transformation networks, from which developers and users of transformation networks both benefit.

Zusammenfassung

In dieser Dissertation formalisieren und analysieren wir die Konsistenzerhaltung verschiedener Artefakte zur Beschreibung eines Softwaresystems durch die Kopplung von Transformationen zwischen diesen und unterstützen sie mit geeigneten Methoden.

Für die Entwicklung eines Softwaresystems nutzen Entwickler:innen und weitere Beteiligte verschiedene Sprachen, oder allgemein Werkzeuge, zur Beschreibung unterschiedlicher Belange. Meist stellt Programmcode das zentrale Artefakt dar, welches jedoch, implizit oder explizit, durch Spezifikationen von Architektur, Deployment, Anforderungen und anderen ergänzt wird. Neben der Programmiersprache verwenden die Beteiligten weitere Sprachen zur Spezifikation dieser Artefakte, beispielsweise die UML für Modelle des objektorientierten Entwurfs oder der Architektur, den OpenAPI-Standard für Schnittstellen-Definitionen, oder Docker für Deployment-Spezifikationen. Zur Erstellung eines funktionsfähigen Softwaresystems müssen diese Artefakte das System einheitlich und widerspruchsfrei darstellen. Beispielsweise müssen Dienst-Schnittstellen in allen Artefakten einheitlich repräsentiert sein. Wir sagen, die Artefakte müssen *konsistent* sein.

In der modellgetriebenen Entwicklung werden solche verschiedenen Artefakte allgemein *Modelle* genannt und bereits als wesentliche zentrale Entwicklungsbestandteile genutzt, um auch Teile des Programmcodes aus ihnen abzuleiten. Dies betrifft beispielsweise die Softwareentwicklung für Fahrzeuge. Zur Konsistenzerhaltung der Modelle werden oftmals Transformationen eingesetzt, die nach Änderungen eines Modells die anderen Modelle anpassen. Die bisherige Forschung beschränkt sich auf Transformationen zur Konsistenzerhaltung zweier Modelle und die projektspezifische Kombination von Transformationen zur Konsistenzerhaltung mehrerer Modelle. Ein systematischer Entwicklungsprozess, in dem einzelne Transformationen unabhängig entwickelt und in verschiedenen Kontexten modular wiederverwendet werden können, wird hierdurch jedoch nicht unterstützt.

In dieser Dissertation erforschen wir, wie Entwickler:innen mehrere Transformationen zu einem Netzwerk kombinieren können, welches die Transformationen in einer geeigneten Reihenfolge ausführen kann, sodass abschließend alle Modelle konsistent zueinander sind. Dies geschieht unter der Annahme, dass einzelne Transformationen zwischen zwei Sprachen unabhängig voneinander entwickelt werden und daher nicht aufeinander abgestimmt werden können. Unsere Beiträge unterteilen sich in die Untersuchung der Korrektheit einer solchen Kombination von Transformationen zu einem Netzwerk und die Optimierung von Qualitätseigenschaften solcher Netzwerke.

Wir diskutieren und definieren zunächst einen adäquaten Korrektheitsbegriff, welcher drei Anforderungen impliziert. Diese umfassen eine *Synchronisations*-Eigenschaft für die einzelnen Transformationen, eine *Kompatibilitäts*-Eigenschaft für das Transformationsnetzwerk, sowie das Finden einer geeigneten Ausführungsreihenfolge der Transformationen, einer *Orchestrierung*. Wir stellen ein Konstruktionsverfahren für Transformationen vor, mit welchem die Synchronisations-Eigenschaft basierend auf einer formal bewiesenen Eigenschaft erfüllt wird. Für dieses zeigen wir Vollständigkeit und Angemessenheit mit einer fallstudienbasierten empirischen Evaluation in der Domäne der komponentenbasierten Softwareentwicklung. Wir definieren die Eigenschaft der Kompatibilität von Transformationen, für welche wir ein formales und bewiesen korrektes Analyseverfahren vorschlagen und eine praktische Realisierung ableiten, deren Anwendbarkeit wir in Fallstudien nachweisen. Schlussendlich definieren wir das Orchestrierungsproblem zum Finden einer Orchestrierung, die zu konsistenten Modelle führt wann immer solch eine Orchestrierung existiert. Wir beweisen die Unentscheidbarkeit dieses Problems und diskutieren, dass eine Einschränkung des Problems, um Entscheidbarkeit zu erreichen, die Anwendbarkeit unpraktikabel beschränken würde. Daher schlagen wir einen Algorithmus vor, der das Problem konservativ behandelt. Er findet eine Orchestrierung unter bestimmten, wohldefinierten Bedingungen und terminiert andernfalls mit einem Fehler. Wir beweisen die Korrektheit des Algorithmus und eine Eigenschaft, die das Finden der Ursache im Fehlerfall unterstützt. Zusätzlich kategorisieren wir Fehler, die auftreten können falls ein Netzwerk den definierten Korrektheitsbegriff nicht erfüllt. Daraus leiten wir mittels den bereits genannten Fallstudien ab, dass die meisten potentiellen Fehler per Konstruktion mit den in dieser Arbeit vorgeschlagenen Ansätzen vermieden werden können.

Zur Untersuchung von Qualitätseigenschaften eines Netzwerkes von Transformationen klassifizieren wir zunächst relevante Eigenschaften, sowie den

Effekt verschiedener Typen von Netzwerktopologien auf diese. Hierbei zeigt sich, dass insbesondere Korrektheit und Wiederverwendbarkeit im Widerspruch stehen, sodass die Wahl der Netzwerktopologie ein Abwägen bei der Optimierung dieser Eigenschaften erfordert. Wir leiten hieraus ein Konstruktionsverfahren für Transformationsnetzwerke ab, welches die Notwendigkeit einer Abwägung zwischen den Qualitätseigenschaften abmildert und, unter gewissen Voraussetzungen, Korrektheit per Konstruktion gewährleistet. Wir unterstützen den Entwicklungsprozess für diesen Ansatz mithilfe einer spezialisierten Spezifikationssprache. Während die Verminderung der Notwendigkeit einer Abwägung zwischen Qualitätseigenschaften durch den Ansatz per Konstruktion erreicht wird, zeigen wir die Erreichbarkeit der Voraussetzungen und die Vorteile der vorgeschlagenen Sprache in einer empirischen Evaluation mithilfe der Fallstudie aus der komponentenbasierten Softwareentwicklung.

Die Beiträge dieser Dissertation unterstützen sowohl Forscher:innen als auch Transformationsentwickler:innen und Transformationsanwender:innen bei der Analyse und Konstruktion von Netzwerken von Transformationen. Sie stellen für Forscher:innen und Transformationsentwickler:innen systematisches Wissen über die Korrektheit und weitere Qualitätseigenschaften solcher Netzwerke bereit. Sie zeigen insbesondere welche Teile dieser Eigenschaften per Konstruktion erreicht werden können, welche per Analyse validiert werden können, und welche Fehler unvermeidbar bei der Ausführung erwartet werden müssen. Zusätzlich zu diesen Einsichten stellen wir konkrete, praktisch nutzbare Verfahren bereit, mit denen Transformationsentwickler:innen und Transformationsanwender:innen korrekte, modular wiederverwendbare Netzwerke konstruieren, analysieren und ausführen können.

Acknowledgements

A dissertation is not written by one person alone. So many people are involved in that process in many different ways that I cannot thank and address all of them by name. Therefore, a first big thanks goes to everyone who supported me in this adventure.

My deepest thanks go to Ralf. He gave me the opportunity to work as a researcher and to write a dissertation, and he inspired me in countless discussions. It was a pleasure to work on visions and research directions at eye level and to see that my opinions are valued. But most importantly, he always had a friendly ear and gave support whenever and for whatever reason that was necessary.

I thank Colin for all the great and inspiring meetings we had and all the valuable and constructive feedback he gave me. Having all these talks with someone from outside our chair clearly helped me to think outside of the box and better frame my ideas.

I am grateful to everyone at SDQ who worked with me, discussed ideas with me, or simply made the time enjoyable. Special thanks go the Vitruvius guys Max, Michael, and Erik. Without them, I would not even have considered writing a dissertation. And without all the great students that supported me in realizing my ideas, it would have taken at least twice the time to finish this thesis. Therefore, particular thanks go to Aurélien, Joshua, Timur, and Torsten. I am especially thankful for all the friends I made along the way. Representative for them, I thank Max for being a friend for years now, Dominik and Stephan for all the great help and the fun we had, and Sandro for many sympathetic talks and the guidance in baking sourdough bread.

I thank my family and my friends for all the love, the sympathy, and every possible way of support, which allowed me to frame my life so that I could place this thesis into it. I am so lucky and infinitely grateful to be surrounded by such people, who take me as I am and stand by me for so many year. These final thanks go to my parents, to Kati, to Marius, and to Mirco.

Contents Overview

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Definitions

List of Theorems

Acronyms

API Application Programming Interface. 32, 34, 42, 44, 269, 358, 436, 456, 489

ASP Answer Set Programming. 486, 498

ATL Atlas Transformation Language. 44, 179, 486, 489, 499

DSL Domain-Specific Language. 30, 42, 489

ECU Electronic Control Unit. 4

EMF Eclipse Modeling Framework. 32, 34, 35, 43–45, 87, 222, 320, 380, 481, 489

EMOF Essential Meta Object Facility. xxi, 32–34, 223, 380

MDA Model-Driven Architecture. 31, 40, 320, 480, 494

MDSD Model-Driven Software Development. 4, 31, 35, 39, 479, 492, 494

MGG Multi Graph Grammar. 496

MOF Meta Object Facility. 32, 33, 43, 45, 60

OCL Object Constraint Language. 43, 61, 159, 160, 164, 166, 168, 172–176, 319–321, 324, 325, 327, 328, 473, 486, 498, 499, 503

OSM Orthographic Software Modeling. 37, 419, 484

PCM Palladio Component Model. xxi, xxiii, xxv, 5–7, 10–14, 16, 21, 30, 38–40, 44–48, 52, 55, 117–119, 122, 124, 181, 196, 197, 219, 220, 233, 234, 338–345, 347, 352, 353, 359, 360, 363–365, 382, 387, 388, 393, 396, 397, 400–403, 407, 412–417, 420–422, 432, 433, 438, 445, 446, 457–459, 461, 463

QVT Query/View/Transformation. 43, 158, 159, 488

QVT-O QVT Operations. 43, 74, 107, 179, 252, 488

QVT-R QVT Relations. xxiii, xxvii, 43, 44, 74, 98, 107, 114, 153, 154, 158–166, 168, 172, 174–176, 179, 220, 226, 252, 295, 317, 319–324, 328, 356, 358, 443, 445–448, 471, 486, 488, 489, 496, 503, 517

QVTd QVT Declarative. 320, 321, 517

RUSP Ready to Use Software Products. 7, 13, 21

SLoC Source Lines of Code. 453, 456, 466, 467, 471

SMT Satisfiability Modulo Theories. 153, 173–176, 320, 324, 325, 327, 328, 330, 499

SUM Single Underlying Model. xxiii, 36–38, 395, 396, 484, 486

SUM metamodel Single Underlying Metamodel. 36, 37, 413

TGG Triple Graph Grammar. 44, 74, 179, 190, 295, 489, 490, 496, 506

UML Unified Modeling Language. i, v, xxi, xxiii, xxv, 4, 6, 7, 10–14, 16, 29, 30, 32–34, 38–41, 44–48, 52, 53, 59, 79, 85, 104, 117–120, 122, 124, 181, 219, 233–235, 264, 338–345, 347–349, 352, 353, 359, 360, 363–366, 382, 387, 388, 393–404, 406, 409, 410, 412–417, 420–422, 428, 432, 433, 436, 438–440, 445, 446, 457–459, 461–463, 465–468, 480, 483

V-SUM Virtual Single Underlying Model. 38, 39, 420–422, 484, 485

V-SUM metamodel Virtual Single Underlying Metamodel. xxi, 38, 39, 420, 422

VIATRA VIsual Automated model TRAnsformations. 43, 179, 489

XML Extensible Markup Language. 5, 380

Part I.

Prologue

1. Introduction

In this thesis, we discuss how multiple artifacts used to develop a software or software-intensive system can be kept consistent by combining transformations between their specification languages. We research how multiple transformations, which specify consistency and its preservation, can be developed *independently*, such that their combination operates *correctly* and such that they can be reused *modularly*.

In the following sections, we first introduce the context of preserving consistency between multiple artifacts and identify existing challenges. We then derive two problem statements from these challenges and define a research goal along with fine-grained questions, as well as according contributions that counter these challenges. Finally, we give an overview of the structure of this thesis and give guidelines how to read it.

1.1. Consistency of Multiple Models

Engineers develop software and software-intensive technical systems of ever increasing scale. This leads to a continual increase in complexity of the artifacts used to describe such systems [MBF11]. As a direct consequence of the increasing system sizes, engineers inevitably have to deal with their inherent *essential complexity*. Various tools support the development process by reducing the *accidental complexity* to allow engineers to focus on handling the essential complexity [Bro87; FM08].

1.1.1. Consistency in System Engineering

To better handle the essential complexity of a system, engineers usually use multiple tools to describe and analyze different parts or properties of a system under development in different artifacts [Fra+18]. In the following, we denote all these artifacts as *models*, according to the notion of Bézivin that "everything is a model" [Béz05], including source code, for example, written in Java [Hei+09a]. This reduces the information to deal with to what is relevant for the development task of each person's role [Ste18]. In classical engineering disciplines like construction, mechanical and electrical engineering, this has been common practice for a long time and is often called *Model-Based Software Engineering (MBSE)* [Est08]. For example, developers of software for Electronic Control Units (ECUs) in automobiles use different tools and standards for specifying the system and software architecture, such as SysML [SysML] or AUTOSAR [Sch15], for defining the behavior, such as MATLAB/Simulink [Simu] or ASCET [ASCET], and for defining the deployment on multi-core architectures, such as Amalthea [Wol+15]. In software engineering, such a development methodology is also getting growing attention. It is often referred to as *Model-Driven Software Development (MDSD)* [Sta+06]. Such a development process considers other artifacts beyond code as primary artifacts to describe the system under construction. While code focuses on specifying the functionality of a system, other tools can be used, for example, to explicitly define the software architecture and its deployment, such as the UML [UML], analyzing and predicting the software performance, such as the Palladio Simulator [Reu+16], and for specifying requirements, like IBM Rational Doors [Lap13].

While this *fragmentation* of information across models developed with different tools eases dealing with the essential complexity of a system, it increases accidental complexity. Since all these models describe the same system, they usually share an overlap of information in terms of implicit *dependencies* or *redundancies*. If modifications in overlapping information are not propagated correctly across all dependencies and redundancies, *inconsistencies* can occur. For example, requirements changes have to be reflected in the software architecture and implementation, and modifications of the architecture must be reflected in the code. Since systems are usually developed iteratively and incrementally, dependencies are not directional but, in general, every model can be changed and require updates of others.

The overlaps of information, for example in tools for ECU software development [GHN10], are often not documented explicitly [Maz+17], but only known by engineers. Performing the task of updating overlapping information manually is, however, time-consuming and error-prone [Sax+17]. The automation of checking and of preserving consistency of information is still poorly supported in current development processes for large systems [Gui+18; PMR16; CCP19]. Automating that process is, however, necessary to reduce the accidental complexity induced by information fragmentation across multiple models.

A common approach to automate the process of checking and preserving consistency of models are *incremental model transformations*, which have already been applied in industrial scenarios [GW09; GHN10]. Tools describe their models in specific languages, for example denoted by XML schemes. A transformation specifies how models of one or multiple such languages have to be updated after engineers make changes to a model of another language. The subclass of *bidirectional* model transformations [Ste10], which specify the relations between two models and routines that restore consistency of their instances after any of them was changed, is particularly well researched [Cle+19; Kah+19]. System development usually involves more than two tools, and thus models of more than two languages have to be kept consistent. The use of transformations to check and preserve consistency between more than two models is, however, less researched [Ste20b]. It recently gained attention in a dedicated Dagstuhl seminar [Cle+19].

1.1.2. Distributed and Reusable Consistency Knowledge

Two general transformation-based approaches for preserving consistency of multiple models are *multidirectional* transformations and combining multiple bi- or multidirectional transformations to networks of them. In theory, a single multidirectional transformation provides higher expressiveness [Ste20b] and benefits from not being prone to contradictions between the transformations to be combined. For practical application, however, multidirectional transformations suffer from missing modularity, as a single person or team must define the overall relations between all languages. Additionally, it is difficult to think about complex multiary relations between models of multiple languages [Ste20b] and, even worse, the required knowledge to define such a relation may not even exist [Kla18].

Domain experts deal with the tools and corresponding models they require for their tasks in developing a system. Usually, each of them is only concerned with a subset of all tools involved in the development of a system. For example, a performance engineer may be concerned with an instance of the Palladio Component Model (PCM), which represents a component-based architecture description of the system for the Palladio Simulator, to perform

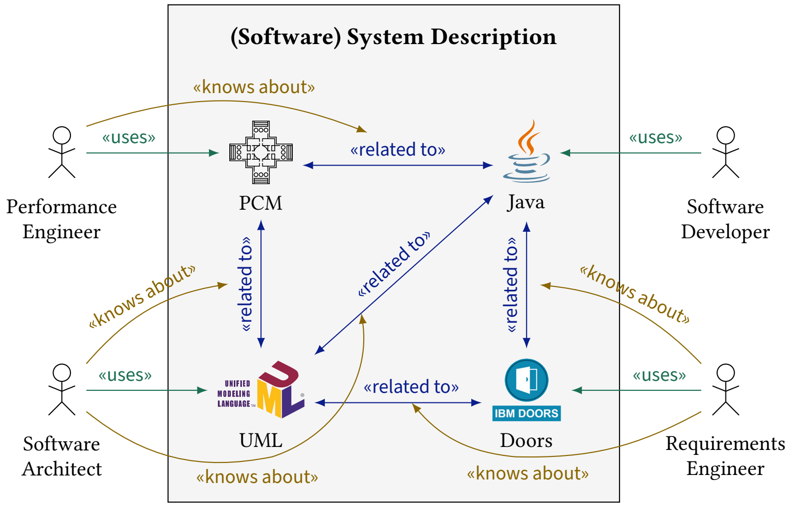


Figure 1.1.: Different tools and roles involved in an exemplary software development process and distributed knowledge about the relations between models of the different tools.

an architecture-based prediction of the system's performance and know how this description is reflected in the system implementation with Java. A software architect may use UML models for the architecture specification and know how they are related to the implementation as well as to the PCM architecture models. Finally, a requirements engineer may use IBM Rational Doors and know how requirements have to be reflected in the architecture specification and implementation to consider the models consistent. These exemplary relations are depicted in Figure 1.1. No matter whether this is how knowledge is actually present at the different roles in a concrete scenario, it emphasizes that knowledge about the relations between languages and their models will usually be distributed across different experts whenever multiple models are involved. In large software systems, a single developer cannot know about all model dependencies [PRV08]. In consequence, a process for specifying consistency by means of transformations has to support a kind of *modularity* to foster independent specification of distributed knowledge.

Furthermore, an automation especially proposes benefits if it is used often. A specification of consistency and its preservation between common languages, such as the UML and a programming language like Java, can be reused across multiple projects. Not every project will, however, use exactly the same tools. Considering the example in Figure 1.1, if the relation between PCM and Java was, at least partly, expressed indirectly across the relations between PCM and UML as well as UML and Java, it would not be possible to reuse that specification in another project that only uses PCM and Java but omits UML. Thus, parts of the consistency specifications, i.e., specifications for subsets of the tools in a project, should be reusable, comparable to Ready to Use Software Products (RUSPs) [I25051]. In consequence, a process for specifying consistency by means of transformations has to support the *independent* specification of *modular* transformations, which can be combined with arbitrary other modular transformations in different contexts.

To support the context induced by the previous considerations, we focus on combinations of transformations, be they bidirectional or multidirectional, instead of having only a single multidirectional transformation. We call such a combination a *transformation network*. To summarize the previous considerations, we need to cover the following context assumptions to the specification of the individual transformations of a network.

Modular: Transformations are defined in a modular way, i.e., each transformation does only specify consistency and its preservation for a subset of the tools used in an actual development project.

Independent: Transformations are defined independently, i.e., each transformation can be developed without considering the contents of the other transformations that it is to be combined with.

1.1.3. Orchestration of Transformation Networks

Combining several modular and independently developed transformations requires their *orchestration*, i.e., the determination of an order in which they are executed to restore consistency. Existing work proposes, for example, to define an execution order explicitly [Pil+08; Van+07] or to derive a kind of topological order [Ste20b]. Such approaches either require manual decisions for the orchestration or restrict the execution to specific topologies, such as directed acyclic graphs or trees. In each case, strong assumptions to the individual transformations or the topology of the networks are made.

It is still unclear how arbitrary modular and independently developed transformations can be combined in a universal way. It is neither known how a

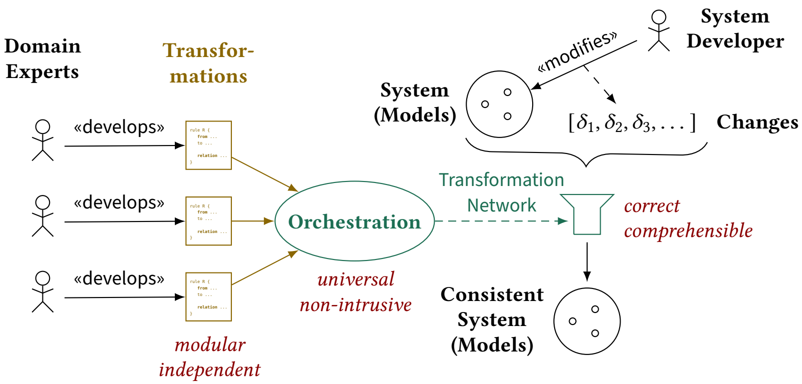


Figure 1.2.: The process of specifying and executing a transformation network. Project-specific development artifacts (transformations) are marked orange and the universal application artifacts (orchestration with a resulting transformation network) are marked green. Concrete systems and changes represent runtime artifacts. The assumed and envisioned properties are denoted in red and italics.

developer can achieve a *correct* transformation network specification, i.e., transformations and an orchestration of them that delivers consistent models when applied, nor how he or she can systematically improve quality properties of the network such as *comprehensibility*.

Under the assumption of a modular and independent specification of the individual transformations, we aim at an approach for executing transformation networks that has the following properties.

Universal: The approach shall be able to process transformation networks of arbitrary topology. In particular, specific topologies cannot be assumed or prescribed if independent development shall be supported.

Non-Intrusive: The approach shall not modify the transformations. When independently developed transformations are combined to a network, they should be treated as black-boxes and there should be no need to adapt them to be used together.

Correct: The approach shall produce correct results. When it applies transformations, it must return consistent models or indicate an error. The identification and definition of an appropriate notion of correctness is part of the contributions of this thesis.

Comprehensible: The approach shall improve comprehensibility. If the transformations are not able to yield models that are actually consistent, it should support the user in finding the reason for that.

The envisioned process with the involved roles, artifacts and required properties is depicted in Figure 1.2. Different domain experts specify transformations, which are combined to a network with an orchestration mechanism that decides in which order transformations have to be executed. If an actual system is developed and a system developer modifies models, the transformations of the network are applied to these models and changes to produce a consistent system description again.

In this thesis, we contribute to support the process of building transformation networks that have the defined properties by providing a formal foundation for transformation networks of arbitrary topology and defining a formal notion of correctness for them. We discuss how correctness of a universal approach to orchestrate and apply the transformations of a network can be achieved by construction or at least by analysis, and which properties the different involved artifacts, such as transformations and their orchestration, have to fulfill for that. The proposed strategy to orchestrate transformations improves comprehensibility in cases in which it is not able to execute transformations in an order such that the resulting models are consistent. Additionally, we classify which kinds of errors can occur when the transformations and their orchestration are not defined correctly. Finally, we analyze how topologies of networks affect the desired properties and propose an approach of defining transformations that resolves trade-offs between the envisioned properties.

In the following, we first discuss the addressed challenges in more detail by considering a specific scenario and generalizing some of the challenges to give a first impression of the issues we have to address. We then derive two general problem statements from the identified challenges. Afterwards, we derive our central, general research goal and define several questions arising from that, which address the problem statements. After more precisely specifying the context and assumptions that we make, we give a detailed overview of our contributions.

1.2. Consistency Specification Challenges

To get an impression of problems arising from the combination of modular transformations, we introduce an exemplary scenario from a software engineering process. We motivate why we expect that multiple executions of the same transformation can be necessary and discuss some of the issues that can occur in that context. Afterwards, we generalize that scenario and derive a more precise problem statement.

We consider an extract of a software engineering scenario, in which three roles using three different tools are involved, according to Figure 1.1. A software developer implements the system with an object-oriented programming language such as Java. An architect manages the object-oriented architecture of the system with the UML. Finally, a performance engineer uses a component-based representation of the architecture with the PCM containing an abstract behavior description at the architecture level to predict the system's performance to evaluate different design options.

The basic entities in PCM models are components, interfaces and data types. Components are units of reuse that define which interfaces they provide or require and contain abstract service specifications for the operations of the interfaces they provide. This allows to assemble a system of components by connecting components through their interfaces, such that every required interface of one component is provided by a defined other component. For the consistency relations between the three languages PCM, UML and Java, which specify when models of those languages are to be considered consistent, we use the ones proposed by Langhammer [Lan17] between PCM and object-oriented design, be it UML or Java, and the intuitive notion of consistency between UML and Java.

Although there are several degrees of freedom when relating UML and Java models, the extracts that we consider follow a simple one-to-one mapping. The relevant relations between elements in PCM and object-oriented design are depicted in Figure 1.3. This involves a one-to-one mapping between interfaces, and the realization of PCM components as classes. Provided interfaces in a PCM model are realized by interface implementations of the class realizing the component. Required interfaces are realized by a field with the type of the interface and constructor parameters that ensure that the required interfaces are set on instantiation of the component.

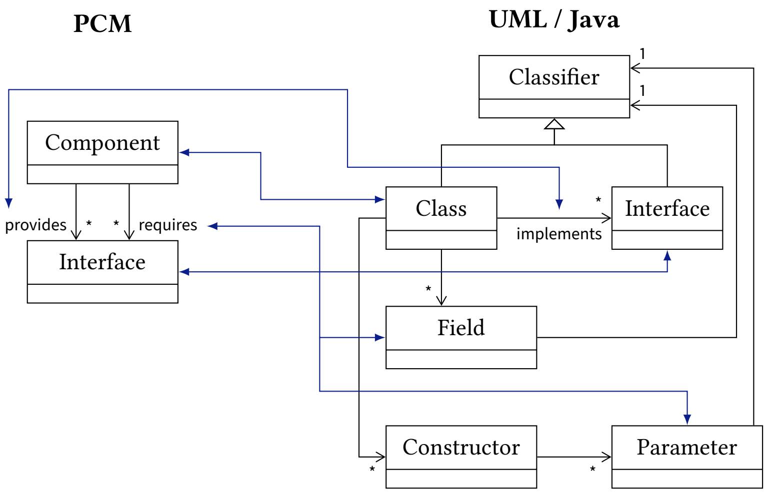


Figure 1.3.: Extract of consistency relations between component-based architectures in PCM and object-oriented design in UML/Java according to Langhammer [Lan17]. (Blue) lines with arrowheads indicate that the connected elements share a consistency relation. Properties, such as names, are omitted.

1.2.1. Correctness of Transformation Networks

One central goal of (software) engineering, and thus the construction of transformation networks as part of that process, is to achieve *correctness*, more precisely *functional correctness* [I25010, p. 11], of the developed artifacts.

Orchestration Challenge

When we consider transformations between PCM and UML, as well as between UML and Java, they can transfer each modification to the other models. For example, adding a PCM component creates a UML class, which in turn creates a class in Java code. Although in many cases each transformation only needs to be executed once, there can be situations that require transformations to be executed repeatedly.

In the process depicted in Figure 1.4, we assume a system description that contains at least one component and class, respectively, and one interface. If

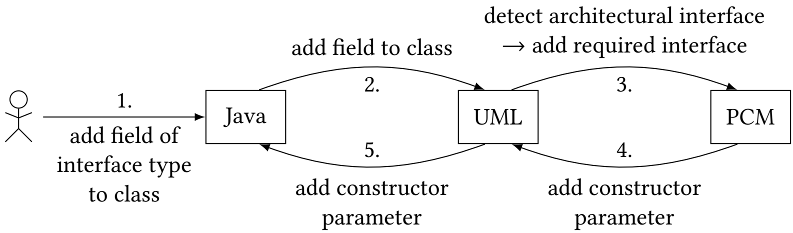


Figure 1.4.: Duplicate transformation execution after adding a field representing a required interface to a Java class. Arrows indicate changes performed by a user or a transformation.

a developer adds a field to the Java class having the type of the interface, the transformation between UML and Java transfers this field to the corresponding UML class. The transformation between UML and PCM detects that the interface is also represented as an architectural interface in the PCM model, thus the field is supposed to represent a required interface in the architectural model. In consequence, the transformation adds a required interface to the PCM component. Since the consistency relations prescribe each required interface to be represented as a constructor parameter, the transformation also adds a constructor parameter to the class in the UML model. This finally requires the transformation between UML and Java to be executed again, because the constructor parameter introduced by the transformation between PCM and UML must also be added to the Java code.

The example demonstrates that it is, in general, necessary to execute each transformation in a network more than once to achieve a consistent state of the models. This is always the case if at least two transformations modify the same model, because the first executed transformation may need to react to changes of second one again, like the transformation between UML and Java needs to react to the one between PCM and UML, because both modify the UML model. We consider the determination how often and in which order transformations must be executed as the *orchestration challenge*.

Synchronization Challenge

Up to now, we have assumed a chain of two transformations, one between PCM and UML and another between UML and Java. There may, however, also be an overlap of information between PCM and Java models that cannot

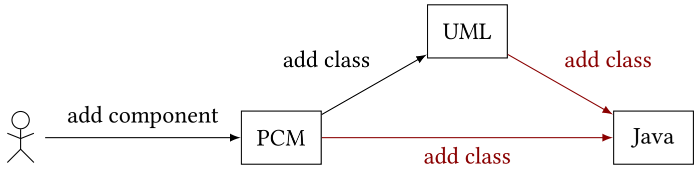


Figure 1.5.: Two transformations propagating the same information to Java code. Arrows indicate changes performed by a user or a transformation.

be represented in the UML, which requires an additional transformation between PCM and Java. This is especially the case for behavioral properties, which cannot be expressed in UML class models, such as the functionality defined by Java methods and the abstract service specifications in the PCM. In consequence, the graph induced by those transformations contains a cycle.

Instead of only having a transformation for that overlapping information of PCM and Java models that cannot be expressed across the UML, the transformation may also contain the relations already expressed across the UML. Reasons for that can be independent development and reusability. Independent development leads to the situation that the developer of the transformation between PCM and Java does not know what the transformations to UML already express. Even if the developer has this information, he or she may want to express it again to foster reusability, i.e., to use the transformation between PCM and Java in projects in which the UML is not used or when the transformation is not supposed to be used in a specific network of transformations, comparable to RUSPs. In consequence, we need to face the situation that multiple transformations propagate the same information, i.e., they contain redundancies.

Figure 1.5 depicts a scenario in which a user creates a PCM component. The transformations, in consequence, create a UML class and, finally, both the transformation between UML and Java as well as the one between PCM and Java specify the creation of an appropriate Java class. These transformations now have to consider that there may be another transformation that has already created that class. Otherwise, there is the risk of creating a duplicate of that class or of overwriting the already created one.

Such a problem can always occur if two sequences of transformations propagate the same information to the same model. How to achieve that transformations deal with such cases constitutes the *synchronization challenge*.

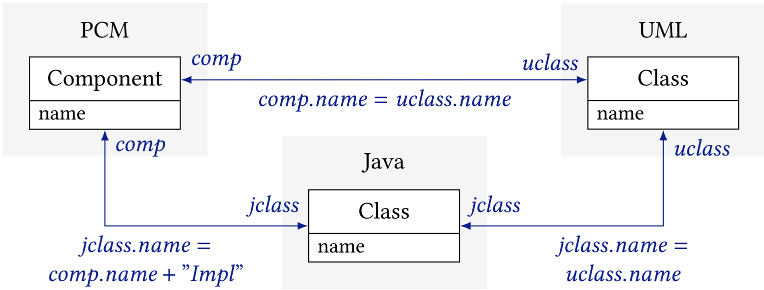


Figure 1.6.: Contradicting consistency relations between components in PCM, classes in UML, and classes in Java. The equations are meant to express that for any existing element another element must exist such that the condition is fulfilled.

Contradiction Challenge

We have seen that it may be necessary to redundantly define the same consistency relations in different transformations. This, however, implicitly assumes that they are true redundancies, i.e., that they equally express the same relations. This, in turn, requires all developers to have the same *notion of consistency* between the different tools.

The example in Figure 1.6 informally depicts exemplary consistency relations between components and classes. They are supposed to express that for each component or class appropriate elements in the other models have to exist that fulfill the given name relation. The constraints for their names can, however, obviously not be fulfilled at the same time. While the class representations are supposed to have the same name, the PCM component is supposed to have the same name as the UML class but the name of the Java class with an “Impl” suffix, as proposed by Langhammer [Lan17].

Such a situation can occur if the developers of different transformations have different notions of consistency. According to the scenario in Figure 1.1, a performance engineer, who knows about the relation between PCM and Java, and a software architect, who knows about the relation between PCM and UML as well as between UML and Java, have different notions of how to represent components in object-oriented design.

If the domain experts encode the defined relations in transformations that preserve them and execute them after any of the elements is added to a model, the transformations will either terminate in an inconsistent state or never terminate at all. Executing the transformation for a finite number of times would always result in an inconsistent state, if not removing the element just added by the user.

In consequence, it is important to avoid or detect situations in which transformations with such contradicting constraints in their consistency relations are combined to a network. We call this the *contradiction challenge*.

Problem Statement

We have discussed three kinds of issues, which can prohibit that a transformation network terminates consistently, and derived according challenges: orchestration, synchronization and contradiction. These challenges only exemplify the relevant correctness issues in transformation networks. In fact, it is even not systematically known which issues can occur. Thus, we derive the following general problem statement.

> **Problem Statement 1**
>
> It is unknown how to correctly combine modular and independently developed transformations to networks to yield consistent models after they were changed.

1.2.2. Quality of Transformation Networks

Like in ordinary (software) engineering, besides the primary goal of producing *correct* artifacts, several quality properties shall or need to be improved. They can range from properties that are relevant for developers, such as reusability and modifiability, to properties relevant for users, such as performance, usability and reliability [I25010, p. 4]. This also applies to transformation networks as artifacts of the (software) engineering process.

Properties and Topologies Challenge

In this thesis, we focus on further properties regarding the development of a transformation network, such as reusability and modifiability, rather

Figure 1.7.: Different transformation network topologies between PCM, UML and Java.

than properties of its usage, such as performance. Reusability is of most importance, because transformations may be used in different contexts within different networks of other transformations.

Consider the two networks sketched in Figure 1.7. The networks contain transformations between PCM and UML as well as between UML and Java. One of them additionally contains a transformation between PCM and Java. They can be considered as representatives of extremes of transformation networks: the graph induced by transformations may on the one end be a tree, and on the other be a complete graph.

It is easy to see that properties are directly affected by the network topology. A complete graph has the benefit of high reusability, because any subset of tools can be used for a development project without loosing consistency. In the example, the tree network is not applicable in development projects not using the UML, because then PCM and Java models cannot be kept consistent. Additionally, a complete graph profits from universality, because arbitrary relations can be expressed, whereas a tree requires that of three languages there is always one that can express the overlap of the two others. If there are overlaps between PCM and Java that cannot be expressed across the UML, like discussed for behavioral specifications, a tree cannot be defined. On the other hand, a tree has the benefit of inherent correctness guarantees. There are no two paths of transformations between the same two languages. Thus, no changes can be propagated across two paths to the same model. This avoids at least two of the three introduced challenges regarding correctness, because neither synchronization problems nor contradictions can occur.

While each kind of topology improves certain properties, it degrades others at the same time. In other words, topologies induce trade-offs between different properties. For example, a tree improves correctness, but degrades reusability in comparison to a complete graph. Deriving how to use this knowledge to mitigate trade-offs and improve different properties at the same time is our *properties and topologies challenge.*

Improvement Challenge

We have seen that topologies directly influence properties of transformation networks. We will see that an appropriate strategy of building networks with a specific topology mitigates trade-offs. Currently, however, there is no known approach that supports building transformation networks of specific topologies improving quality properties. Research approaches have considered approaches and languages for single transformations or specific composition purposes, such as transformations between the same two languages [WVD10; Wag+11], or chains of transformations [Pil+08; Van+07].

To relieve the developer from identifying a topology to improve different properties, a universal approach to define an according topology and an appropriate language that supports its definition should be provided. Investigating such a strategy and design options for an according specification language constitutes our *improvement challenge*.

Problem Statement

We have discussed that topologies affect different correctness and quality properties of transformation network and that they impose trade-offs between them. It is unclear how this insight can be used to systematically improve different properties of transformation networks by building networks of specific topologies. Thus, we derive the following problem statement.

> **Problem Statement 2**
>
> It is unknown how to systematically mitigate trade-off decisions between correctness and quality properties, such as reusability, of transformation networks.

1.2.3. Challenges Overview

We have discussed several issues regarding the construction of transformation networks. Figure 1.8 summarizes the identified problem statements and challenges. We have identified two central problem statements, one regarding the correctness of networks and another regarding the improvement of

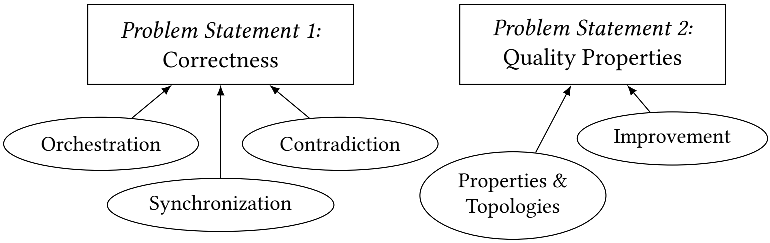


Figure 1.8.: The two identified problem statements and their challenges.

quality properties, each driven by specific challenges. We have discussed orchestration, synchronization and contradiction as central challenges for constructing correct transformation networks. For the improvement of quality properties, we have emphasized that the relation between properties and topologies enables the construction of topologies mitigating trade-offs.

1.3. Research Objective

We have identified specific challenges and generalized problem statements in the construction of transformation networks. In the following, we derive our research goal and the actual research questions that we answer in this thesis in response to the problem statements. Afterwards, we summarize the context and the assumptions of our work. Finally, we give an overview of the contributions to answer the defined research questions.

1.3.1. Research Goal and Questions

The central goal of our research can be summarized as follows.

> **Research Goal**
>
> Define a notion of correctness for networks of modular, independently developed transformations and classify relevant quality properties. Provide approaches to systematically improve correctness and quality properties of transformation networks by construction or by analysis.

The benefits of achieving that goal are twofold. First, researchers and transformation developers both gain systematic knowledge about how to achieve correctness and improve quality properties in transformation networks. Second, transformation developers are provided with concrete techniques and languages that help to achieve correctness and improve other properties either by construction or at least by analysis.

The research goal consists of two parts, one regarding correctness of transformation networks and one regarding the improvement of their quality properties. For each part, we identify fine-grained research questions.

Building Correct Transformation Networks

The first part of our research goal concerns correctness of transformation networks. We want to know what *correctness* means for transformation networks and which aspects of correctness we can achieve for every network. In particular, we want to identify which of them we can achieve by proper construction of each transformation, which we can analyze, and for which we need to deal with potential incorrectness until their execution. We derive the following research questions for the first part of our research goal.

RQ 1 When should networks of independently developed transformations be considered *correct* and how can correctness be achieved?

- **RQ 1.1** What are relevant notions of correctness in transformation networks and how can they be formalized?
- **RQ 1.2** When are the constraints induced by transformations contradictory and how can that be analyzed?
- **RQ 1.3** Which requirements must a transformation fulfill for being used in a network in comparison to using it on its own?
- **RQ 1.4** How can transformations in a network be orchestrated and which properties can such an orchestration strategy fulfill?
- **RQ 1.5** Which errors can occur in transformation networks, how can they be classified regarding their avoidability, and how severe are they?

RQ 1.1 is the fundamental question to precisely define what *correctness* means, beyond our up to now informally given notion. **RQ 1.2**, **RQ 1.3** and **RQ 1.4** directly map to the previously identified challenges regarding orchestration, synchronization and contradiction. Finally, **RQ 1.5** asks for the inverse, i.e., for the case in which errors occur due to incorrectness, to find out how incorrectness manifests during execution and how likely and thus severe the errors are.

Improving Quality Properties of Transformation Networks

The second part of our research goal concerns quality properties of transformation networks. We want to known how we can systematically improve the quality of transformation networks. This includes the identification of properties that are relevant when building transformation networks and how they are affected by different topologies. We use this to systematically derive a proper construction approach achieving a specific topology that resolves trade-offs between quality properties. We derive the following research questions for the second part of our research goal.

RQ 2 How can quality properties of transformation networks be improved systematically?

- **RQ 2.1** What are relevant properties and topologies of transformation networks and how are they related?
- **RQ 2.2** How can topologies of transformation networks improve quality properties of transformation networks?
- **RQ 2.3** How can a specialized language support the specification of a network topology that improves quality properties?

RQ 2.1 maps to the properties and topologies challenge for identifying how topologies affect the fulfillment of properties. **RQ 2.2** and **RQ 2.3** map to the improvement challenge to identify how the proper construction of a topology can improve quality properties and how an appropriate language can support that.

1.3.2. Context and Assumptions

In this thesis, we consider the context of model-driven development processes for software or software-intensive technical systems. Thus, we assume that a system under construction is described by several models containing information about different extracts or properties of the system. We assume that they usually share some overlap of information. Our discussions will focus on software development artifacts. If they follow the same formalism, however, the insights and techniques may be applied to artifacts from arbitrary domains.

We assume that the knowledge about different transformations to be combined to a network is distributed. To foster the development of transformations that can be used as RUSPs, we assume that transformations are developed independently. Thus, transformations may not be adapted to be used within transformation networks.

We do not restrict the kinds of relations between models to keep consistent in any way. We will, however, discuss different types of consistency and their relations to different kinds of processes to preserve consistency in Subsection 3.1.2. In fact, our contributions, although theoretically not restricted to that, will be best applicable to a kind of *structural* dependencies rather than *behavioral* dependencies.

Finally, transformations may not always be able to restore consistency on their own, because necessary information to do so is missing. For example, a developer may add a class in Java code and a transformation has to decide whether that class shall represent a PCM component or not. That problem can either be solved by requiring the class to fulfill certain patterns, like containing “Component” in the class name, or by asking the user about his intent. In cases where information is transformed to a semantically richer model, often further information about how to transform it is needed. Kramer [Kra17, p. 57] provides a classification for different levels of automation, starting from no automation over suggestions and semi-automated repair to fully automated repair. In this thesis, we assume that consistency is preserved in a fully automated way, thus excluding the semi-automatic case. We will finally discuss how our finding generalize to cases in which user decisions need to be included.

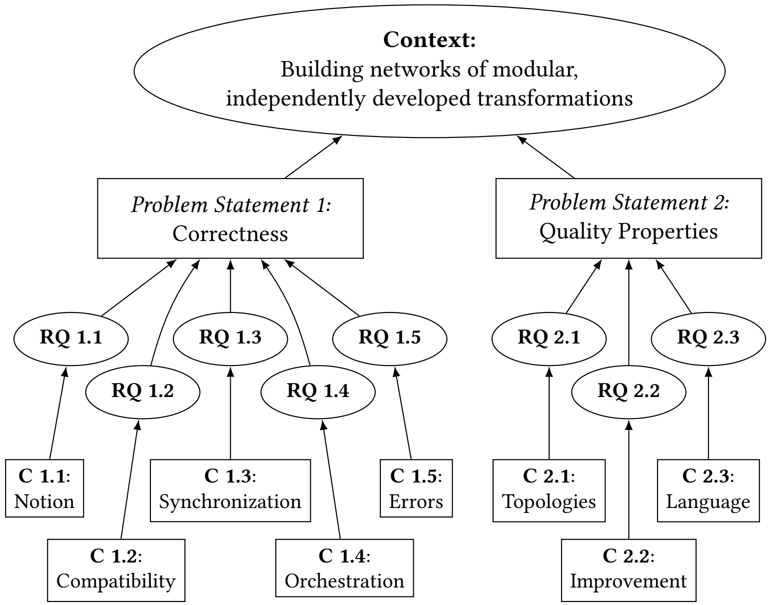


Figure 1.9.: Relations between context, problem statements, research questions, and contributions.

1.3.3. Contributions

The contributions that we make in this thesis are structured along the same dimensions as the problems and the research questions, namely correctness and quality properties of transformation networks. The contributions directly map to the research questions. Figure 1.9 gives an overview of the relations between the context of our work, the problem statements, the research questions and the contributions that we make.

We make the following contributions regarding transformation network correctness.

C 1.1 *(Notion)*: We discuss different notions of correctness for transformation networks and precisely define the one relevant for our context. We derive that compatibility, synchronization and orchestration constitute relevant correctness notions.

C 1.2 *(Compatibility)*: We precisely define a notion of compatibility to express when transformations contain contradictory constraints. We propose an approach that validates compatibility of transformations and prove its correctness.

C 1.3 *(Synchronization)*: We discuss how synchronization can be achieved for transformations defined with existing transformation languages. We prove that transformations fulfilling a specific property can be applied in transformation networks. We provide an algorithm to execute the transformations in that case and propose a strategy to fulfill the required property by construction.

C 1.4 *(Orchestration)*: We prove that transformations can, in general, neither be executed only once nor an arbitrary number of times in a fixed-point iteration without the risk of non-termination. We prove that finding an execution order of the transformations that yields consistent models is an undecidable problem and discuss why we cannot make practicable restrictions to the transformations to achieve its decidability. We propose an algorithm for orchestration that executes the transformations according to a well-defined strategy that helps to find the cause whenever it does not return consistent models.

C 1.5 *(Errors)*: We systematically derive which errors can occur when correctness of a transformation network is not given. We empirically evaluate the probability of the different errors to occur to classify their severity and thus the importance of avoiding them.

We make the following contributions regarding the improvement of quality properties of transformation networks.

C 2.1 *(Topologies)*: We discuss how different quality properties of transformation networks are affected by the network topology. We derive that trade-off decisions have to be made regarding the improvement of different properties.

C 2.2 *(Improvement)*: We propose a strategy for building a specific network topology, which makes the consistency relations explicit in terms of auxiliary models rather than transformations. We show that this approach systematically improves different quality properties and mitigates necessary trade-off decisions.

C 2.3 *(Language)*: We propose a specialized language for the definition of a transformation network according to the strategy of **C 2.2**. We discuss different design options for the language and its operationalization.

1.3.4. Expected Benefits

The contributions that we make in this thesis provide several benefits for researchers, developers of transformations and transformation networks, as well as transformation (network) users. All of them profit from systematic knowledge about what *correctness* means for transformation networks, how correctness is affected and can be guaranteed, and about relevant *quality properties* in transformation networks as well as how they can be improved. The contributions, however, have an intended focus on supporting transformation and transformation network developers.

Researchers can base on our definitions for correctness of transformation networks and can thus precisely contribute to particular parts of the correctness notions, such as approaches to achieve correctness with explicitly knowing how and which kinds of potential errors of transformation networks are affected by that. Additionally, they can base further research on the insights about property trade-offs induced by different network topologies.

The developers of actual transformation networks consist of developers of the individual transformations and the ones combining them to a network. The development of individual transformations is supported by the provision of systematic approaches to build transformations that can be used within networks, especially in terms of supporting synchronization. Transformation network developers benefit from the knowledge that they have to deal with undecidability of orchestration. They also benefit from approaches to validate transformations they want to combine regarding compatibility, an actual and practical orchestration strategy to execute transformations, and an approach to build networks that mitigate property trade-offs.

Finally, the users of a transformation network, i.e., those who develop a system using a transformation network to preserve consistency of its artifacts, benefit from the ability to use networks for which correctness was systematically achieved. They also profit from an orchestration strategy that supports them in finding and understanding the reasons why the network may not be able to process certain changes to preserve consistency.

1.4. Thesis Outline

The remainder of this thesis is structured as follows. We briefly introduce fundamental terms, concepts and ideas in Chapter 2 and define own terminology and notions on which we rely in Chapter 3. Part II and Part III then structure the contributions along the topics of transformation network correctness and the improvement of quality properties. Within Part II, Chapter 4 first derives a reasonable notion of correctness for transformation networks, from which the three topics of proving compatibility (Chapter 5), achieving synchronization (Chapter 6), and orchestrating transformations (Chapter 7) are derived. We discuss potential errors if correctness is not given in Chapter 8, before we evaluate approaches presented in these chapters in Chapter 9. Within Part III, Chapter 10 first discusses quality properties of transformations networks and how they are affected by the network topology. Chapter 11 derives an approach for mitigating trade-offs between these quality properties, which is supported by a language proposed in Chapter 12 and which we evaluate in Chapter 13. Each of the Chapters 4–7 and 10–12 addresses one of the identified research questions and provides one of the depicted contributions, whose central insight is summarized at the end of each chapter. After relating our work to different fields of research in Chapter 14, we conclude with a summary of future work in Chapter 15.

Beyond sequential reading, there are multiple other modes for readers particularly interested in specific topics. We suggest to always read Chapter 3 and, with less importance but for better understanding, also Chapter 4, as they define essential notions and notations. Readers especially interested in topics related to correctness of transformation networks can proceed with any of the Chapters 5–8, which are almost independent, and follow back references where necessary. Readers particularly interested in the improvement of quality properties of transformation networks can skip Chapters 5–9 and proceed with Chapters 10–13, which should be read sequentially. These chapters also refer to the insights from Chapters 5–9 but will also be comprehensible without reading them or by following back references where necessary. Readers who only want to obtain a better general overview of the contributions of this thesis also have the option to read the insights at the ends of the chapters and the conclusions in Chapter 15, potentially complemented by the fundamental notions in Chapter 3 and Chapter 4.

2. Foundations and Notation

In this chapter, we introduce fundamental concepts and notations that we use throughout this thesis. We consider modeling in terms of a notion of models and methods in which they are used, and depict important formalisms and frameworks for modeling. We introduce the idea of multi-view modeling and, in particular, the Vitruvius approach, which we employ for evaluations. Finally, we discuss model transformations and languages to describe them. After introducing the case studies used for our evaluations and, partly, for explanations of our contributions, we depict the mathematical notations that we use in this thesis.

2.1. Modeling

This thesis researches the employment of transformations to keep multiple models, which are used to describe a single software system, consistent. Therefore, we first introduce a notion of models and how to use them.

2.1.1. Models and Model Theory

Models are a ubiquitous concept, which is used throughout many technical and non-technical domains. The term *model* is used differently in various contexts from informal depictions to mathematical formalizations [Sta73]. In his work on general model theory, Stachowiak characterizes models by three criteria: representation, abstraction and pragmatics [Sta73, p. 131–133].

Representation: The *representation* characteristic requires a model to be a mapping or representation of some *original*. An original not necessarily needs to be a natural, existing entity, but can also be any kind of concept, which can, again, be a model [Sta73, p. 131]. We always consider models that are representations of a software-intensive system under construction. This characteristic requires a model to contain no information that is not related to the system, such that, if the system and the model could be represented by a set of explicit properties, we would be able to define a mapping, or, more precisely, a homomorphism between them.

Abstraction: The *abstraction* characteristic requires a model to only represent a subset of the properties of its original. Properties are limited to those that seem relevant to the creator of the model [Sta73, p. 132]. This abstraction should be driven by the pragmatics of the model, defined as the third characteristic. For example, an architecture model of a software system may only represent properties relevant for some information need at the architectural level, which could, for example, abstract from behavior or implementation details.

Pragmatics: The *pragmatic* characteristic requires a model to be designed for a specific purpose, such that it can only be related to its original for specific users, for specific points in time, and for specific operations [Sta73, pp. 132]. Models of software systems can, for example, have the purpose of depicting or editing the system structure or its behavior, or of performing some analyses or simulations for specific properties of the system. The pragmatics influences the abstraction, as a specific purpose implies a certain information need to be provided by a proper abstraction.

While this is a rather general notion of a model, it also fits to the one relevant for software engineering, as depicted in our examples. One appropriate definition for models in the domain of software design has been given by Rumbaugh et al.: "A model is an abstraction of something for the purpose of understanding it before building it" [RB05, p. 15]. This fits well to the notion of making predictions about the systems upfront, such as the already mentioned Palladio Simulator making performance predictions about a software system based on an architectural model of it. Models may, however, not only be used to understand the system but also to build it, especially when considering code as a model as well.

2.1.2. Metamodels and Languages

To automatically or semi-automatically process models, such as compiling source code, these models need to follow some specification, which can be considered a model that defines how a valid model for a specific purpose looks like. Such a model of a model is often denoted as a *metamodel*. Models and their metamodels induce an *instance-of* relationship, such that a metamodel can be considered the type of a model, and a model is considered an instance of a metamodel. This conforms to the notion of type and instance level known from programming. The grammar of a programming language, such as the Java language specification [Gos+18], can be considered a metamodel for programs of that language.

In a simple notion, a metamodel can be considered as a set of models, such that a model is an instance of that metamodel if it is contained in that set. This is sometimes also referred to as *model sets* [Ste20b]. Usually, metamodels will be described with some formalism, which we discuss in more detail in Section 2.2. Such a formalism defines the elements of which a metamodel consists and how these elements are instantiated in the models, along with some constraints that a model has to fulfill to be considered a valid instance.

Models, especially in software engineering, are often understood as structures of objects and relations between them, which can be depicted in UML class diagrams. Although this notion fits well to how we consider models and how we later define them more precisely, the elements of models must also have a meaning, i.e., a semantics [HR04], in the specific context they are used for. This semantics is given by the pragmatics characteristic of Stachowiak's classification. For models in software engineering, this semantics is usually defined by *modeling languages* and tools defined for that modeling language, in which these models are defined and used. These languages and tools, for example, transform models into another representation, i.e., into another model, for which the semantics is known For code, execution semantics can be given by its compilation to machine code for some, potentially virtual, machine whose execution semantics is known. This is known as *transformational semantics* [Pep79].

A modeling language consists of a specification of abstract and concrete syntax, as well as its static and execution semantics [Völ+13, p. 26].

Abstract Syntax: Defines a data structure containing the relevant information about a system or program, usually in terms of a tree or graph.

Concrete Syntax: The notation in which a user can express models, such as a textual or graphical representation.

Static Semantics: A set of constraints that a model has to fulfill in addition to conforming to the syntax, such as a type system.

Execution Semantics: The semantics of a program or model when it is executed, which can also be given by a transformation to another model.

Völter et al. [Völ+13] use the term *Domain-Specific Language (DSL)* instead of modeling language. DSLs are supposed to increase productivity and conciseness for specifying models in a specific domain [Völ+13, p. 30] in contrast to using a *General-Purpose Language (GPL)*. A language is, however, not either domain-specific or general-purpose, but domain specificity of a language is a gradual notion [Völ+13, p. 30]. DSLs being designed for a specific domain are usually assumed to have restricted expressiveness [Fow10, Chap. 2]. The term *domain* can have different meanings. Völter et al. distinguish between *technical* and *application domain* DSLs, although emphasizing that there is no clear border between them [Völ+13, p. 26]. In the context of this work, we can distinguish DSLs used by software developers and DSLs used by developers of software development tools. DSLs for software developers can again be separated into rather generic DSLs, such as the UML for general software design and the PCM for general performance prediction, and rather application specific DSLs, such as MATLAB/Simulink [Simu] or AUTOSAR [Sch15] in automotive software development. DSLs for software development tool developers cover languages to specify transformations and editors to be used for developing software and keeping software models consistent. In this work, especially transformation languages used by developers of transformation networks to support software development are relevant, whereas languages of software developers are used to define the models that transformations have to keep consistent. Since we are not concerned with domain specificity of a language, we only use the general term *modeling language*.

Metamodels are often considered as the abstract syntax of models [Völ+13, p. 27], whose semantics is defined by the modeling language it is used in. In this thesis, we use a notion of models and metamodels that we define more precisely in Section 3.3, which does also not reflect the semantics of the models explicitly. Some semantics of models is, however, represented implicitly by the transformations preserving consistency.

2.1.3. Model-Driven Software Development

Model-Driven Software Development (MDSD) [Sta+06] is a general term for the idea of increasing abstraction in software development by using models instead of or in addition to program code [AK03]. It also appears as *model-driven software engineering* or simply *model-driven development* [AK03]. It has been considered the natural continuation of increasing abstraction, like achieved with more powerful compilers and higher abstraction in programming languages before, by automating repetitive tasks such as support for persistence or interoperability [AK03]. This includes that models are central development artifacts, from which even code can be derived, rather than documentation artifacts.

The Model-Driven Architecture (MDA) [MDA] proposed a standard for an MDSD process, in which abstract, platform-independent, and thus highly reusable and portable models are used to generate code for different platforms. Stahl et al. propose a more sophisticated process, in which repetitive and generic code is separated from individual code, such that repetitive code can be generated and extended by individual code [Sta+06, Fig. 2.1].

We consider MDSD as an even more generic process using any models to describe a system under construction, which do not only serve documentation purposes but which all contain some information that is not represented in the other models, while still sharing common information that, as a central part of the motivation of this thesis, needs to be kept consistent. Thus, we do especially not split the code into repetitive and individual code, as we also treat code as a model that can be changed like the other models. In this thesis, for example, we employ a metamodel for Java code [Hei+09a]. This follows the notion of Bézivin that "everything is a model" [Béz05].

2.2. Modeling Formalisms and Frameworks

Models are instances of metamodels, as discussed in Subsection 2.1.2, which usually rely on some formalism that defines which elements metamodels can contain and how they are instantiated in models. Such a *modeling formalism* can, again, be defined as a model of the metamodel, which is then called a *meta-metamodel*. We call each of the instantiation levels of models and their metamodels a *meta-level*. While there can, in general, be an arbitrary number of meta-levels, for practical reasons there has to be a topmost model in this hierarchy that is *self-describing*.

We depict two modeling formalisms, the *Meta Object Facility* and *Ecore*, which are commonly used in software engineering and which we use in this thesis. Using a common modeling formalism for all models and metamodels enables the application of common tooling to them, which, in our case, especially concerns transformations. A *modeling framework* provides the infrastructure for such common tooling of a modeling formalism. Ecore belongs to the *Eclipse Modeling Framework*, which defines an infrastructure for tooling on models and metamodels defined with Ecore, based on a code representation of models with a well-defined Application Programming Interface (API).

2.2.1. Meta-Object Facility

The Meta Object Facility (MOF) [MOF] is a standardized modeling formalism, i.e., it defines a self-describing meta-metamodel, which is also called MOF. It contains the Essential Meta Object Facility (EMOF), which is a subset of the MOF derived from class models in the UML [UML]. The MOF standard does not prescribe a specific number of meta-levels [MOF, Sec. 7.3]. We do, however, usually assume four meta-levels, as defined by the UML standard [UML] and as used for Ecore as a realization of the EMOF. These meta-levels, denoted M3–M0, comprise the meta-metamodel at M3, metamodels at M2, models at M1 and, finally, instances of models at M0.

The modeling formalism used in this work will be even more generic than the one proposed by the EMOF, but can be considered a generalization of it. For a less abstract understanding, the reader may, thus, have the EMOF in mind and apply the discussions to it. In addition, we denote examples and perform our evaluations with EMOF-compliant models and metamodels. To support this, we depict an important subset of the EMOF meta-metamodel as a UML class diagram in Figure 2.1. Comparable to class models in the UML, the EMOF defines classes consisting of properties, which have multiplicities and a type. The type of a property can, again, be a class but also an enumeration or a primitive type. Each property has multiplicities that define an upper and lower bound for the number of elements to refer to. In addition, a property defines whether it is composite, denoting that the elements referenced in an instance are to be considered contained in an instance of the class containing that property. This simple structure of classes and relations between

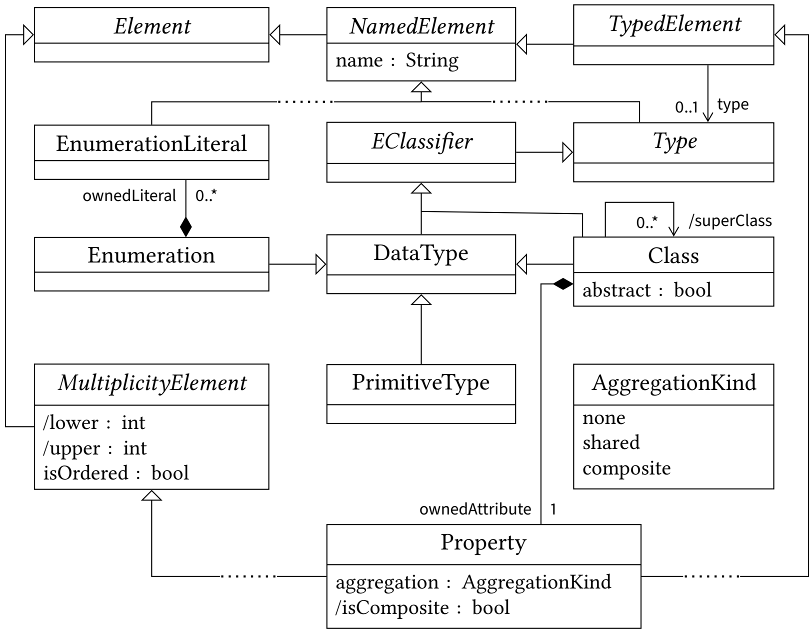


Figure 2.1.: Simplified class diagram with central metaclasses of the EMOF modeling formalism [MOF, p. 27]. Dotted lines denote indirect inheritance. Adapted from [Kra17, Fig. 2.2].

them leads to models and metamodels of the EMOF that are mathematically equivalent to attributed, typed graphs with inheritance [Kra17, Sec. 2.1.3.1], such that they are widely applicable. Even common engineering tools such as AUTOSAR [Sch15] and SysML [SysML] use MOF-compliant models.

We usually denote the types of model elements as *metaclasses* rather than classes, especially to avoid confusion with classes of UML class models. UML class models, defined at M1, contain classes, which are instances of a `Class` metaclass in the UML metamodel at M2, which, in turn, is an instances of the `Class` metaclass of the EMOF.

Since the restriction to type and instance level of models and metamodels at M1 and M2 increases accidental complexity in models [AK08], other formalisms such as *multi-level modeling* support an arbitrary number of metalevels and precisely separate ontological and linguistic modeling [AK03]. This accidental complexity is complementary to the one introduced by replicating information across different models, which we aim to manage with consistency preservation mechanisms, as it concerns the accidental complexity within the single models due to restricted modeling capabilities. Although multi-level modeling gained more attention in the last years [AGK14], common modeling frameworks such as the Eclipse Modeling Framework are still restricted to linguistic instantiation relations between metamodels, models and their instances, which is why we stick to such formalisms.

2.2.2. Ecore and EMF

The Eclipse Modeling Framework (EMF) [Ste+09] is a modeling framework for Eclipse, which is a plugin-based, extensible Integrated Development Environment (IDE). It uses the meta-metamodel *Ecore* and provides an infrastructure for defining tools on models based on Ecore. This bases on a code generator for metamodels [Ste+09, pp. 237], which does not only relieve the developer from manually specifying a metamodel as a data structure in code manually but also ensures that the code provides a well-defined API, on which tools can rely, as it is provided by any metamodel developed with the EMF. This enables the definition of tools, such as editor frameworks that only require configuration files for providing a sophisticated graphical editor for a model, or transformation languages that enable the definition of transformations between arbitrary Ecore metamodels. Regarding meta-levels, the EMF provides the Ecore meta-metamodel for which it allows the definition of metamodels and which can then be instantiated in models.

Ecore can be considered a reference implementation of the EMOF standard. Thus, Ecore and EMOF share most concepts, but, apart from minor structural and naming changes, Ecore provides some refinements compared to EMOF. We depict the relevant subset of the Ecore meta-metamodel as a UML class diagram in Figure 2.2. The most notable difference is that Ecore separates EMOF properties, called *features*, into attributes and references, of which attributes refer to enumerations and primitive types, whereas references refer to other classes. In contrast to properties being composite in EMOF, references in Ecore have an explicit `containment` attribute.

In this thesis, whenever referring to an existing modeling formalism rather than the more general one we propose, we use the terminology of Ecore. The distinction of attributes and references in Ecore eases understanding, as it conforms to the notion of class properties and associations in the UML.

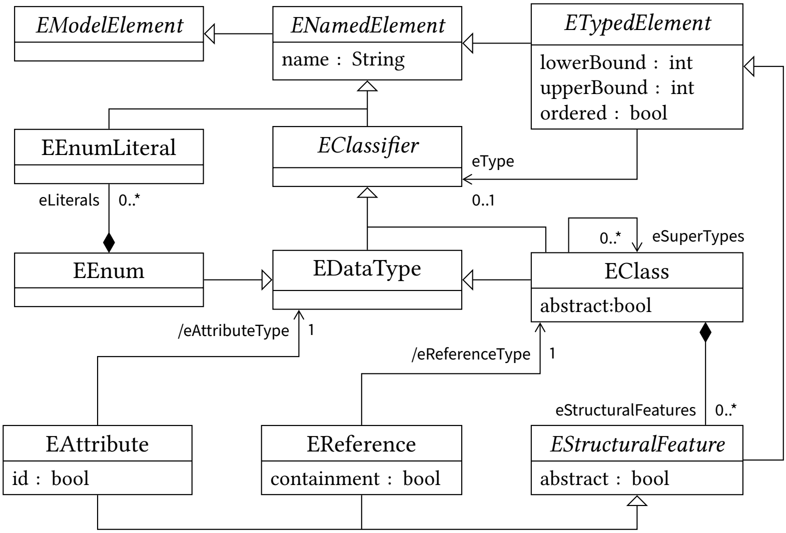


Figure 2.2.: Simplified class diagram with central metaclasses of the Ecore modeling formalism [Ste+09, p. 107]. Adapted from [Kra17, Fig. 2.3].

For the EMF, many tools such as editor frameworks, transformation languages and language workbenches have been developed. We explicitly discuss transformation languages in Section 2.4. Language workbenches allow the specification of modeling languages. One such workbench is Xtext [Bet16] for defining languages with a textual concrete syntax. It allows to define the language grammar, from which it derives the metamodel in terms of an abstract syntax, as well as parsers and editors. A compiler or generator can be defined to transform models in that language into another representation, such as executable code, giving them their semantics. Such a language workbench can be used for languages to define domain models but also for languages used as tooling in MDSD processes. We use Xtext for the implementation of the prototype of a transformation language that we propose in this thesis, and it has also been used to develop the Reactions language, which is a transformation language that we introduce in Subsection 2.4.3 and that we reuse for the evaluation in this thesis.

As already introduced in Subsection 2.1.3, code can also be considered a model. For the representation of Java code as an Ecore model, JaMoPP has been proposed [Hei+09a; Hei+09b]. It defines an Ecore metamodel for the Java language and also provides parsing and printing capabilities for treating Java source code files as Ecore models.

2.3. Multi-View Modeling

Multi-view modeling covers the general topic of describing a system by means of multiple views or, in general, multiple models [RST19]. A key challenge in multi-view modeling is consistency [RST19], as we have motivated in Chapter 1. Preserving consistency between multiple views is referred to as *model repair* [MJC17], *consistency restoration* [Ste10; Kra17] or *model synchronization* [Dis+16b], with slightly different meanings. We usually refer to this as *consistency preservation*.

The term *architecture view* has been defined in the context of system architecture as an expression of the architecture regarding specific concerns in an ISO standard [I42010, p. 2]. We generalize this to *views* as representations of system extracts or properties regarding specific concerns. Approaches for constructing views can be separated into *synthetic* and *projective* ones [I42010, p. 22]. A synthetic approach composes a system description of views, such that each of them represents some information not contained in the others. Projective approaches derive the information in a view completely from an underlying repository, thus views are only projections from that repository. In projective approaches, the underlying repository can, again, be seen as a model, such that views in projective approaches are projections of that model [Kla+21, Fig. 5]. This underlying model is also called a Single Underlying Model (SUM) [ASB10, p. 210], which conforms to a metamodel, the Single Underlying Metamodel (SUM metamodel) [Kla+21, Def. 2].

In a projective approach, the problem of preserving consistency between the views, as it is necessary in a synthetic approach, is transferred to ensuring consistency within the SUM, from which the views are projected. Consistency of this SUM can be achieved in different ways [Mei+19; Mei+20], especially depending on whether a SUM is *essential* or *pragmatic* [ATM15]. An essential SUM is free of any redundancies or implicit dependencies, such that every instance of its SUM metamodel is inherently consistent, whereas a pragmatic SUM can allow arbitrary redundancies and dependencies, which then have to be kept consistent by explicit mechanisms for consistency preservation, such as transformations. While the former approach is followed by the Orthographic Software Modeling approach, the latter is used in the Vitruvius approach, which we depict in more detail in the following.

2.3.1. Orthographic Software Modeling

Orthographic Software Modeling (OSM) is an approach to multi-view modeling based on the idea of an essential SUM and proposed by Atkinson et al. [ASB10]. It assumes a SUM, which is, in the best case, free of any redundancies and dependencies and thus inherently consistent. The approach focuses on the creation and management of projective views from this SUM [ASB10, p. 211]. It proposes to structure these views along their properties, which span different dimensions and induce a cube in which each cell potentially represents a view, at least if the associated combination of property values makes sense [ASB10, p. 212]. Dimensions can be static, such as the abstraction level or the notation, or dynamic, such as the elements to display. For example, one might select a graphical view at the architecture level for a specific component, or a textual view at the implementation level for a specific class. These views are created dynamically and on-demand from the SUM [ASB10, p. 211], and views are assumed to be the only possibility to modify information in the SUM. Views, like models, base on a metamodel that defines when views are valid, which is called a *view type* [Gol11, p. 133].

Consistency in this approach is achieved by proper construction of a SUM metamodel, which ensures that instances are always consistent. It requires transformations between the views and the SUM to first generate a view and later propagate changes in the view back to the SUM. The approach does, however, not inherently solve the problem of concurrent modifications to different views to be merged.

2.3.2. The Vitruvius Approach

The Vitruvius approach [Kla+21] bases on the OSM idea of having a SUM from which projective views are derived through which the information in the SUM can be modified. Instead of essential SUMs, is uses pragmatic SUMs, which can contain redundancies and dependencies that are kept consistent. The SUM internally consists of models, which are kept consistent by model

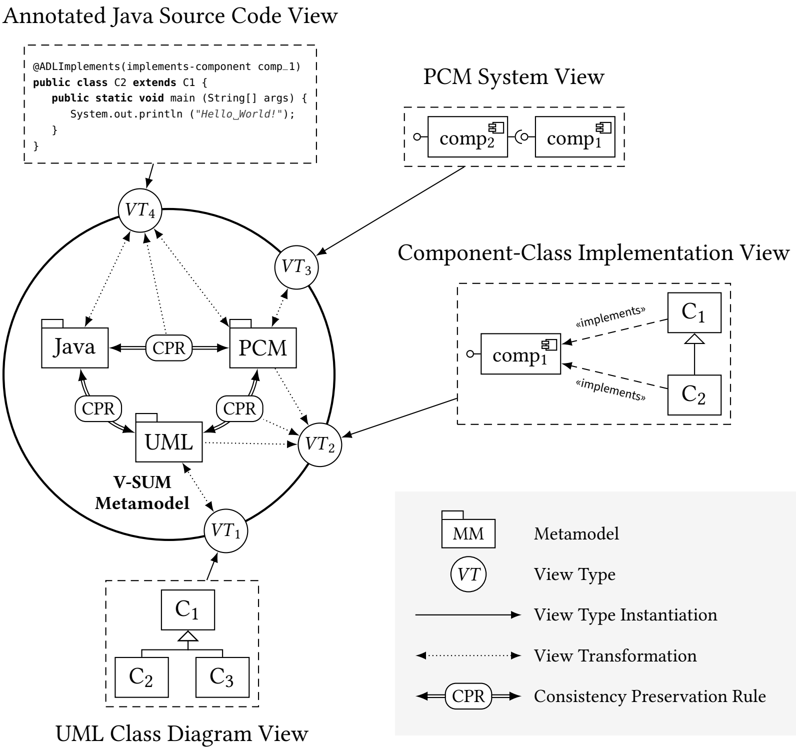


Figure 2.3.: Exemplarily V-SUM metamodel consisting of three metamodels for component-based development and exemplarily views derived from them. Adapted from [Lan17, Fig. 4.4].

transformations, called *consistency preservation rules*, and is denoted as a Virtual Single Underlying Model (V-SUM) [Kla+21, Def. 9]. The metamodel of a V-SUM is denoted as a Virtual Single Underlying Metamodel (V-SUM metamodel) [Kla+21, Def. 10]. It is motivated by the insight that constructing an essential, redundancy-free SUM is hard to achieve [Mei+20]. In addition, to achieve compatibility with existing tools and their modeling languages it may be easier to combine their metamodels with a synthetic approach, because then the view used by each tool is only a projection given by an isomorphism to one of the models within the V-SUM [Kla+21]. Still, in contrast to a purely synthetic approach, it allows to define further projective views derived from the information of the models in the V-SUM.

Figure 2.3 depicts an exemplary V-SUM metamodel for component-based development, using Java for the source code representation, the UML for depicting object-oriented design, and the PCM for representing the architecture of the system and potentially performing quality predictions. These three metamodels form the V-SUM metamodel, whose instances can be accessed via views that can be instantiated from four exemplary view types. VT_1 and VT_3 depict existing view types, already used as visualizations of UML and PCM models, whereas VT_2 and VT_4 represent view types projected from multiple models within a V-SUM and potentially further information defined by the consistency preservation rules. We will also introduce consistency between these metamodels as a case study used for explanations and evaluations of this thesis in Section 2.5.

Within a V-SUM, multiple models need to be kept consistent, which is one application area for the contributions of this thesis. Vitruvius serves both as a motivation for the contributions of this thesis, but its implementation in the Vitruvius framework [GitVit] and especially its languages for consistency preservation also serve as a basis for our prototypical implementation and validation purposes. For Vitruvius, we have provided a simple but sufficient formalism defining consistency [Kla+21]. The formalism in this thesis bases on it but will be more detailed and fine-grained. Additionally, we will see that the abstraction provided by a layer of projective views onto the models that are kept consistent in a V-SUM provides additional benefits in our approach for improving quality properties explained in Chapter 11 rather than using it standalone.

2.4. Model Transformations

In addition to models and formalisms to define them, model transformations are another core element of MDSD processes. They are sometimes considered the “heart and soul” [SK03] of MDSD. Model transformations, which we also simply denote as *transformations* throughout this thesis, generate one model or even code from another model.

According to Kleppe et al. [KWB03], a transformation defines how to generate a *target model* from a *source model* by a *transformation definition*. A transformation definition consists of *transformation rules*, which in turn define how one or more constructs of the source language or metamodel

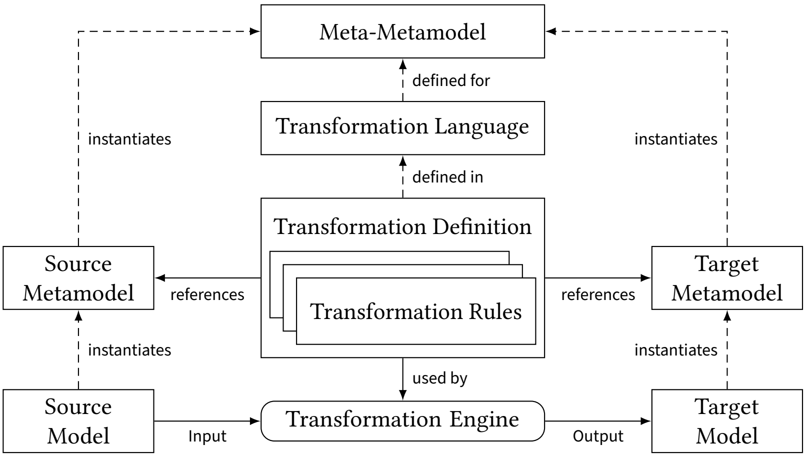


Figure 2.4.: Artifacts of a transformation and transformation language, as well as their relations. Adapted from [KWB03, Fig. 9-5].

are transformed into constructs of the target language or metamodel. For example, a transformation definition may define how to transform a PCM model into a UML model, which consists of transformation rules, of which one could define how a component is transformed into a class. Transformation definitions and their rules need to fulfill some format expected by the transformation engine, which is responsible for applying the transformation rules. A transformation engine is often supported by a *transformation definition language* [KWB03, Sec. 9.2], or short *transformation language*, in which transformation rules can be defined and from which appropriate artifacts for the transformation engine are generated. These terms and their relations are depicted in Figure 2.4, restricted to transformations between two metamodels. While the notions of Kleppe et al. [KWB03] are specific to the MDA, thus deriving more specific from abstract artifacts, we have generalized them to transformations between arbitrary languages.

2.4.1. Properties and Bidirectional Transformations

Transformations do not only support the simple case of taking one model and generating another, known as a *batch transformation*, but there are several degrees of freedom how information from one or more models can be transferred into one or more other models by a transformation. This also includes the incremental update of multiple models after concurrent changes for restoring consistency. Czarnecki et al. [CH06] provides a classification of transformations regarding a variety of features. For our use case, in particular *directionality* and *incrementality* [CH06, p. 14] are important.

Directionality: Regarding directionality [CH06, Fig. 19], transformations can be separated into unidirectional and multidirectional ones, of which the latter includes the well-researched bidirectional transformations. It describes whether a transformation can be applied in only one or multiple directions. For consistency preservation purposes, transformations usually need to be executed in multiple directions, depending on which of the models was changed and requires others to be updated.

Incrementality: Incrementality [CH06, Fig. 19] concerns *source incrementality* and *target incrementality*. We use the term incrementality specifically for target incrementality, which describes the ability of a transformation to update an existing target model after changes to the source model. This is essential for consistency preservation, because otherwise changes and additions made to the target models would become overwritten. For example, if Java code is generated from a UML class model, then a change to the UML model should incrementally update the Java code instead of generating it anew to avoid that additions to the code, such as method implementations, get lost. Target incrementality is also referred to as *change propagation*. Source incrementality is about re-executing only transformation rules for changed parts of the source model. Instead of using this term, we later introduce the notion of *delta-based* transformations, which operate on the actual source model changes.

Another feature of transformations that is relevant for some of our contributions are *intermediate structures* [CH06, p. 10]. These structures concern additional models, which are often temporarily used for transformation execution and especially include traceability models. Traceability models represent which elements of the source and target model are related to each other by a transformation rule and, to enable incremental execution, are usually persisted in contrast to other structures [CH06, p. 10]. These models define which model elements have some kind of dependency and thus serve as information about or even a witness for consistency, which can even be used to define transformations [DGC17].

Among all options from unidirectional to multidirectional transformations, *bidirectional* transformations are the ones that are of most interest for consistency preservation and thus well-researched. These transformations relate only two metamodels, which makes them less complex than other multidirectional transformations. They define how consistency is restored in both directions [Ste10], which is important for consistency preservation if instances of both metamodels may be modified. Bidirectional transformations consist of a *relation* defining when two models are consistent and two *consistency restorers*, one for each direction. A consistency restorer is a function that accepts two potentially inconsistent models and returns an updated instance of one of the models, depending on the direction. There are also derivations that expect two consistent models and explicit changes to one or both of them, as we discuss in detail when introducing our formalism.

Important properties of such transformations are *correctness* and *hippocraticness*, as defined by Stevens [Ste10]. A bidirectional transformation is correct if the resulting models are consistent, i.e., if the updated instance of one model and the input instance of the other model are in the relation of the transformation. A transformation is hippocratic if the consistency restorers do not alter the input models if they are already consistent. Thus, consistent models are induced by the image of a hippocratic transformation. We recapture the notion of bidirectional transformations and the depicted properties to define them more precisely for the formalism that we introduce in Chapter 4.

2.4.2. Transformation Languages

Although transformations can be implemented manually by directly modifying the models [CH06, p. 16], they usually rely on some engine that accepts rules implemented for a specific API and automate tasks such as scheduling or orchestrating the execution of transformation rules. Such an engine can be defined on its own but is often provided together with a transformation language, which uses a specific syntax for defining transformation rules and from which implementations of these rules for the specific API of the engine are generated. Transformation languages can be considered DSLs (see Subsection 2.1.2). We have already depicted these artifacts in Figure 2.4.

Among various degrees of freedom to define a transformation language, just like a transformation itself, we especially distinguish between rather *imperative* and *declarative* transformation languages. We say "rather" because being declarative is actually a gradual and not a total notion. Imperative languages allow to define how consistency is restored whenever changes are performed, whereas a declarative language allows to define when models are considered consistent and the language derives how to restore this after changes. This distinction is most relevant for us, because it maps to different concepts in our formalization, which we present in Chapter 4. Although languages can actually contain imperative and declarative constructs, we make this rather broad distinction, as the basic distinction of whether the developer specifies how to preserve consistency or whether the language has to derive it from a declarative specification applies no matter whether the complete language or only single constructs of it can be considered declarative. In the classification of Czarnecki et al. [CH06], this is covered by different paradigms of transformation languages, especially distinguishing procedural and logic paradigms [CH06, Fig. 20], depending on whether they describe how to achieve or restore consistency, or whether they only define the constraints. They usually come along with a specific way of specifying values [CH06, Fig. 20], in particular imperative assignment and constraints.

Czarnecki et al. [CH06] also distinguish different transformation approaches, such as operational, relational, or graph-based approaches. Although we usually only consider transformations and not the actual languages to define them, the languages we explicitly consider or even propose in this thesis follow either an operational approach, which imperatively specifies how to preserve consistency, or a relational approach, which declaratively specifies constraints between two metamodels.

Examples for transformation languages for the MOF are the languages of the Query/View/Transformation (QVT) standard [QVT], namely QVT Operations (QVT-O), an imperative, operational and unidirectional language, and QVT Relations (QVT-R), a declarative, relational and bidirectional language. QVT-R is relevant for this thesis, as we propose a practical realization of one of our approaches for that language. It uses the Object Constraint Language (OCL) [OCL] for specifying the constraints that have to hold between instances of two metamodels. QVT-R is even multidirectional and allows to define relations between multiple metamodels, but we only consider the bidirectional case.

For the QVT languages, implementations for the EMF (see Subsection 2.2.2) exist. Further common EMF-based languages are VIsual Automated model

TRAnsformations (VIATRA) [Ber+15], an imperative and unidirectional transformation language, and the Atlas Transformation Language (ATL) [Jou+06; MTD17], which is a hybrid language containing imperative and declarative constructs. Another well-researched approach are Triple Graph Grammars (TGGs), originally developed as a graph transformation approach [Sch95], and later applied to the EMF [Leb+14] with tools like eMoflon [Anj14].

2.4.3. The Reactions Language

The Vitruvius framework (see Subsection 2.3.2) provides several languages for defining consistency preservation [Kra17]. This comprises the *Mappings language* [Wer16], which is a bidirectional, declarative language comparable to QVT-R, and the *Reactions language* [Kla16] for defining imperative, unidirectional transformations. While the Mappings language is used as a conceptual basis for the language that we propose in Chapter 12, the Reactions language is of specific importance for this thesis, because we use it for prototypical implementations and evaluations. The Vitruvius framework defines a transformation engine, which processes changes performed to a model and calls given transformation rules that implement an API. The rules update models that are accessed via a traceability model, which is called *correspondence model*. This correspondence model represents between which elements consistency has to be preserved. The Reactions language generates implementations of transformation rules according to this API provided by the framework. The Mappings language, in turn, generates specifications in the Reactions language.

A transformation rule defined in the Reactions language is called a *Reaction*. Listing 2.1 gives an impression of the language at an example that transforms a PCM component into a UML class. A Reaction specifies after which type of change it shall be executed, which, in this case, is the insertion of a component into a repository. It may call one or more reusable *routines* that restore consistency. Such a routine consists of a *match* block, which checks whether it is responsible for restoring consistency and retrieves all relevant elements from the models and the correspondence model, and an *action* block, which restores consistency. In the example, the routine retrieves an appropriate package in the UML model to place the class in. It then creates a class, assigns its name, and adds a correspondence between the elements. For a detailed explanation of the example, we refer to previous work [Kla+21].

```
reaction {
   after element pcm::Component inserted in
      pcm::Repository[components]
   call {
      val component = newValue
      createClass(component)
   }
}

routine createClass(pcm::Component component) {
   match {
      val componentsPkg = retrieve uml::Package
         corresponding to component.repository
         tagged with "componentsPackage"
   }
   action {
      val class = create uml::Class and initialize {
         class.package = componentsPkg
         class.name = component.name + "Impl"
      }
      add correspondence between component and class
   }
}
```

Listing 2.1: Reaction creating a UML class for a PCM component. Adapted from [Kla+21, Lst. 2].

2.5. Case Studies

We use case studies from component-based software engineering for several examples in this thesis that are more realistic than the ones based on a running example that we introduce in Section 3.4 and for the evaluation of several of our contributions. They cover a scenario already depicted in Chapter 1, which is based on three metamodels. The PCM [Reu+16] is used for defining the component-based architecture of a software system, the UML [UML] is used for depicting the fine-grained object-oriented design in terms of class models, and Java [Gos+18] depicts the implementation in code. The UML is defined in a standard based on the MOF and Java is specified with a grammar-based specification. Nevertheless, for all three languages an Ecore-based metamodel for the EMF (see Subsection 2.2.2) exists.

We assume basic concepts of class models of the UML and Java, or in general object-oriented programming languages, to be known to the reader. The elements of the PCM that we use in this thesis only require a broad understanding of those component-based architecture descriptions. Basic elements are *components*, *interfaces* and *data types*, which are all contained in a *repository*. Data types specific structures for data, including *primitive types*, such as integers or strings, *composite types*, which compose a type of multiple other types, and *collection types*, which can contain multiple elements of a defined other data type. Regarding interfaces, we only consider *operation interfaces*, which contain operation signatures consisting of return types and parameters, similar to methods in programming languages. The PCM also provides further types of interfaces, which we do not consider in this thesis. Finally, components define the reusable, architectural elements of a software systems. They have *provided* and *required roles*, which define on which interfaces a component depends and which interfaces it provides to other components. Since PCM models do not only specify the architecture of a software system but enable predictions of its performance, they allow to define an abstract behavior specification of services provided by components, called *service effect specifications*. We do not explain them in more detail, as we do not consider these behavior specifications in this thesis.

In the case studies used in thesis, we consider a specific notion of consistency between PCM, UML and Java models. We explain our notions of consistency and, in particular, of consistency relations in detail in Chapter 3 and Chapter 4. Broadly speaking, consistency relations define under which conditions one model is considered consistent to another. We depict the consistency relations for the metamodels of the case studies in such a general way that this broad notion is sufficient for their comprehension. In the way we introduce the relations, they are supposed to mean that if some elements are present in a model, according other elements need to be present in another model, such as that for every UML class a Java class with the same name has to exist.

The consistency relations between PCM, UML and Java consist of two parts. First, the relations between PCM and object-oriented design in both UML and Java were defined and explained in detail by Langhammer [LK15; Lan17]. He, in particular, proposed different options for relations between PCM and Java, which can be generalized to object-oriented design. We have selected the mapping of architectural components to classes and packages, as that mapping was studied most intensively and its implementation is most mature. This conforms to the mapping that we have already sketched in Chapter 1.

PCM Element	Object-Oriented Design Element
Repository	Three packages: `main`, `contracts`, `datatypes`
Basic component	Package in `main` package and a component realization class within that package
Operation interface	Interface in `contracts` package
Signature + parameters	Method + parameters
Composite datatype	Class in `datatypes` package with getter and setter for inner types
Collection datatype	Class in `datatypes` package that inherits from a collection type (e.g., `ArrayList` in Java)
Required role	Field typed with required interface in the component realization class and constructor parameter for the field in the component realization class
Provided role	Component realization class of providing component implements the provided interface

Table 2.1.: Consistency relations between elements of the PCM repository metamodel and object-oriented design elements (UML/Java). Adapted from [Lan17, Tab. 4.1].

Second, the relations between UML and Java reflect the usually implicitly known mapping between the two languages, as both similarly describe the object-oriented structure of a software system.

Table 2.1 sketches the relevant consistency relations between PCM models and object-oriented design, which can be reflected in both UML and Java. A PCM repository model consists of data types, interfaces (also denoted as *contracts*) and components, which are all contained in one repository. The repository is represented as a package structure of three packages in object-oriented design. Each component is represented as a package containing a so called *component realization class*. Interfaces with their signatures and parameters are mapped to corresponding object-oriented elements as they are. Composite data types are represented as a class containing the composed types, and collection data types are represented as subclasses of a collection type. Provided roles are realized by an implementation of the provided interfaces in the component realization class. A required role, on the contrary,

UML Element	Java Code Element
Package	Package
Class / Enumeration	Class / Enumeration
Interface	Interface
Method	Method
Parameter [0-1 .. 1]	Parameter of same type
Parameter [0-* .. 2-*]	Parameter of collection type with type parameter
Field [0-1 .. 1]	Field of same type
Field [0-* .. 2-*]	Field of collection type with type parameter
Association [0-1 .. 1]	Field of same type
Association [0-* .. 2-*]	Field of collection type with type parameter

Table 2.2.: Consistency relations between UML class models and Java code.

is represented as a field in the component realization class, which must be set via a constructor parameter. All these relations include further constraints for their features, especially regarding their names.

We have mentioned that PCM models can also contain *service effect specifications* as an abstract behavior specification of components, whose consistency to the implementation in Java was researched in detail by Langhammer [Lan17]. We do, however, not consider such behavioral specifications in our case studies, for which we explain the reasons in Subsection 3.1.2.

Table 2.2 shows the relevant consistency relations between UML models and Java code. They reflect the intuitive notion of the relation between the UML and Java of mostly one-to-one mappings, since we only consider Java elements that are present in the abstraction provided by the UML, i.e., we do especially not consider method bodies. The only special cases are fields having a type of another class in Java, which can be expressed as associations in the UML, as well as parameters, fields and associations, which can have multiplicities in the UML that have to be expressed as collection types with an appropriate type parameter in Java if the upper bound is higher than 1.

Notation	Description
$\mathbb{S} = \mathbb{s} = \{a, b, \ldots\}$	A set $\mathbb{S}$ or $\mathbb{s}$ of elements
$\mathfrak{T} = \mathfrak{t} = \langle a, b, \ldots \rangle$	A tuple $\mathfrak{T}$ or $\mathfrak{t}$ of elements
$S[] = s[] = [a, b, \ldots]$	A sequence $S[]$ or $s[]$ of elements
$S[i]$	Element at index i of sequence $S[]$
Func	A function

Table 2.3.: Notations for sets, tuples, sequences, and functions.

2.6. Mathematical Notations

For most of our definitions, we use standard mathematical notations. Whenever we deviate from that within the thesis, we explicitly denote it and define the used constructs. We use specific formatting especially for sets, tuples, sequences, and functions to ease their distinction. We introduce this notation in Table 2.3. Additionally, we define some shortcut operators for tuples, which we frequently require throughout the thesis.

We usually denote variables representing sets of any kinds of elements in blackboard bold font $\mathbb{S}$ and the definition of a set of elements by putting them in curly brackets, e.g., $\{a, b, \ldots\}$. Likewise, we denote variables representing tuples of elements in Gothic font $\mathfrak{T}$ and write elements forming a tuple in angle brackets, e.g., $\langle a, b, \ldots \rangle$. Finally, we denote variables representing sequences of elements by subsequent square brackets $S[]$ and the definition of a sequence of elements by putting them into square brackets, e.g., $[a, b, \ldots]$. To access an element at index i of a sequence $S[]$, we write $S[i]$. We denote the addition of an element e to a sequence $S[] = [s_1, \ldots, s_n]$ as:

$$S[] + e \coloneqq [s_1, \ldots, s_n, e]$$

Sequences are mathematically equal to tuples, but we make them explicit as representations of orders of potentially equal elements, rather than combining elements of potentially different types in tuples. This is why we define an access operator for contained elements of sequences. We deviate from the described formatting of sets and tuples in specific situations whenever the focus of the semantics of the variable is not that it is a set or a tuple. For example, if we consider a relation that is a set of tuples, we do not denote it in our set syntax, as its semantics is to be a relation and not a set. If we consider a set of relations, however, we denote it in the set syntax. We ensure that the meaning of the variables stays clear from the context.

We often use tuples to ensure that the elements can be indexed, although they cannot contain duplications and thus behave as sets if not interested in the order of elements. Since we need to treat the tuples similar to sets in several situations, especially to describe that a tuple contains an element or that is has a specific relation to another tuple, we define several operators which treat them as sets. For tuples $\mathfrak{t}$ and $\mathfrak{v}$ with $\mathfrak{t} = \langle t_1, \dots, t_n \rangle$, we define:

$$e \in \mathfrak{t} :\Leftrightarrow \exists\, i \in \{1, \dots, n\} : e = t_i$$
$$\mathfrak{t} \subseteq \mathfrak{v} :\Leftrightarrow \forall\, e \in \mathfrak{t} : e \in \mathfrak{v}$$
$$\mathfrak{t} \cap \mathfrak{v} := \{e \mid e \in \mathfrak{t} \wedge e \in \mathfrak{v}\}$$

Note that the intersection of tuples is not a tuple but a set, because we are not interested in matching their orders.

In several situations, we define binary relations, which are sets of pairs, whereat pairs are tuples of two elements. We define the concatenation of two relations to express their transitive relation. For two binary relations $R_1 = \{\langle a_l, a_r \rangle, \dots\}$ and $R_2 = \{\langle b_l, b_r \rangle, \dots\}$, we define their concatenation $R_1 \otimes R_2$ as:

$$R_1 \otimes R_2 := \{\langle a, b \rangle \mid \exists\, z : \langle a, z \rangle \in R_1 \wedge \langle z, b \rangle \in R_2\}$$

This conforms to the composition of relations often denoted as $R_1; R_2$.

We usually denote function names in small caps, e.g., Func. For functions, we use the standard notation for their composition. For two functions F_1 and F_2, we denote their composition for an input x as:

$$\mathrm{F}_1 \circ \mathrm{F}_2(x) := \mathrm{F}_1(\mathrm{F}_2(x))$$

For partial functions, we write $\mathrm{F}(x) = \perp$ if a function F is undefined for x.

3. Consistency, Processes, and Models

In this chapter, we discuss general terms and notions as considered by us to clarify the scope of this thesis. We discuss different dimensions of consistency, its specification and preservation, as well as the process of specifying consistency with a depiction of the involved roles and relevant scenarios. We introduce the general notion of models used in this thesis and the notations for them. Finally, we introduce a running example.

3.1. Dimensions of Consistency

In the following, we clarify different dimensions of how consistency can be considered and specified, which types of consistency can be distinguished, and how these types induce different processes of checking and enforcing them. This leads to the restriction of our work to normative specifications of preservation for structural consistency relations.

3.1.1. Normative and Descriptive Specification

So far, we have informally considered consistency as the absence of contradictions between different models. It is, however, unclear when to consider information in models contradictory. Consistency can be considered *normatively* or *descriptively* [Kra17, Sec. 3.1.2], depending on whether a notion of consistency already exists.

With a normative (or *prescriptive*) specification of consistency, we consider models consistent whenever we want them to be consistent. Thus, if someone specifies consistency, for example, in terms of a transformation, models are considered consistent when they adhere to that specification. Anything that this person defines as consistent is actually considered as consistent, i.e., the transformation *prescribes* consistency. Such a specification can always be considered *correct*, because there is no external specification to which it has to adhere. For example, it is usually not predefined under which conditions an architecture specification, be it defined in the UML, the PCM, or some other language, is considered consistent to its realization in code, so a transformation normatively defines how consistency is considered.

In the case of a *descriptive* specification of consistency, we assume that consistency is already defined and we have to adhere to that definition. Thus, if somebody specifies a transformation, it has to follow that existing definition of consistency. The transformation does only *describe* consistency. Such an existing specification may not exist explicitly but can exist implicitly, for example, because there is some common notion of consistency for specific languages. A descriptive specification may be *incorrect*, because it has to adhere to the existing definition of consistency. For example, there is, at least for most constructs, a common understanding of when UML class models and Java code are considered consistent, even if this understanding is not represented explicitly. Thus, any transformation has to describe that existing notion of consistency.

In this thesis, we always assume a normative specification of consistency. This does not mean that we exclude languages for which some notion of consistency already exists, such as the UML and Java code, but we assume that a specification of that consistency is normative. This means, if there is an existing notion of consistency, we do not consider whether the specification is correct with respect to that existing notion, but we assume it to be correct by construction. It is subject to other research, including general requirements engineering [TZL16] and especially transformation validation [AW15], to check whether a transformation is correct with respect to some expectation, which reflects an existing notion of consistency. This includes validation or verification of invariants [Cab+10] or contracts [AZK17; Val+12].

3.1.2. Structural and Behavioral Consistency

In addition to the distinction between normative and descriptive consistency specification, we can distinguish different types of consistency relations.

From a pragmatic perspective, we can at least differentiate between *structural* and *behavioral* consistency relations, conforming to the distinction of structural and behavioral models in the UML standard [UML]. While structural consistency concerns everything that has no execution semantics, behavioral consistency concerns semantics and thus also, for example, method bodies. Structural consistency can thus be checked without executing the model, comparable to the distinction between *static* and *execution* semantics of models, as introduced in Subsection 2.1.1. For example, having the same classes and method signatures in a UML model and Java code would be considered a structural relation, whereas the equivalence of a UML state machine and its Java implementation would be considered a behavioral relation, as they must have the same execution semantics. Thus, the mechanisms for checking these two types of consistency are likely to be different.

The execution semantics of models are often defined in a Turing-complete formalism, be it because the model has some semantics itself or because it is transformed into another specification of a Turing-complete formalism, such as executable code. Behavioral consistency relations referring to the execution semantics of models thus have to put Turing-complete specifications into relation. In consequence, one option for a clear distinction between behavioral and structural consistency relations is their decidability, since behavioral relations between Turing-complete specifications will, in general, be undecidable, while we would intuitively assume structural relations to be decidable. This leads to different levels of statements that we can make about the different types of relations, especially including existentially and universally quantified statements.

Universally Quantified: The approach can validate that a consistency relation holds for all instances of the modeled system. This can, for example, be achieved with verification techniques, model checking and other analyses. An exemplary application scenario is the equivalence of decidable consistency relations.

Existentially Quantified: The approach can validate that a consistency relation holds at least for some instances of the modeled system. This can, for example, be achieved with tests. In the best case, the test cases cover a representative subset of the possible instances. An exemplary application scenario is the equivalence of undecidable behavior descriptions.

Statistical: The approach can make statistical statements about the consistency relations, such as the probability for a relation to be fulfilled in an instance. This can, for example, be achieved by simulation. An exemplary application is consistency between quality requirements and the system realization, such as the probability that for a given requirements model and an according implementation of the system the implementation fulfills a performance requirement.

While universally quantified statements can only be made about decidable consistency relations, i.e., structural relations, existentially quantified and statistical statements can be made for both of them, thus also for behavioral consistency relations.

At a Dagstuhl seminar about multidirectional transformations [Cle+19], different consistency relation scenarios in which more than two models are related were considered. A central hypothesis was that relations between more than two models can be decomposed into binary relations as long as the relations are structural. Whether two or more models fulfill a behavioral requirement may, however, not be easily decomposed into multiple binary relations between model pairs.

In this thesis, we focus on structural relations, i.e., relations that are decidable and about which we can make universally quantified statements without executing the models. This does not mean that our contributions are restricted to these kinds of structural relations. In fact, we do not make assumptions that exclude other types of consistency relations, so as long as they conform to the formalism that we propose our contributions also apply for them. We do, however, only consider structural relations in our examples, considerations and evaluations, such that a generalization to other relations types needs to be evaluated.

3.1.3. Checking and Preserving Consistency

Based on a specification of consistency and potentially its preservation, consistency between different models can be checked and potentially enforced during the development of a system (cf. [QVT]). Basically, we can distinguish whether a process is only *checking* or also *preserving* consistency. Some consistency relations may only be checked and have to be manually ensured, whereas others can (semi-)automatically be enforced.

Behavioral consistency relations may be hard to enforce but can, in the best case, at least be checked. This also includes relations for quality properties, such as performance of an implementation regarding performance requirements. For example, it will usually not be possible to automatically adapt the source code after a change leads to the violation of a consistency relation between the implementation's performance and the performance requirements. On the contrary, we expect that structural consistency relations can often also be enforced, at least by collecting additional information from the developer, because for redundant representations of structural elements likely only one or few options to restore consistency exist in contrast to solving the violation of a performance requirement.

In addition, it can be reasonable to check and enforce structural consistency relations more often, because they can be checked in a rather fine-grained way and more efficiently, in the extreme case even just-in-time. Checking behavioral relations may also include long-running analyses or simulations and may only make sense at specific points in time, indicated by the developer. This at least applies to relations for which only existentially quantified or statistical statements can be made. For example, adding an architectural component to a PCM model can and should directly lead to the creation of an implementing class in Java code. But whether a Java method fulfills some behavioral consistency relation to another model, such as the behavioral service specifications in a PCM model, usually makes sense less often, as it requires more coarse-grained modifications to achieve consistency, such as rewriting a complete method or multiple of its statements, whereas changes of structural relations often only concern a single element, such as a name or a type of a parameter. Checking such behavioral consistency relations may thus take more time because of complex analyses or simulations to run. The developer may explicitly indicate when a development state is reached at which behavioral consistency relations can be checked. For behavioral relations about which universally quantified statements can be made, such as a security analysis, it may be up to the scenario whether checks should be performed just-in-time or only at specific points in time.

In consequence, the distinction between structural and behavioral consistency relations is also relevant for the processes of checking and preserving consistency. While structural consistency relations may be preserved often in a fine-grained way, behavioral consistency relations may be checked less often. We depict the proposed process in Figure 3.1. In the best case, a consistency mechanism can give hints to potential behavioral consistency violations more often. For example, a performance-relevant modification of the implementation could lead to a hint for the developer that performance

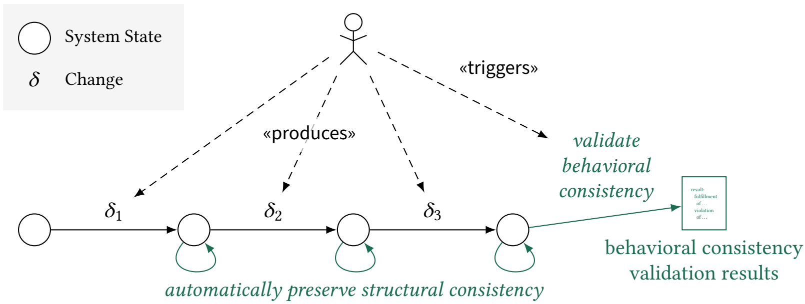


Figure 3.1.: Proposed process for continuously and automatically preserving structural and explicitly checking behavioral consistency.

may be affected by his modification with the information about the previous analysis result, such that he or she can guess whether his or her modification will violate the requirement. Given the information that a response time requirement of 10 milliseconds was fulfilled during the last validation by an actual response time of 1 millisecond can help the developer to decide that his or her modification will unlikely violate that requirement.

In this thesis, we are interested in processes that continuously preserve and not only check consistency. This is why we explicitly focus on structural consistency relations in this thesis, although the insights might be transferable to behavioral relations as well. As another consequence, the structural relations that we consider are supposed to be decomposable into binary relations, as discussed in Subsection 3.1.2.

In addition, we restrict ourselves to supporting the case in which only one model is changed at a time and for which consistency with the other models needs to be preserved. In general, there may be multiple developers performing changes to one or more models concurrently. This scenario is already difficult for the case in which only two models need to be kept consistent by a single transformation, as changes can be conflicting and conflicts need to be resolved. It becomes even more complicated when transformations preserve consistency of multiple models and thus conflicts need to be resolved across multiple models and transformations. We refer this this topic as future work and discuss solution options in Section 15.2.

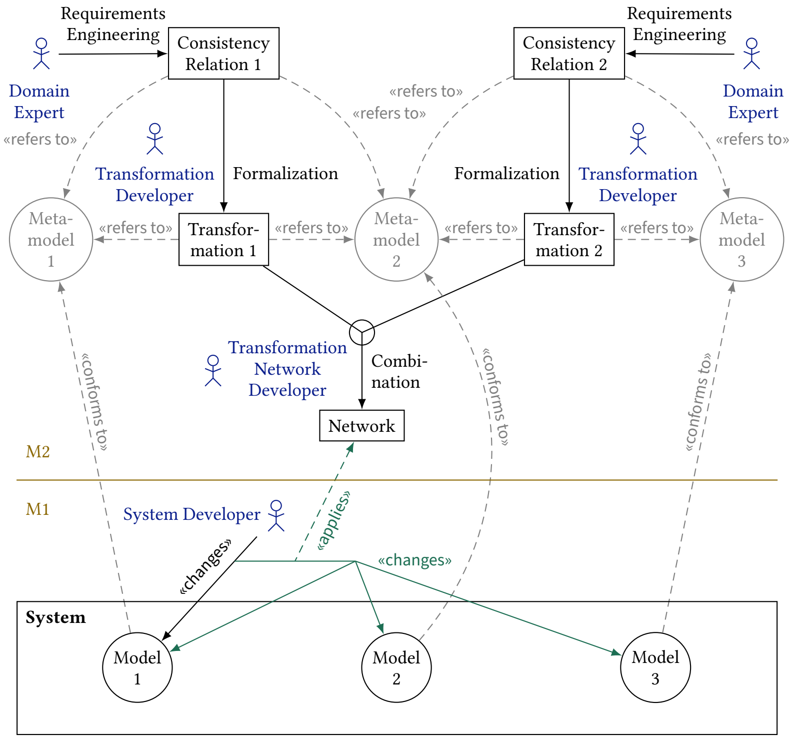


Figure 3.2.: Roles involved in a process for specifying a transformation network, their responsibilities and dependencies. Extended from [TK19, Fig. 2].

3.2. Consistency Specification Process

In this thesis, we are concerned with the process of specifying consistency in terms of a transformation network and different problems arising in that process. We therefore discuss which roles are involved in that process and which scenarios can be considered that induce specific requirements and exemplify the application contexts of our contributions. Figure 3.2 gives an overview of the roles and the essential specification process. While that process focuses on the metamodel level (M2), a transformation network is finally applied at the model level (M1) to an actual system under development.

3.2.1. Roles

The specification of a transformation network involves the definition of the individual transformations by *domain experts* and *transformation developers* as well as their combination to a network by *transformation network developers*. The usage of the network involves its application to changes to a system under development by a *system developer*, sometimes also called *tool user* [TK19]. Apart from the explicit transformation network, these roles and their responsibilities are comparable to the ones that were defined in a working group of a Dagstuhl seminar, in which the author of this thesis participated [TK19].

A domain expert has the knowledge about the consistency relations between two (or more) tools and their languages or, more specifically, the metamodels describing them. He or she performs the requirements engineering task for the information to define in a transformation. A transformation developer is then responsible for formalizing these relations and their preservation in a transformation. We usually only refer to the transformation developer, as it is not relevant for us where the information about the relations comes from but only that it is encoded into a transformation. Finally, a transformation network developer combines different transformations, which were usually developed by different transformation developers, to a transformation network. It may even be possible that several transformation network developers compose several transformation networks to a larger transformation network. Whenever the distinction is not relevant, we refer to transformation and transformation network developers as *transformation developers*.

Actual systems are developed with the use of transformation networks by system developers, who perform changes of models via the tools they use, which is why they are also called tool users. Usually different system developers will be responsible for different models. In our introductory example, we distinguished between software architects, developers, performance and requirements engineers. Performing changes leads to the application of the transformation network to restore consistency of the models. In this thesis, we refer to system developers also as *users*, as they are the ones using the transformation networks we are concerned with.

The roles reflect the different responsibilities when specifying and using transformation networks. Several of them can, however, be fulfilled by the same persons. This especially applies to domain experts and transformation developers. The same person may know about the relations and formalize them in a transformation. Potentially, a domain expert may even be the one who develops an actual system as a system developer.

3.2.2. Scenarios

Both for the development of transformations as well as for their combination to a network, different development scenarios can be distinguished. Transformations can be developed generically or specific for a project.

Generic: Transformations are developed as artifacts off-the-shelf, which can be used in any project. This especially applies for descriptive transformations (see Subsection 3.1.1), which encode a common understanding of consistency, such as for UML class models and Java code.

Project-Specific: Transformations are developed for a specific project. This can occur if a project requires specific rules how elements shall be related. For example, the mapping of components to their implementation can be project-specific [Lan17]. Eventually, such transformations can later be used in a generic way.

The combination of transformations to networks can be distinguished especially regarding the point in time at which the combination takes place.

Big Bang: Transformations are developed first, and after they have been completed a transformation network developer combines them to a network. Problems regarding the compatibility of the transformations are first recognized during this combination, thus transformations may need to be adapted afterwards to properly work together.

Continuous: Transformations are combined to a network already during their development. Starting with partial or even empty transformations, the structure of the network can be defined early. This allows for a continuous validation of compatibility of the developed transformations. Ultimately, even an online checking of compatibility after each change to a transformation can be performed to get early feedback.

For us, it is not relevant whether transformations are developed in a generic or project-specific way. The distinction of scenarios in which transformation networks are developed is, however, of special interest. It can be beneficial for transformation developers to get feedback about the compatibility of

Properties and Classes	
P	Property (attribute or reference)
$I_P = \{p_1, p_2, \ldots\}$	Property values of a property P
$C = \langle P_1, \ldots, P_n \rangle$	Class
$I_C = \{o = \langle p_1, \ldots, p_n \rangle \mid p_i \in I_{P_i}\}$	Instances (objects) of a class C
$o \in I_C$	Object of a class C
(Meta-)Models	
$M = \{C_1, \ldots, C_m\}$	Metamodel
$I_M = \{m \mid m \subseteq \bigcup_{C \in M} I_C\}$	Instances of a metamodel
$\mathfrak{M} = \langle M_1, \ldots, M_k \rangle$	Tuple of metamodels
$I_{\mathfrak{M}} = I_{M_1} \times \cdots \times I_{M_k}$ $= \{\langle m_1, \ldots, m_k \rangle \mid m_i \in I_{M_i}\}$	Instances of a metamodel tuple $\mathfrak{M} = \langle M_1, \ldots, M_k \rangle$
$m \in I_M$	Model of metamodel M
$\mathfrak{m} \in I_{\mathfrak{M}}$	Model tuple of a metamodel tuple $\mathfrak{M}$

Table 3.1.: Models, metamodels, their elements and notations.

their developed transformations with others on-the-fly. This makes locating a problem easier, because only the recent changes may have introduced it, whereas with an a posteriori checking in a big bang process the effort to find compatibility problems may increase because of missing locality.

While generic and project-specific transformations can obviously be mixed in a single project, the combination processes may also be mixed. Some transformations may be integrated in a big bang fashion whereas others are integrated continuously. Project specificity of transformations can imply this, because a generic transformation cannot be integrated continuously.

3.3. Models and Metamodels

The most essential elements used for descriptions in this thesis are models and the metamodels they conform to. In Chapter 2, we have introduced what we consider a model and that we adhere to the MOF modeling formalism.

We use a sufficiently simplified notion of models, metamodels, and their elements, which we summarize in Table 3.1. In the following, we introduce the used notation and its conventions, as well as the used modeling elements. Finally, we clarify assumptions that we make and discuss their impact.

3.3.1. Notation and Conventions

We use uppercase variables for elements at the metamodel level (M2), such as M for a metamodel or C for a class, an depict elements at the model level (M1) in lowercase, such as m for a model and o for an object.

We use the notations for sets and tuples introduced in Section 2.6 for denoting sets and tuples of the different elements, such as metamodels and models. When considering multiple metamodels or models, we are usually not interested in their order and the same model or metamodel cannot appear twice. Still, we always treat them as tuples rather than sets to be able to easily relate a model to its metamodel by its index within the tuple. Thus, if not further specified, we use the same indices to relate an element at the metamodel and the model level, such as as m_1 being an instance of M_1, i.e., $m_1 \in I_{M_1}$. This could also be expressed by an explicit instantiation relation, but the used notation is more concise and thus proposes to easy readability.

3.3.2. Modeling Elements

In general, we consider metamodels as a composition of metaclasses, which, in turn, are composed of properties representing attributes or references. Models instantiate metamodels and are composed of objects, which are instances of metaclasses and, in turn, consist of property values, which instantiate properties.

We denote *properties*, which are the information a metaclass consists of, such as attributes or references, as P and the *property values* as instances of a property as $I_P = \{p_1, p_2, \ldots\}$ of property P. We do not need to further differentiate between attributes and references, like it is done in other formalizations such as the OCL standard [OCL, A.1] or the thesis of Kramer [Kra17, Sec. 2.3.2].

We denote *metaclasses*, also shortly called *classes*, as tuples of properties $C = \langle P_1, \ldots, P_n \rangle$. Instances of a class are *objects*, each being a tuple of instances of the properties of the class. We denote all instances of a class $C = \langle P_1, \ldots, P_n \rangle$ as $I_C = \{o = \langle p_1, \ldots, p_n \rangle \mid p_i \in I_{P_i}\}$.

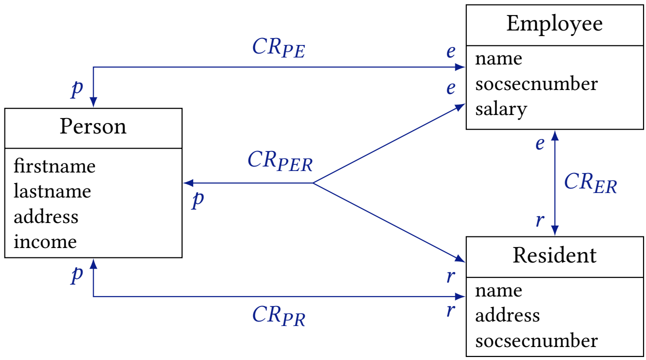
We denote a metamodel $M = \{C_1, \ldots, C_m\}$ as a finite set of classes. The instances of a metamodel are sets of objects $I_M = \{m \mid m \subseteq \bigcup_{C \in M} I_C\}$. In other work such as the articles by Stevens [Ste20b], such instance sets are also called *model sets* and implicitly define a metamodel, thus representing a lightweight definition of metamodels by simply enumerating its instances. Each instance of a metamodel is called a *model* and represents a finite set of objects that instantiate the classes in the metamodel. For a tuple of metamodels $\mathfrak{M} = \langle M_1, \ldots, M_k \rangle$, we denote the set that contains all sets of instances of those metamodels as $I_{\mathfrak{M}} = \{\langle m_1, \ldots, m_k \rangle \mid m_i \in I_{M_i}\}$.

With I_C and I_M, we denote the sets of instances of a class and metamodels, i.e., the objects and models instantiating them. Usually, additional constraints exist that further restrict these sets. For example, a property can represent a reference to another object, thus if a class contains a specific property value representing a reference to an object, the referenced object must be contained in the model as well. Thus, the sets of *valid* instances of classes and metamodels are usually only subsets of the sets we denote with I_C and I_M, respectively. For reasons of simplicity, we do, however, usually only refer to the denoted instance sets. The statements still apply to the sets of valid objects and models as subsets of the considered sets.

3.3.3. Assumptions

We assume models to be finite, so for each model m, we assume that $|m| < \infty$. Additionally, our proposed formalism assumes objects to be unique within a model m. This is already implicitly covered by the definition of I_M for the instances of a metamodel M.

In practice, it is usually allowed to have the same object, i.e., an element with the same type, attribute and reference values, multiple times within the same model. This is, however, only a matter of identity, which, in practice, is given at least by different objects being placed at specific places in memory. We assume, without loss of generality, the necessary information to distinguish two elements to be represented within their properties.



$$CR_{PER} = \{\langle p, e, r\rangle \mid p.firstname + \text{"␣"} + p.lastname = e.name = r.name$$
$$\wedge\ p.address = r.address \wedge p.income = e.salary$$
$$\wedge\ e.socsecnumber = r.socsecnumber\}$$

$$CR_{PE} = \{\langle p, e\rangle \mid p.firstname + \text{"␣"} + p.lastname = e.name \wedge p.income = e.salary\}$$

$$CR_{PR} = \{\langle p, r\rangle \mid p.firstname + \text{"␣"} + p.lastname = r.name \wedge p.address = r.address\}$$

$$CR_{ER} = \{\langle e, r\rangle \mid e.name = r.name \wedge e.socsecnumber = r.socsecnumber\}$$

Figure 3.3.: Three simple metamodels for persons, employees and residents. One ternary relation CR_{PER} between them and three binary relations CR_{PE}, CR_{PR}, CR_{ER} between each pair of them describing consistency. Adapted from [Kla+20, Fig. 1].

3.4. Running Example

We use different variations of a running example throughout several parts of this thesis. The basic example is depicted in Figure 3.3. It contains three metamodels, one with persons, one with employees and one with residents, each containing the name and some information specific for that metamodel. Although these metamodels are rather simple and do not cover metamodels from the software engineering domain, they are sufficient to explain many concepts in this thesis and are easy to comprehend.

The example also contains a description of consistency between these three metamodels, although only informally given at this point and more precisely defined later on. It requires that if any person, employee or resident is contained in a model, there must also be the other two elements with the same names, addresses, incomes and social security numbers. Like for the metamodels themselves, it can be challenged whether this consistency relation may be reasonable, but it is easy to comprehend and sufficient for explaining the essential concepts and also several issues in this thesis. This relation can be expressed as a single ternary relation, denoted as CR_{PER}, or as three binary relations $CR_{PE}, CR_{PR}, CR_{ER}$. Three models fulfill the ternary relation in exactly those cases in which all pairs fulfills the binary relations. The relations consist of tuples of the elements that are considered consistent, i.e., the element pairs or triples that fulfill the specified constraints of their property values.

The metamodels and consistency relations are defined in a way such that no pair of the three binary consistency relations is equivalent to the ternary relation in the sense that the same models are considered consistent to these two binary relations whenever they are considered consistent to the ternary relation. This is a consequence of each pair of metamodels sharing some unique information, which is the income, the address and the social security number. In consequence, we cannot omit one of the binary relations without loosing consistency guarantees compared to the ternary relation.

Part II.

Building Correct Transformation Networks

4. Correctness in Transformation Networks

In this chapter, we first discuss a rather informal notion of consistency and its preservation. It is supposed to describe the different dimensions in which consistency and its preservation can be considered to then discuss how *correctness* can be reasonably defined. After identifying the correctness notion that is relevant in the context of our work, we define a suitable formal notion of consistency. We formally define correctness of different artifacts relevant for that notion of consistency. Finally, we present a refined notion of consistency, which we do not require for the initial overview, but which we later use for several detailed considerations.

This chapter thus constitutes our contribution **C 1.1**, which is composed of four subordinate contributions: a discussion of consistency notions; a discussion and determination of correctness notions for consistency specifications; a formalization of a relevant correctness notion; and finally a refinement of our consistency notion for later detailed considerations. It answers the following research question:

RQ 1.1: What are relevant notions of correctness in transformation networks and how can they be formalized?

Parts of the contributions in this chapter have been published in previous work [Kla18; Kla+21; Kla+20]. We have motivated and informally derived the correctness notion that we formalize in the following and gave an overview of the goal regarding correctness of transformation networks [Kla18]. We have used a simplified version of the formalization that we introduce in this chapter and especially identified the challenge of orchestration [Kla+21], which is central for the formalization of transformation networks. Finally, we have introduced a fine-grained consistency notion [Kla+20], which is required for detailed statements on compatibility.

4.1. Notions of Consistency and its Preservation

We begin with an informal discussion of different ways to consider consistency and its preservation. This involves *intensional* and *extensional*, as well *monolithic* and *modular* notions, and different execution strategies.

4.1.1. Intensional and Extensional Consistency Notions

When we consider a tuple of models, we may intuitively assume it to be consistent if it fulfills some kind of constraints. Defining these constraints to derive or check whether a given tuple of models is consistent constitutes an *intensional specification* of consistency, because the set that contains all consistent model tuples is intensionally represented by these constraints and can be derived from it. We can consider a set of constraints as a predicate, i.e., a Boolean-valued function P, which indicates whether a model tuple $\mathfrak{m} \in I_{\mathfrak{M}}$ fulfills the constraints $P : I_{\mathfrak{M}} \rightarrow \{\textsc{true}, \textsc{false}\}$. Then we can say that:

$$\mathfrak{m} \textit{ consistent to } P :\Leftrightarrow P(\mathfrak{m}) = \textsc{true}$$

Alternatively, one can enumerate the (possibly infinite number of) consistent tuples of models. Thus, a model tuple is considered consistent if that enumeration contains it. This constitutes an *extensional specification* of consistency. Given such an enumeration $E = \{\mathfrak{m} \mid \mathfrak{m} \textit{ is consistent}\}$, we can say that:

$$\mathfrak{m} \textit{ consistent to } E :\Leftrightarrow \mathfrak{m} \in E$$

Both kinds of specification have equal expressiveness. For each intensional specification, the extensional one can be derived by enumerating all models that fulfill the constraints:

$$E = \{\mathfrak{m} \mid P(\mathfrak{m}) = \textsc{true}\}$$

An extensional specification can also be transferred to an intensional one by defining constraints that are fulfilled by exactly the enumerated instances:

$$P(\mathfrak{m}) \mapsto \begin{cases} \textsc{true}, & \textit{if } \mathfrak{m} \in E \\ \textsc{false}, & \textit{if } \mathfrak{m} \notin E \end{cases}$$

For us, it will only be relevant that an intensional specification can be transformed into an extensional one.

A developer who defines consistency usually wants to use an intensional specification, as tools like transformation languages allow the specification of constraints rather than enumerating consistent instances. Since there is usually an infinite number of consistent models, he or she cannot explicitly enumerate them but only define constraints that allow to derive them. From a theoretical perspective, however, we prefer to consider extensional specifications, because they allow to directly apply set theory in a concise way. Due to the fact that each intensional specification can be transformed into an extensional one, we can make theoretical statements about extensional specifications that also hold for intensional ones. In the following, we always consider extensional specifications unless otherwise stated. So we define which models are considered consistent in terms of relations, which we also call *consistency relations*.

4.1.2. Monolithic and Modular Consistency Notions

Consistency, be it specified intensionally or extensionally, can be considered in an either monolithic or modular way. Having a single specification of consistency for an arbitrary number of models constitutes a *monolithic* notion of consistency. Like discussed for intensional and extensional consistency specifications, this can be expressed by a tuple of models fulfilling constraints or being contained in a relation. A *modular* notion of consistency considers several relations for subsets of the relevant metamodels, which together define when models are considered consistent.

For an extensional notion of consistency between three metamodels M_1, M_2 and M_3, a modular specification could manifest in three relations $CR_{1,2}, CR_{1,3}$ and $CR_{2,3}$ defining the model pairs that are considered consistent. If two models are consistent to one of the relations, we can say that they are *locally* consistent to that relation. We are, however, interested in whether models are *globally* consistent to all these relations, so we say:

$$\begin{aligned}
&m_1, m_2, m_3 \textit{ are consistent } :\Leftrightarrow \\
&\quad \langle m_1, m_2 \rangle \in CR_{1,2} \wedge \langle m_1, m_3 \rangle \in CR_{1,3} \wedge \langle m_2, m_3 \rangle \in CR_{2,3}
\end{aligned}$$

Due to the assumptions of independent development and modular reuse, which we have defined in Subsection 1.3.2, we are interested in a modular notion of consistency. In the example, we have considered a modular notion based on binary relation. Such a modular notion, however, can also be based on multiple multiary relations. But even with multiary relations, modularity is necessary for reasons of independent development and reuse. For reasons of simplicity, we stick to modular notions of binary relations, although most of our considerations can be transferred to multiary ones.

4.1.3. Consistency Preservation

Consistency preservation is the process of ensuring that models stay consistent. Based on a notion of consistency relations that describe when models are considered consistent, this process ensures that models stay in that relation. If models become changed such they that are not in the relation anymore, consistency preservation updates the models such that they are in that relation again. In consequence, consistency preservation is always relative to relations defining consistency.

Consistency preservation can be considered as a function $\textsc{Cp}$ that takes (potentially inconsistent) models and returns a consistent tuple of models:

$$\textsc{Cp} : I_{\mathfrak{M}} \rightarrow I_{\mathfrak{M}}$$
$$\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \textsc{Cp}(\mathfrak{m}) \textit{ is consistent}$$

The definition of *is consistent* depends on whether we rely on a monolithic or modular notion of consistency. Thus it may require the models to be in one or multiple relations. For example, given a monolithic relation CR, $\textsc{Cp}$ is supposed to fulfill that:

$$\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \textsc{Cp}(\mathfrak{m}) \in CR$$

Since these functions define how consistency is preserved, we also call them *consistency preservation rules.*

Like for the proposed notion of consistency, we can also consider consistency preservation in an either monolithic or modular way. With a modular notion of consistency preservation, we may have multiple consistency preservation rules that preserve consistency, each of them for a consistency relation

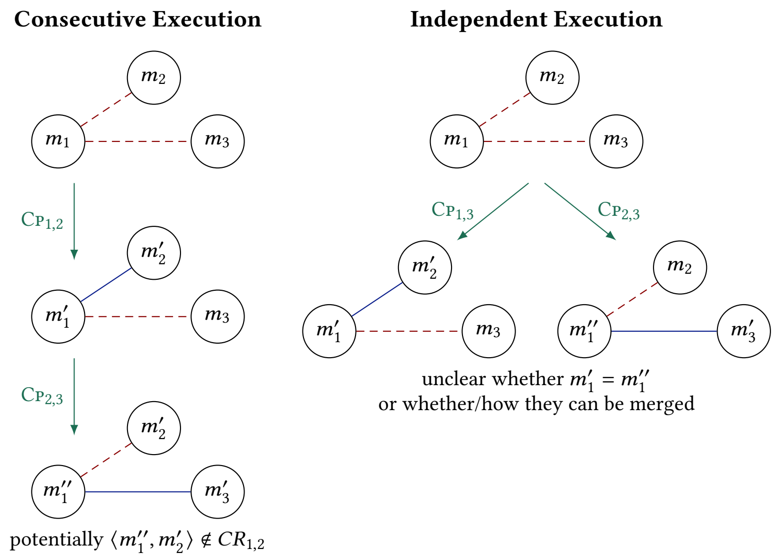


Figure 4.1.: Scenarios for independently executing consistency preservation rules on input models and consecutively executing them on the results of other rules. Circles denote models, lines between models denote fulfilled (solid blue) or violated (dashed red) consistency relations, and arrows between the model states (unidirectional green) denote the conduction of user changes or consistency preservation execution.

that defines consistency for a subset of the involved models. Unlike for the relations defining consistency, which can be evaluated independently to identify whether models are consistent, the functions, i.e., consistency preservation rules, cannot be evaluated independently. If each function is executed independently, each of them returns new models that may need to be merged. This is exemplified in the following scenario, which is also depicted in Figure 4.1. Imagine two functions $\textsc{Cp}_{1,2}$ and $\textsc{Cp}_{1,3}$ that preserve consistency for relations $CR_{1,2}$ and $CR_{1,3}$, respectively. Consider the input models $\langle m_1, m_2, m_3 \rangle$ that are not consistent to $CR_{1,2}$ and $CR_{1,3}$, i.e., $\langle m_1, m_2 \rangle \notin CR_{1,2}$ and $\langle m_1, m_3 \rangle \notin CR_{1,3}$. This can, for example, occur because m_1 was changed by a user. Now if we apply the functions independently, we have $\textsc{Cp}_{1,2}(\langle m_1, m_2 \rangle) = \langle m_1', m_2' \rangle \in CR_{1,2}$ and $\textsc{Cp}_{1,3}(\langle m_1, m_3 \rangle) = \langle m_1'', m_3' \rangle \in CR_{1,3}$. It is now unclear how to unify m_1' and m_1'' to m_1''', such that $\langle m_1''', m_2' \rangle \in CR_{1,2}$ and $\langle m_1''', m_3' \rangle \in CR_{1,3}$.

An intuitive approach to execute the functions is their composition, i.e., a consecutive execution that does not apply the functions for consistency preservation to the original models but to the models delivered by the previous executions of the functions, which is also exemplarily depicted in Figure 4.1. If we consecutively apply the two given functions, we know that $\textsc{Cp}_{1,2}(\langle m_1, m_2 \rangle) = \langle m'_1, m'_2 \rangle \in CR_{1,2}$ and $\textsc{Cp}_{1,3}(\langle m'_1, m_3 \rangle) = \langle m''_1, m'_3 \rangle \in CR_{1,3}$. It is, however, unclear whether $\langle m''_1, m'_2 \rangle \in CR_{1,2}$, so it may be necessary to execute $\textsc{Cp}_{1,2}$ again. In fact, we need some method to decide in which order and how often the consistency preservation rules are applied to result in a consistent tuple of models. We call this an *orchestration*. The challenge to find an execution order of transformations without leading to execution cycles has also been identified by Kramer [Kra17, Sec. 3.9].

Even if consistency preservation rules were supposed to only modify one model instead of two, the same problems of unifying changes of their independent execution or orchestrating their consecutive execution occur as soon as there are two sequences of consistency preservation rules that change the same models.

In our work, we follow the approach of orchestrating and consecutively executing consistency preservation rules. The benefits of this approach are twofold. First, there is no additional logic required for unifying the changes performed by independently executed consistency preservation rules. Second, the unification may deliver a model that is not consistent to any of the consistency relations anymore, whereas consecutive execution at least guarantees that the models are consistent to the last applied consistency preservation rule. With this approach, the repeated execution of consistency preservation rules can be seen as a negotiation of a solution by reacting to the changes the other consistency preservation rules performed.

Remark. Finally, every monolithic notion of consistency and its preservation can be considered a special case of a modular notion. Having only one consistency relation and one function that preserves it degrades the problem by making the necessity to perform an orchestration of functions obsolete.

For now, the introduced consistency preservation rules can be any kind of functions that return consistent models. Their realization may, for example, be transformations that define how to react to certain changes for restoring consistency, or constraint solvers that find consistent models by solving

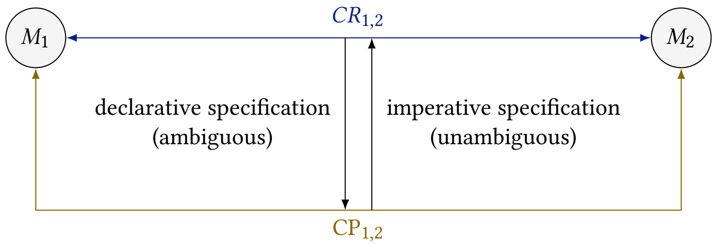


Figure 4.2.: Declarative and imperative specification of consistency relations and consistency preservation rules for two metamodels M_1 and M_2.

consistency constraints. We do not yet need to consider how these functions are realized to derive consistent models, although we later focus on transformation-based approaches.

4.1.4. Declarative and Imperative Specifications

We have discussed that consistency preservation can be considered as functions, called consistency preservation rules, that preserve consistency according to some relations. In practice, however, one will usually not specify both the consistency relation and the consistency preservation rule that preserves it. Instead, one artifact is given and the other is implied or derived. This leads to the two approaches of *declarative* and *imperative* consistency specifications, depending on whether the specification defines *how* consistency is achieved. The relation between the two approaches regarding a consistency relation and a consistency preservation rule is depicted in Figure 4.2.

As a first option, a developer may only define relations that specify consistency. Functions that preserve these relations can be derived from that. This is called a *declarative* specification, because it only declares when models are consistent but not *how* consistency is achieved. In general, there are multiple valid options for deriving a consistency preservation rule from a relation. It can, for example, calculate the result with minimal differences to the input according to some defined metric. Or, especially if there is an intensional specification of the relations, the approach may consider the type of input change and calculate an appropriate change according to the constraints in the intensional specification. This approach is followed by many declarative transformation languages, such as QVT-R [QVT] or TGGs [Anj+14].

As a second option, a developer can define consistency preservation rules without explicitly specifying the consistency relations to which they preserve consistency. Instead, these functions imply the underlying consistency relations that they preserve, at least if we assume that a consistency preservation rule does not perform changes when the input models are already consistent. Given a function $\textsc{Cp}$, the relation CR it preserves is implied by its fixed points: $CR = \{\mathfrak{m} \mid \textsc{Cp}(\mathfrak{m}) = \mathfrak{m}\}$. If a function preserving consistency does not perform any changes, the models are, by definition, consistent. Usually, we will assume that such a function returns consistent models with a single application. Thus, if it does not perform changes when the input models are already consistent, the function is idempotent and then the consistency relation is given by its image, i.e., $CR = \{\mathfrak{m} \mid \exists\, \mathfrak{m}' : \textsc{Cp}(\mathfrak{m}') = \mathfrak{m}\}$. This is called an *imperative* specification, because it declares *how* consistency can be achieved. Such an approach is followed by many imperative transformation languages, such as QVT-O [QVT].

4.1.5. Consistency Preservation Artifacts

We have discussed that consistency can be considered in a monolithic or modular way. We have, however, also mentioned that the monolithic case can be considered as a special case of the modular one. For the general case, we thus know from the previous considerations that in a consistency preservation process at least specifications that define consistency, called *consistency relations*, functions that preserve consistency, called *consistency preservation rules*, and a function for orchestrating the functions, in the following called *orchestration function*, are necessary. Finally, we also need a function that applies the consistency preservation rules in the order that is determined by the orchestration function, which we call the *application function*. To summarize, we consider the following four artifacts necessary to handle consistency preservation.

Consistency Relations: Binary relations that specify which pairs of models shall be considered consistent.

Consistency Preservation Rules: Functions that restore consistency for a pair of models that became inconsistent by modification.

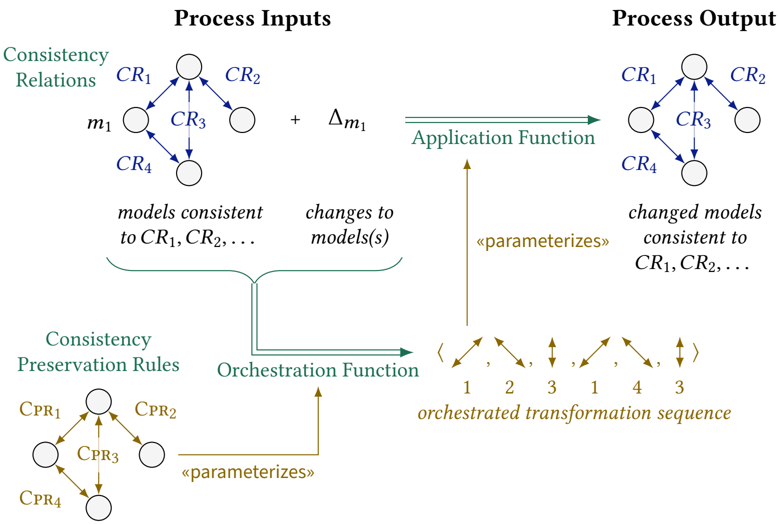


Figure 4.3.: Execution process and artifacts for a modular consistency specification. Central artifacts are annotated in (green) normal font.

Orchestration Function: A function that determines the execution order of the consistency preservation rules to restore consistency.

Application Function: A function that applies the consistency preservation rules in the order determined by the orchestration function.

We explicitly distinguish the orchestration and the application to be able to make more fine-grained statements about the responsibilities for the orchestration and its actual execution. This is particularly useful to determine the behavior in cases in which no orchestration of transformations that results in consistent models can be found. The process is depicted in Figure 4.3. Given models that are consistent according to some consistency relations and changes to them that lead to inconsistencies, the orchestration function delivers an order of consistency preservation rules, which is used to parametrize the application function that executes these rules in the given order. The result is, in the best case, a model tuple that is consistent to the relations again.

4.2. Notions of Correctness for Consistency Specifications

Before we formally define the above introduced artifacts, such as consistency relations, consistency preservation rules, an orchestration function and an application function, we first discuss different notions of *correctness* for them. Since there are different dimensions of correctness, we need to clarify which of them is relevant in the context of our research questions and will be defined in the formalization.

4.2.1. Relative Correctness Notions

The overall objective regarding correctness of consistency preservation is to find models that are actually consistent. Intuitively speaking, artifacts are correct if they fulfill their intended purpose. In our case, this means that consistency relations should consider models consistent whenever they are actually supposed to be considered consistent. Consistency preservation rules should return models that are consistent according to a consistency relation to be considered correct. This also conforms to existing notions of correctness for transformations [Ste10], which realize consistency preservation rules. And finally, the orchestration and application functions should execute the consistency preservation rules such that they yield models that are consistent according to all relations afterwards.

Correctness of an artifact is usually considered with respect to some other specification, be it formally defined or only an informal notion. For example, consistency relations may be considered correct with respect to some informal notion of correctness that is collected by domain experts and requirements engineers. A consistency preservation rule should always be consistent with respect to a consistency relation. As discussed before, this relation may either be defined explicitly and the preservation rule has to be correct with respect to it, or it may be induced by the fixed points of the preservation rule. In the latter case, the consistency preservation rule will always be correct by construction.

4.2.2. Correctness regarding Global Knowledge

We previously distinguished between monolithic and modular consistency notions. In the above considerations, we have related the artifacts of a modular specification to each other. Another notion of correctness can be defined by relating a modular artifact to a corresponding monolithic artifact. For example, a set of modular consistency relations may be considered correct with respect to a monolithic relation when it considers the same model tuples consistent. For three metamodels M_1, M_2, M_3 with three modular consistency relations $CR_{1,2}, CR_{1,3}, CR_{2,3}$ between them, as well as a ternary consistency relation $CR_{1,2,3}$, we could say that $CR_{1,2}, CR_{1,3}, CR_{2,3}$ are correct (with respect to $CR_{1,2,3}$) if, and only if,

$$\begin{aligned} &\forall\, m_1 \in M_1, m_2 \in M_2, m_3 \in M_3 : \big(\langle m_1, m_2, m_3\rangle \in CR_{1,2,3} \\ &\quad \Leftrightarrow \langle m_1, m_2\rangle \in CR_{1,2} \wedge \langle m_1, m_3\rangle \in CR_{1,3} \wedge \langle m_2, m_3\rangle \in CR_{2,3}\big) \end{aligned}$$

We may, analogously, define correctness for consistency preservation rules, an orchestration function, and an application function with respect to a monolithic preservation rule by defining that both deliver the same results for the same inputs or at least return a consistent result in the same cases.

4.2.3. Dimensions of Correctness

The discussed correctness notions induce two dimensions: First, correctness can be considered between artifacts within a monolithic or modular specification. Second, correctness can be considered between artifacts of a modular specification and corresponding artifacts of a monolithic specification. These dimensions are depicted in Figure 4.4. The former dimension is depicted vertically. Consistency preservation rules need to be correct with respect to their consistency relations. In the modular case, in addition to each preservation rule being *locally* correct with respect to its relation, the combination of preservation rules by an orchestration and application function must also be *globally* correct with respect to the combination of all relations. The latter dimension is depicted horizontally. Each modular artifact must be consistent with respect to a corresponding monolithic artifact.

Although correctness of modular with respect to monolithic artifacts can be interesting from a theoretical perspective, its practical relevance is limited.

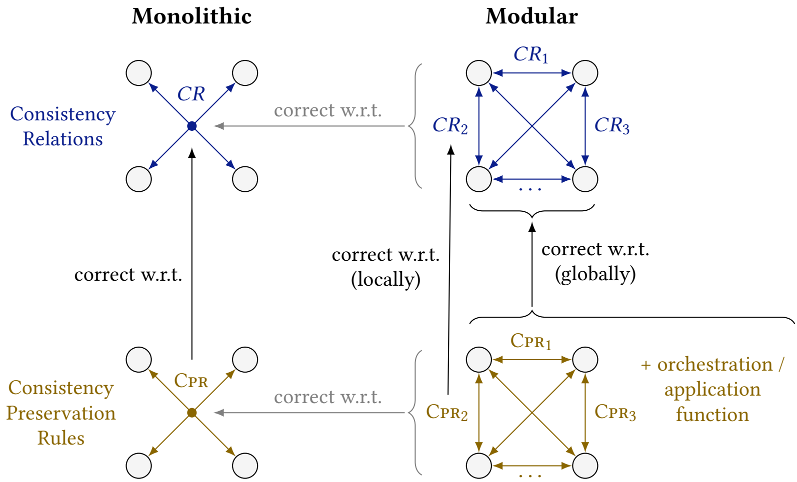


Figure 4.4.: Different notions of correctness for consistency and its preservation. Circles denote metamodels with arrows between them representing consistency relations and consistency preservation rules. Further unidirectional arrows denote different notions of correctness of one or more artifacts with respect to others.

That notion of correctness assumes that there is some kind of global truth that has to be reflected by a modular specification. This, however, has the following two essential drawbacks.

Validation Artifacts: The artifacts to validate correctness against, i.e., the global, monolithic consistency relation as well as an appropriate monolithic consistency preservation rule, do usually not exist. If they existed, they could directly be used to preserve consistency. Thus, it is impossible to validate a set of consistency relations and consistency preservation rules against such a global specification.

Modular Knowledge: This notion of correctness requires that the developers have some global knowledge that represents a monolithic consistency relation and its consistency preservation rule. We assume the knowledge about relations between models to usually be distributed across several persons. Thus, there will be no such global knowledge, and not even an implicit notion of the necessary artifacts to validate the modular specifications against exists.

Since this conflicts with our assumption of distributed knowledge about relations and independently developed, modular specifications, we do not further consider this notion of consistency. We focus on correctness between the artifacts of a modular consistency specification. We have discussed this correctness notion as correctness between a *modularization level* and a *global level* of consistency specifications in previous work [Kla+19b].

4.2.4. Correctness of Consistency Relations

The consistency notion that we consider in the following especially requires that consistency preservation rules and the functions to orchestrate and apply them must be correct with respect to consistency relations. This notion does, however, not define when consistency relations are considered *correct*. One option is to only consider correctness with respect to monolithic artifacts for the case of consistency relations, as we have proposed in previous work [Kla+19b]. This, however, suffers from the discussed drawback of requiring a global notion of consistency. Another notion of correctness would be conformance of the specified relations with what developers expect to be consistent, i.e., a validation of requirements. For example, a consistency relation between UML and Java may only be considered correct if it fulfills some "natural" notion of consistency, as developers know how elements are related because they represent similar things, such as classes, or because a standard like the UML [UML] prescribes it. In this work we do not consider such a correctness notion with respect to external, maybe not formally specified artifacts, as it is part of separate research on requirements engineering and validation.

In consequence, we might say that consistency relations are simply *correct by construction*. Thus, relations would normatively define what is to be considered consistent. However, a consequence of not assuming a global knowledge of consistency is that different domain experts may have different and even conflicting notions of when models are to be considered consistent. Consider for three metamodel M_1, M_2, M_3 the three modular consistency relations $CR_{1,2} = \{\langle m_1, m_2 \rangle\}$, $CR_{1,3} = \{\langle m_1, m_3 \rangle\}$, and $CR_{2,3} = \{\langle m_2, m_3' \rangle\}$. Then there is no triple of models that is considered consistent to all relations. Although we still do not want to assume a global knowledge about consistency to which the modular one must conform, we might say that these relations are *incompatible*, as we do not want to combine relations that induce an empty set of consistent model tuples. Identifying an appropriate notion of *compatibility* and how to check it constitutes **RQ 1.2** and will be discussed as our contribution **C 1.2** in Chapter 5.

In fact, every set of modular consistency relations induces a monolithic one. The monolithic relation CR for metamodels $M_1, \ldots, M_n$ and pairwise relations $CR_{i,k}$ is defined by:

$$CR = \{\langle m_1, \ldots, m_n \rangle \mid \bigwedge_{1 \leq i < k \leq n} \langle m_i, m_k \rangle \in CR_{i,k}\}$$

At least if this induced relation is empty, we probably want to consider the modular relations incompatible, because if no models are considered consistent, we cannot describe any system consistently.

4.3. A Formal Notion of Transformation Networks

We have so far discussed a general notion of consistency and its preservation with a focus on a modular way of specifying it. This notion was introduced in a rather informal way to first be able to discuss correctness notions and determine which notion is relevant for the considerations in this thesis. In the following, we define a formal notion of consistency and its preservation, based on the informal explanation given before. It extends the one we have presented in previous work [Kla+21]. We also give a precise definition of notions for correctness between the artifacts of a modular specification. Furthermore, we now focus on transformation-based approaches, i.e., we consider specifications that transform changes within one or more models into changes in one or more other models, as a specialization of the general notion for consistency preservation used before.

4.3.1. Modular Consistency Specification

As discussed informally before, an extensional specification of consistency defines a relation between models by enumerating all tuples of models that are considered consistent.

Definition 4.1 (Model-Level Consistency Relation)

Given a tuple of metamodels $\mathfrak{M} = \langle M_1, \ldots, M_n \rangle$, a *model-level consistency relation CR* is a relation for instances of the metamodels $CR \subseteq I_{\mathfrak{M}} = I_{M_1} \times \cdots \times I_{M_n}$.

For a tuple of models $\mathfrak{m} \in I_{\mathfrak{M}}$, we say that:

$$\mathfrak{m} \textit{ consistent to } CR :\Leftrightarrow \mathfrak{m} \in CR$$

Otherwise, we call $\mathfrak{m}$ *inconsistent to CR*.

We consider a tuple of models consistent if the consistency relation contains it. This conforms to existing consistency definitions for bidirectional transformations [Ste10]. We denote this kind of consistency relation as *model-level*, because we later need to refine the notion of consistency relations to the level of metaclasses and distinguish them.

If a single relation describes consistency between all relevant models, consistency is defined by means of model tuples being contained in that relation. We call such a relation *monolithic*. If a relation only defines consistency between some of the relevant models and the global consistency relation is defined by a combination of several such relations, we need an explicit definition of such a *modular* notion of consistency. For the sake of simplicity, we focus on *binary* relations as a modular representation of consistency.

Definition 4.2 (Model-Level Consistency)

Let $\mathfrak{M} = \langle M_1, \ldots, M_n \rangle$ be metamodels and let $CR_{i,k} \subseteq I_{M_i} \times I_{M_k}$ be a binary model-level consistency relation for $M_i, M_k \in \mathfrak{M}$. We say that a model tuple $\mathfrak{m} = \langle m_1, \ldots, m_n \rangle \in I_{\mathfrak{M}}$ is *consistent to* $CR_{i,k}$ if, and only if, the instances of M_i and M_k are in that relation:

$$\mathfrak{m} \textit{ consistent to } CR_{i,k} :\Leftrightarrow \langle m_i, m_k \rangle \in CR_{i,k}$$

For a set of binary model-level consistency relations $\mathbb{CR}$ for metamodels $\mathfrak{M}$, we say that a tuple of models $\mathfrak{m} \in I_{\mathfrak{M}}$ is *consistent to* $\mathbb{CR}$ if, and only if, it is consistent to each consistency relation in that set:

$$\mathfrak{m} \textit{ consistent to } \mathbb{CR} :\Leftrightarrow \forall CR \in \mathbb{CR} : \mathfrak{m} \textit{ consistent to } CR$$

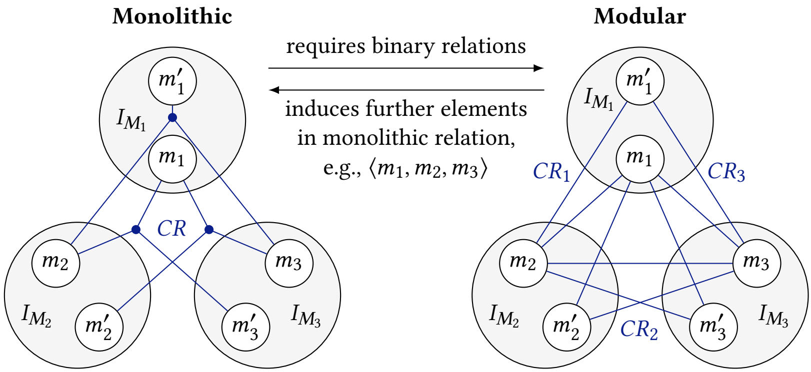


Figure 4.5.: A monolithic consistency relation that cannot be expressed by binary relations. Small circles denote models and blue, solid hyperedges relate tuples of consistent models.

The definition states that models are consistent to a set of model-level consistency relations if they are consistent to each relation in that set. Consider, for example, for $m_i \in I_{M_i}$ the relations $CR_1 = \{\langle m_1, m_2\rangle\}$, $CR_2 = \{\langle m_2, m_3\rangle\}$, and $CR_3 = \{\langle m_1, m_3\rangle\}$. Then the model tuple $\langle m_1, m_2, m_3\rangle$ is consistent to these relations. These consistency relations are equivalent to a monolithic relation $CR = \{\langle m_1, m_2, m_3\rangle\}$, because a model tuple $\mathfrak{m}$ is consistent to CR exactly when it is consistent to $\{CR_1, CR_2, CR_3\}$.

For reasons of simplicity, we assume only one consistency relation between each pair of metamodels. This also includes that there are no two consistency relations $CR_{i,j}$ and $CR_{j,i}$ for metamodels M_i and M_j, which means that the relations do not have a direction. This assumption is without loss of generality, because two relations between the same metamodels are, independent from their direction, equivalent to only considering their intersection, i.e., only the model pairs that are considered consistent by both relations.

Although in the preceding exemplary case the binary relations are equivalent to a monolithic relation, such an equivalence is not always given. In general, two interesting insights come along with the definition of consistency based on modular relations. First, expressiveness of defining consistency modularly by a set of relations is not equivalent to defining one monolithic relation. Second, a modular definition of consistency can easily contain contradictions, which can lead to an empty tuple of consistent models.

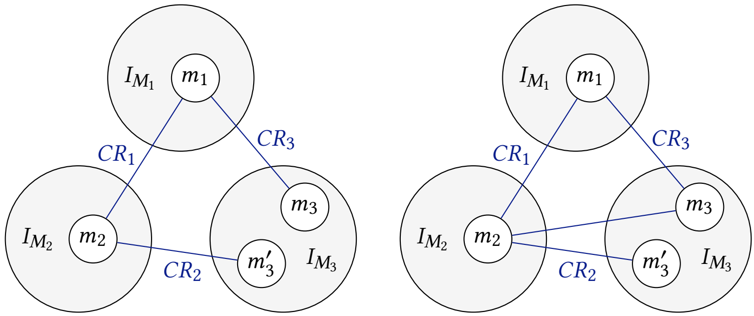


Figure 4.6.: Modular consistency relations, which together cannot be fulfilled (left) or which cannot be fulfilled for some of the consistent model pairs (right). Small circles denote models and (blue) lines relate consistent model pairs.

Obviously, combining binary relations has not the same expressiveness as defining a monolithic relation. For example, binary relations cannot express the monolithic relation $CR = \{\langle m_1, m_2, m_3' \rangle, \langle m_1, m_2', m_3 \rangle, \langle m_1', m_2, m_3 \rangle\}$, as depicted in Figure 4.5. The binary relations necessarily need to contain $\langle m_1, m_2 \rangle$ because $\langle m_1, m_2, m_3' \rangle \in CR$, $\langle m_1, m_3 \rangle$ because $\langle m_1, m_2', m_3 \rangle \in CR$, and $\langle m_2, m_3 \rangle$ because $\langle m_1', m_2, m_3 \rangle \in CR$. However, this would mean that $\langle m_1, m_2, m_3 \rangle$ is consistent to the binary relations although it is not consistent to the monolithic relation CR. Thus, using sets of binary relations in contrast to a single monolithic relation reduces expressiveness. Stevens [Ste20b] discusses the property of a multiary relation to be expressible by binary ones as *binary-definable* in detail. She proposes restrictions to binary relations that may be sufficient and still practical for expressing consistency, such as a notion of *binary-implemented* relations. We have reasoned the assumption that relations are specified independently and thus modularly, thus we have to accept these theoretic restrictions in expressiveness anyway.

Additionally, it can easily occur that multiple binary relations can be fulfilled by certain models, but no tuple of models exists that is consistent to all of them. Consider the relations $CR_1 = \{\langle m_1, m_2 \rangle\}$, $CR_2 = \{\langle m_2, m_3' \rangle\}$, and $CR_3 = \{\langle m_1, m_3 \rangle\}$, which are also depicted at the left of Figure 4.6. Although for each of these relations a consistent pair of models exists, which is exactly the one defined in each relation, no tuple of models exists that fulfills their combination. This example illustrates the worst case, in which no consistent models exist for a set of relations. In other cases, only for some models that are consistent according to one or some of the relations no model tuple may exist that is consistent to all relations. Consider the relations $CR_1 = \{\langle m_1, m_2\rangle\}$, $CR_2 = \{\langle m_2, m_3\rangle, \langle m_2, m_3'\rangle\}$, and $CR_3 = \{\langle m_1, m_3\rangle\}$, which are also depicted at the right of Figure 4.6. In this case, the tuple $\langle m_1, m_2, m_3\rangle$ is considered consistent to the relations, but although $\langle m_2, m_3'\rangle \in CR_2$ there exists no consistent model tuple containing m_3, i.e., there is no $m_1^* \in I_{M_1}$ such that $\langle m_1^*, m_2, m_3'\rangle$ is consistent to all relations.

It is easy to see that one monolithic relation can be equally represented by an arbitrary number of sets of binary relations by simply adding model pairs to these binary relations that are never consistent to the other relations, like we have seen for the pair $\langle m_2, m_3'\rangle$ in the previous example. This means that the combination of relations can lead to the situation that some models are actually forbidden (like m_3' in the example before) due to the combination of consistency relations. Whether such a situation is intended can eventually depend on the semantics of the models and relations, but we will discuss which situations are unintended in general. We have informally discussed this as a notion of *compatibility*, for which we investigate in Chapter 5 how far this behavior should be expected.

4.3.2. Incremental Consistency Preservation

While the previous discussion only concerned when models are considered consistent, it is of particular interest to ensure that consistency of models is preserved. We informally introduced such specifications as consistency preservation rules. In the following, we will restrict ourselves to *incremental* and *inductive* consistency preservation and give a precise definition for that. This means that we make the following assumptions to the process.

Information Preservation (Incrementality): After a change to one model, the others are not generated from scratch but updated according to the performed changes. This ensures that information that cannot be generated but was added by users to the other models is preserved.

Consistency Assumption (Induction): We assume models to be consistent before a change is processed by consistency preservation rules. Otherwise, the preservation rules would need to be able to handle arbitrary states of the models and intentions of performed changes could not be incorporated to restore consistency.

Incrementality is an essential requirement whenever consistency shall be preserved to avoid information loss. Otherwise, if for example Java code is always generated anew after changes to a UML model instead of adapting it incrementally, all implementations of methods in Java get lost every time the UML model is changed. Inductivity, on the other hand, may not be necessary, as consistency preservation rules could also be defined to restore consistency from arbitrarily inconsistent states. We, however, make this assumption to avoid requiring from the consistency preservation rules that they need to be able to process an inconsistent state without knowing which changes introduced it. From a theoretical point of view, we could omit that requirement, but this would make the specification of consistency preservation rules impractically complicated, such that omitting that requirement is not practically relevant anyway.

Like we have discussed for consistency preservation rules in general, incremental preservation rules can be realized in an either monolithic or modular way. A monolithic consistency preservation rule takes a tuple of models that is consistent to a consistency relation and a change to these models, and it returns a tuple of models that is consistent again. In a modular specification of consistency preservation rules, a set of such rules is given of which each preserves consistency of a subset of the given models according to a modular consistency relation. In our case, we consider such rules for two models, each of them restoring consistency according to a binary consistency relation.

In existing terminology for transformations, a consistency preservation rule that restores consistency of models according to a consistency relation in one direction is called *directional transformation* [Ste10] or *consistency restorer* [Ste20b]. That terminology usually considers model states instead of changes and defines a consistency preservation rule $\textsc{Cpr}$ for metamodels M_1 and M_2 to modify the instance of M_2 for restoring consistency as:

$$\textsc{Cpr} : I_{M_1} \times I_{M_2} \rightarrow I_{M_2}$$

This notion, however, has two properties that imply essential drawbacks:

State-Based: Information about the performed changes that led to the inconsistent state is missing. Thus the specification is not aware of how the inconsistent state was reached.

Unidirectional: The specification is unidirectional, which always requires to only update one model to restore consistency.

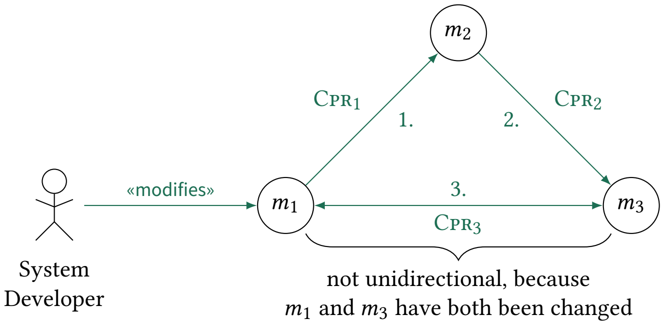


Figure 4.7.: Execution of consistency preservation rules of which at least one cannot be unidirectional, because both involved models (circles) have been modified by a user or other consistency preservation rules.

State-based transformations suffer from not knowing which changes were made that led to an inconsistent state, and reconstructing them from the difference between two states is only a heuristic approximation [Dis+11]. This, for example, includes that information about elements that were moved or renamed can potentially not be reconstructed, leading to elements that are deleted and created anew and losing all information that was potentially added to them. Unidirectionality may be reasonable when assuming that only one of the models was modified. In that case, it is sufficient to update the other model to restore consistency. With a modular specification of consistency preservation, however, several consistency preservation rules modifying the same models may need to be executed.

Figure 4.7 depicts an example in which unidirectional consistency preservation rules cannot be applied when used in combination with other such rules. If the depicted consistency preservation rules CPR$_1$ and CPR$_2$ are executed first, CPR$_3$ cannot be unidirectional, because both involved models m_1 and m_3 have been modified by either the user or another consistency preservation rule. Thus, it is, in general, not possible to only consider changes in one model and unidirectionally propagate them to the other model. In consequence, the preservation rules need to be able to deal with changes performed in both models and, consequentially, need to update both models to reflect the changes in each other.

To be able to combine several consistency preservation rules without the discussed drawbacks, we define a *synchronizing* rather than a unidirectional notion of them. Those rules can react to changes in both models and produce changes in both models again. This is sometimes also called the capability of handling *concurrent* modifications (e.g. [Leb+14]). To precisely define this behavior, we introduce a notion of *changes* and *consistency preservation rules*, which we also refer to as *synchronizing* consistency preservation rules.

As motivated before, we base our notion of consistency preservation on changes to explicitly express how an inconsistent state was derived from a previously consistent one. We consider these changes as functions that take a model and return a new one. They are not restricted to a specific model but defined for all instances of a metamodel, because a change is supposed to represent how specific elements are modified, such as adding, removing or modifying them. Thus, they can be applied to any models containing these affected elements. This is also how actual implementations, such as the one in the EMF behave. When elements affected by a change are not present in a model, applying the change may fail. For that reason, we consider the function describing a change to be partial. We denote partiality by returning $\perp$ for inputs the function is undefined for.

Definition 4.3 (Change)

Given a metamodel M, a change δ_M is a partial function that takes an instance of that metamodel and returns another one or $\perp$:

$$\delta_M : I_M \rightarrow I_M \cup \{\perp\}$$

We denote the identity change, i.e., the one always returning the input model, as δ_{id}:

$$\delta_{id}(x) := x$$

We denote the universe of all changes in M, i.e., all injective subsets of $I_M \times I_M$, as:

$$\Delta_M := \{\delta_M \subseteq I_M \times I_M \mid \forall \langle m_1, m_2 \rangle, \langle m_1, m_2' \rangle \in \delta_M : m_2 = m_2'\}$$

Definition 4.4 (Change Tuple)

For a given metamodel tuple $\mathfrak{M} = \langle M_1, \ldots, M_n \rangle$, we denote a tuple of changes to an instance of each metamodel as:

$$\delta_{\mathfrak{M}} = \langle \delta_{M_1}, \ldots, \delta_{M_n} \rangle \in \Delta_{M_1} \times \cdots \times \Delta_{M_n}$$

We define the universe of change tuples in instance tuples of $\mathfrak{M}$ as:

$$\Delta_{\mathfrak{M}} \coloneqq \Delta_{M_1} \times \cdots \times \Delta_{M_n}$$

We define the application of a change tuple $\delta_{\mathfrak{M}} = \langle \delta_{M_1}, \ldots, \delta_{M_n} \rangle$ to a model tuple $\mathfrak{m} = \langle m_1, \ldots, m_n \rangle \in I_{\mathfrak{M}}$ as the element-wise application:

$$\delta_{\mathfrak{M}}(\mathfrak{m}) \coloneqq \langle \delta_{M_1}(m_1), \ldots, \delta_{M_n}(m_n) \rangle$$

For us, it does not matter how the function behaves in cases in which the encoded change cannot be applied, e.g., because the changed or removed element does not exist. The function may do nothing, i.e., return the identical model, or even be undefined for those models, i.e., be partial and return $\perp$. In fact, we do not restrict the actual behavior of a change in any way. It may return an empty model regardless of the input, or it may perform arbitrary changes to different models instead of affecting only specific elements. Since we do not need such restrictions, they are not reflected in the formalism.

With that notion of changes, we can define consistency preservation rules as functions that receive two models and changes to them, and that return new changes to both models. While the general definition does not prescribe this, we assume the resulting changes to include the input changes such that not both of them have to be executed consecutively. This will also be reflected by a correctness notion for such rules.

Definition 4.5 (Consistency Preservation Rule)

Let $CR \subseteq I_{M_1} \times I_{M_2}$ be a binary model-level consistency relation between metamodels M_1 and M_2. A *consistency preservation rule* $\textsc{Cpr}_{CR}$ for the relation CR is a function:

$$\textsc{Cpr}_{CR} : (I_{M_1}, I_{M_2}, \Delta_{M_1}, \Delta_{M_2}) \rightarrow (\Delta_{M_1}, \Delta_{M_2}) \cup \{\perp\}$$

For reasons of practical applicability, the rules need to be partial, as we may not want to require them to be able to process arbitrary models and changes. Like for changes, we denote this partiality by allowing the function to return $\perp$. First, this is because we do not require it to produce changes when the input models were not consistent. Second, even if the input models are consistent, it may not be possible to preserve consistency for the given changes. For example, if conflicting changes in both changes are made, i.e., changes that require one of them to be reverted, it may be desired that the consistency preservation rule does not return an unexpected result but to indicate a failure by returning $\perp$. Our formalism does not restrict such a behavior, in fact it even allows to always return the same changes or to return changes that always deliver empty models. Finally, it is up to the developer to define reasonable consistency preservation rules and to define in which cases the function does not return a result.

This notion of synchronizing consistency preservation conforms to the definition of *synchronizers* given by Xiong et al. [Xio+13], which also reflect the case that both models have been modified and can be updated by the consistency preservation rule. They do, however, encode the changes in terms of new model states rather than explicit changes.

To consider a consistency preservation rule *correct*, it has to return changes that, when applied to the input models, result in models that are consistent according to the model-level consistency relation for which the preservation rule is defined. This conforms to the notion of correctness defined for bidirectional transformations [Ste10] and the notion of consistency given for synchronizers by Xiong et al. [Xio+13].

Definition 4.6 (Consistency Preservation Rule Correctness)

We call a consistency preservation rule $\textsc{Cpr}_{CR}$ *correct* if, and only if, it either returns $\perp$ or changes that applied to the input models yield models that are consistent to CR:

$$\begin{aligned}
&\textsc{Cpr}_{CR}\ \mathit{correct}\ :\Leftrightarrow \forall\, m_1 \in I_{M_1}, m_2 \in I_{M_2}, \delta_{M_1} \in \Delta_{M_1}, \delta_{M_2} \in \Delta_{M_2} :\\
&\quad \forall\, \delta'_{M_1} \in \Delta_{M_1}, \delta'_{M_2} \in \Delta_{M_2} : \big(\textsc{Cpr}_{CR}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) = (\delta'_{M_1}, \delta'_{M_2})\\
&\qquad \Rightarrow \langle \delta'_{M_1}(m_1), \delta'_{M_2}(m_2)\rangle\ \mathit{consistent\ to\ CR}\big)
\end{aligned}$$

This definition does not restrict how the input and output changes are related. In fact, a valid (and especially correct) consistency preservation rule could always return identity changes. In consequence, the rule would simply revert all input changes to achieve a consistent state. Although this may not be the expected behavior, there is no reason to restrict this behavior by definition. Actually, the developer should specify a preservation rule in a *reasonable* way, such that it defines an expected behavior.

We have discussed that consistency preservation rules can be derived from consistency relations and that consistency preservation rules can imply the consistency relations by their image, i.e., the set of all models that can be derived by applying the consistency preservation rule to any models and changes for which it is defined. In practice there will only be one of these specifications and the other is implied or derived. We thus define a *synchronizing transformation*, in extension to *bidirectional transformations* [Ste10], as an artifact that encapsulates a model-level consistency relation together with a consistency preservation rule, no matter which of them is defined and which is derived or implied.

Definition 4.7 (Synchronizing Transformation)

Let CR be a model-level consistency relation and $\textsc{Cpr}_{CR}$ a consistency preservation rule that restores consistency according to that relation. A *synchronizing transformation* is a pair $\mathfrak{t} = \langle CR, \textsc{Cpr}_{CR} \rangle$.

We also use the short term *transformation* for a synchronizing transformation. Correctness of a transformation is then given by correctness of its consistency preservation rule.

Definition 4.8 (Synchronizing Transformation Correctness)

Let $\mathfrak{t} = \langle CR, \textsc{Cpr}_{CR} \rangle$ be a synchronizing transformation. We say that $\mathfrak{t}$ is *correct* if, and only if, $\textsc{Cpr}_{CR}$ is correct according to Definition 4.6:

$$\mathfrak{t} \text{ } correct \text{ } :\Leftrightarrow \textsc{Cpr}_{CR} \text{ } correct$$

Transformations are usually expected to by *hippocratic* [Ste10]. This means that a transformation, or more precisely its consistency preservation rule, does not perform any changes if the input changes applied to the input models already yield consistent models. We define the application of hippocraticness to synchronizing transformations as follows.

Definition 4.9 (Hippocratic Synchronizing Transformation)
Let $\mathfrak{t} = \langle CR, \textsc{Cpr}_{CR} \rangle$ be a transformation for metamodels M_1 and M_2. We say that $\mathfrak{t}$ is *hippocratic* if, and only if, it returns the input changes if their application to the input models yields consistent models:

$$\begin{aligned}\mathfrak{t}\ \textit{hippocratic}\ &:\Leftrightarrow \forall\, m_1 \in I_{M_1}, m_2 \in I_{M_2}, \delta_{M_1} \in \Delta_{M_2}, \delta_{M_2} \in \Delta_{M_2} :\\ &\quad \big(\langle \delta_{M_1}(m_1), \delta_{M_2}(m_2)\rangle \in CR\\ &\qquad \Rightarrow \textsc{Cpr}_{CR}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) = (\delta_{M_1}, \delta_{M_2})\big)\end{aligned}$$

Although hippocraticness is not a necessary requirement for our considerations in most cases, it is usually a desired property in practice [Ste10]. One benefit of hippocraticness with regards to transformations is given if a transformation is only defined by its consistency preservation rule and thus implies the underlying consistency relation as its fixed points, as discussed in Subsection 4.1.4. Actually, a consistency preservation rule according to our definition does not have fixed points, because the signatures of definition and value set of the function are different due to the models only occurring in the definition set. Transferred to our definition, the consistency relation is implied by iteratively applying the function to each pair of models and changes with the changes delivered by the function until they are not modified by the function anymore. In case that the transformation is correct and hippocratic, it does always deliver changes that yield consistent models already upon its first execution and does not modify them upon further applications, thus the consistency relation is implied by applying the function to each pair of models and changes only once.

In the following, we only refer to transformations rather than consistency relations and consistency preservation rules if the distinction is not necessary. We thus also say that models are consistent to a transformation, which is supposed to mean that they are consistent to the consistency relation encapsulated by that transformation.

Definition 4.10 (Consistency to Transformation)

Let $\mathfrak{t} = \langle CR, \textsc{Cpr}_{CR} \rangle$ be a synchronizing transformation. We say that a tuple of models $\mathfrak{m}$ is *consistent to* $\mathfrak{t}$ if, and only if, it is consistent to its consistency relation:

$$\mathfrak{m} \textit{ consistent to } \mathfrak{t} :\Leftrightarrow \mathfrak{m} \textit{ consistent to } CR$$

For a set of transformations $\mathbb{t}$, we say that a model tuple $\mathfrak{m}$ is *consistent to* $\mathbb{t}$ if, and only if, it is consistent to all transformations in it:

$$\mathfrak{m} \textit{ consistent to } \mathbb{t} :\Leftrightarrow \forall \mathfrak{t} \in \mathbb{t} : \mathfrak{m} \textit{ consistent to } \mathfrak{t}$$

Although Definition 4.8 precisely defines correctness of a transformation, it is unclear how to define a transformation that fulfills that property. In particular, most existing transformation languages are restricted to input changes to one model or to delivering changes to one model. We thus discuss how we can achieve a correct synchronizing transformation with such a restricted formalism. This question was introduced as **RQ 1.3**, and an approach for that constitutes our contribution **C 1.3**, which we discuss in Chapter 6.

4.3.3. Transformation Orchestration

Preserving consistency between instances of multiple metamodels after changes with multiple transformations requires their orchestration, i.e., the decision in which order to execute them. We have discussed in Subsection 4.1.3 that transformations, or more precisely their consistency preservation rules, may be executed independently, which requires their results to be unified, or to execute them consecutively. We have identified the drawbacks of concurrent execution, including the necessity to define unification operators and the missing guarantee of consistency after unification. This is why we follow the approach of consecutively executing transformations.

To consecutively execute transformations, an execution order has to be determined. While in practice a dynamic algorithm will determine that order, from a theoretical perspective that algorithm realizes a function that returns the execution order. We call this an *orchestration function*, as it is responsible for orchestrating the transformation execution.

Definition 4.11 (Transformation Orchestration Function)

Let $\mathbb{t}$ be a set of transformations for metamodels $\mathfrak{M}$. A transformation orchestration function $\textsc{Orc}_{\mathbb{t}}$ for these transformations is a function that delivers a sequence of transformations for given models and changes:

$$\textsc{Orc}_{\mathbb{t}} : (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \rightarrow \mathbb{t}^{<\mathbb{N}}$$

$\mathbb{t}^{<\mathbb{N}}$ denotes all finite sequences in $\mathbb{t}$, i.e., $\mathbb{t}^{<\mathbb{N}} := \bigcup_{i=0}^{\infty} \mathbb{t}^i$

The orchestration function returns a sequence of transformations and determines that their consistency preservation rules need to be executed in the given order. This especially includes that transformations may occur more than once in such a sequence.

Without further restrictions to the transformations, an orchestration function may not always find an execution order that yields a consistent model tuple for given transformations, models, and changes to them. Such an order may not exist, because due to the transformations making local decisions to restore consistency for two models that are never consistent with the other transformations. Additionally, even if such an order exists, it may not be possible to find it. We discuss these problems in detail in Chapter 7 and prove that the decision problem whether an orchestration that leads to a consistent result exists is undecidable without further restrictions. For that reason, the definition does not require that an orchestration of transformations has to lead to a consistent result.

An orchestration function only determines an order of transformations. Consistency for given models and changes can be preserved by requesting an orchestration from that function and executing the transformations in that order. We make this process explicit by defining an *application function* that performs consistency preservation based on given transformations, an orchestration function for them and the actual models and changes.

Before defining that application function, we first need to define an auxiliary function to concatenate transformations, more precisely their contained consistency preservation rules. Consistency preservation rules according to Definition 4.5 are restricted to the two metamodels they are defined for. Additionally, they require initial models and changes as input, but only return changes. For these two reasons, the functions describing the preservation rules cannot be easily concatenated. This, however, is necessary to compose them to formally describe their consecutive execution. We define a *generalization function* for transformations, which generalizes them to arbitrary metamodel tuples and a conforming signature for their input and output, which eases the description of their concatenation.

Definition 4.12 (Transformation Generalization Function)

Let $\mathfrak{M} = \langle M_1, \ldots, M_i, \ldots, M_k, \ldots, M_n \rangle$ be a metamodel tuple and let $\mathfrak{t} = \langle CR, \textsc{Cpr}_{CR} \rangle$ be a transformation for metamodels M_i, M_k. A transformation generalization function $\textsc{Gen}_{\mathfrak{M},\mathfrak{t}}$ for metamodels $\mathfrak{M}$ and transformation $\mathfrak{t}$ is a partial function:

$$\textsc{Gen}_{\mathfrak{M},\mathfrak{t}} : (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \rightarrow (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \cup \{\bot\}$$

It generalizes the consistency preservation rule $\textsc{Cpr}_{CR}$ of $\mathfrak{t}$ such that it can be applied to changes in $\mathfrak{M}$ instead of M_i and M_k, i.e., it applies the changes delivered by $\textsc{Cpr}_{CR}$ for the corresponding models to the given change tuple. Let $\mathfrak{m} \in I_{\mathfrak{M}}$ be a model tuple and let $\delta_{\mathfrak{M}} = \langle \delta_{M_1}, \ldots, \delta_{M_i}, \ldots, \delta_{M_k}, \ldots, \delta_{M_n} \rangle$ be a change tuple. We define $\langle \delta'_{M_i}, \delta'_{M_k} \rangle := \textsc{Cpr}_{CR}(m_i, m_k, \delta_{M_i}, \delta_{M_k})$. Then we define:

$$\begin{aligned} &\textsc{Gen}_{\mathfrak{M},\mathfrak{t}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \\ &\quad := \begin{cases} \bot, & \textit{if } \langle \delta'_{M_i}, \delta'_{M_k} \rangle = \bot \\ (\mathfrak{m}, \langle \delta_{M_1}, \ldots, \delta'_{M_i}, \ldots, \delta'_{M_k}, \ldots, \delta_{M_n} \rangle), & \textit{otherwise} \end{cases} \end{aligned}$$

Like consistency preservation rules, a generalization function must be partial and return $\bot$ for inputs it is undefined for to reflect cases in which it cannot return a result. This is a direct consequence of consistency preservation rules being partial, thus a generalization function is defined to return $\bot$ in the same cases as the consistency preservation rule it generalizes. The generalization function is a universally-defined auxiliary function only necessary for formalizing the concepts. It must neither be specialized for each transformation, nor must a transformation developer specify it at all.

Finally, either the orchestration function or an application function must be able to reflect the cases in which no execution order of transformations that restores consistency can be found. In accordance to existing terminology [Ste20b], we call these cases *unresolvable*. From a theoretical perspective, it does not matter whether the orchestration or application function makes that decision, as the orchestration function could even be encoded into the application function. From a practical perspective, however, we may want to determine an execution order even if there is no order that results in a consistent state. This supports finding out why no such order is found, e.g., which transformation induces that problem.

We define a transformation application function that applies transformations to a given tuple of models and changes according to an order delivered by an orchestration function. This function is partial to allow it to indicate that no result with consistent models could be found, e.g., because the input models were inconsistent or because a transformation within the orchestration delivered $\bot$. We indicate those cases with the result $\bot$.

Definition 4.13 (Transformation Application Function)

Let $\mathbb{t}$ be a synchronizing transformations set for consistency relations $\mathbb{CR}$ on metamodels $\mathfrak{M}$ and $\textsc{Orc}_{\mathbb{t}}$ an orchestration function. A transformation application function $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ for them is a partial function:

$$\textsc{App}_{\textsc{Orc}_{\mathbb{t}}} : (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \rightarrow I_{\mathfrak{M}} \cup \{\bot\}$$

The function takes a consistent tuple of models and a tuple of changes that was performed on them and returns a changed tuple of models by acquiring changes from the consistency preservation rules of $\mathbb{t}$. Thus, it has to fulfill the following condition:

$$\begin{aligned}
&\forall\, \mathfrak{m} \in I_{\mathfrak{M}} \mid \mathfrak{m} \textit{ consistent to } \mathbb{CR} : \forall\, \delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \\
&\quad \forall\, \mathfrak{m}' \in I_{\mathfrak{M}} : \big[\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = \mathfrak{m}' \Rightarrow \\
&\qquad \exists\, \mathfrak{t}_1, \ldots, \mathfrak{t}_n \in \mathbb{t} : \exists\, \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \big(\textsc{Orc}_{\mathbb{t}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = [\mathfrak{t}_1, \ldots, \mathfrak{t}_n] \\
&\qquad \wedge \textsc{Gen}_{\mathfrak{M},\mathfrak{t}_n} \circ \ldots \circ \textsc{Gen}_{\mathfrak{M},\mathfrak{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}}) \wedge \delta'_{\mathfrak{M}}(\mathfrak{m}) = \mathfrak{m}'\big)\big]
\end{aligned}$$

While the previous definition does not restrict in which cases $\bot$ and in which an actual tuple of models is returned, we define when we consider an application function *correct*. Correctness can be defined in several ways. For example, we might say that the function is correct if it returns a consistent tuple of models whenever there is an order of transformations that leads to those consistent models. As we will see later, this correctness notion is, however, inappropriate, because the underlying decision problem is undecidable. In consequence, the application function needs to operate conservatively, i.e., it may return $\bot$ even if there is a sequence of transformations whose application leads to consistent models. As an alternative, we might require the function to return consistent models whenever the orchestration function delivers a sequence of transformations whose application leads to a consistent tuple of models. Since we have to deal with conservativeness anyway, this, however, does not provide any benefits. In fact, the above discussed requirements encode a kind of *optimality* for the functions, which we will specify more precisely in Chapter 7. For now, we stick to the simple notion of correctness that the application function does never return inconsistent models, i.e., if a tuple of models is returned, it must be consistent.

Definition 4.14 (Transformation Application Function Correctness)
Let $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ be an application function for an orchestration function $\textsc{Orc}_{\mathbb{t}}$ for transformations $\mathbb{t}$. Let $\mathbb{CR}$ be the set of consistency relations of transformations in $\mathbb{t}$. We say that $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ is *correct* if, and only if, its result is either $\bot$ or consistent to $\mathbb{CR}$:

$$\begin{aligned}\textsc{App}_{\textsc{Orc}_{\mathbb{t}}} \textit{ correct } &:\Leftrightarrow \forall\, \mathfrak{m} \in I_{\mathfrak{M}} \mid \mathfrak{m} \textit{ consistent to } \mathbb{CR} : \forall\, \delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \\ &\big(\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = \bot \vee \textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \textit{ consistent to } \mathbb{CR}\big)\end{aligned}$$

This is, in fact, a rather weak notion of correctness. An application function that always returns $\bot$ is correct according to that definition. Because the orchestration and application function have to operate conservatively, a binary correctness notion is less relevant than a gradual one anyway. The question how to determine such an orchestration was introduced as **RQ 1.4**. We present and discuss a concrete approach as contribution **C 1.4** in Chapter 7.

4.3.4. Transformation Networks

Based on the previous definitions of transformations, orchestration and application functions, we define what we consider a *transformation network* and when we consider it *correct*. A transformation network is composed of transformations, an orchestration and an application function. Although we define these artifacts specifically for one transformation network, i.e., an orchestration and application function according to their definitions are specific for one set of transformations, the goal will be to find an orchestration and application function that is independent from the actual transformations.

Definition 4.15 (Transformation Network)

Let $\mathbb{t}$ be a transformation set, $\textsc{Orc}_{\mathbb{t}}$ an orchestration function for these transformations, and $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ an application function. A transformation network $\mathfrak{N}$ is a triple:

$$\mathfrak{N} := \langle \mathbb{t}, \textsc{Orc}_{\mathbb{t}}, \textsc{App}_{\textsc{Orc}_{\mathbb{t}}} \rangle$$

Correctness of a transformation network is given by correctness of the individual transformations and the application function, according to Definition 4.8 and Definition 4.14.

Definition 4.16 (Transformation Network Correctness)

Let $\mathfrak{N} = \langle \mathbb{t}, \textsc{Orc}_{\mathbb{t}}, \textsc{App}_{\textsc{Orc}_{\mathbb{t}}} \rangle$ be a transformation network. We say that $\mathfrak{N}$ is *correct* if, and only if, its transformations in $\mathbb{t}$ and the application function $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ are correct:

$$\mathfrak{N} \textit{ correct } :\Leftrightarrow \forall\, \mathfrak{t} \in \mathbb{t} : \mathfrak{t} \textit{ correct } \wedge \textsc{App}_{\textsc{Orc}_{\mathbb{t}}} \textit{ correct}$$

We have already discussed that we will show that the application function has to operate conservatively, which is why correctness is an essential property but not the most interesting one to achieve. Additionally, we discussed that the consistency relations of the transformations can be considered correct by definition, but that we will discuss a notion of *compatibility* to reflect when those relations contain unintended contradictions.

4.4. A Fine-Grained Notion of Consistency

We have up to now given a common definition of consistency [Ste10] by enumerating consistent pairs of models in a relation. That notion is sufficient for defining transformation networks, correctness of their artifacts, and also the essential considerations regarding orchestration, as presented in the preceding section. Domain experts and transformation developers, however, usually think in terms of a more fine-grained notion of consistency. They do not consider when complete models are consistent, but when specific relations between some of their elements are fulfilled, i.e., which other elements they require to exist if some elements are present in models. For example, they consider consistency between architectural components and object-oriented classes instead of complete models containing these elements.

This is also reflected by transformation languages, such as QVT-R. First, they require relations to be defined at the level of classes and their properties. They define how properties of some classes are related to properties of other classes. Second, they are defined in an *intensional* way, i.e., constraints specify which elements are consistent rather than enumerating all consistent instances in an *extensional* specification. We have already discussed that intensional and extensional specifications have equal expressiveness and can be transformed into each other, which is why we stick to extensional specifications for reasons of simplicity. However, we reuse the concept of specifying relations at the level of classes and their properties.

This reflects a natural understanding of consistency and, in particular, makes it easier to make statements about dependencies between consistency relations, which we need to make statements about compatibility of consistency relations. Thus, we introduce an appropriate, fine-grained notion of consistency relations in the following. Finally, from such a fine-grained specification, a model-level consistency relation can always be derived by enumerating all models that fulfill all the fine-grained specifications, thus it does not restrict expressiveness in any way and can be seen as a *compositional approach* for defining consistency, which is only a refinement of the notion of model-level consistency relations. We have presented the following definitions of a fine-grained consistency notion, partly literally, in previous work [Kla+20]. The definitions are based on those proposed in the work of Kramer [Kra17, Sec. 2.3.2, 4.1.1] and Klare et al. [Kla+21].

4.4.1. Fine-Grained Consistency Relations

The central idea of the fine-grained consistency notion is to have consistency relations that contain pairs of objects and, broadly speaking, requires that if the objects in one side of the pair occur in a model, the others have to occur in another model as well. A *condition* encapsulates such objects, for which we require objects in another model to occur.

Definition 4.17 (Condition)
A condition $\mathbb{c}$ for a class tuple $\mathfrak{C}_{\mathbb{c}} = \langle C_{\mathbb{c},1}, \ldots, C_{\mathbb{c},n} \rangle$ is a set of object tuples with:

$$\forall \langle o_1, \ldots, o_n \rangle \in \mathbb{c} : \forall\, i \in \{1, \ldots, n\} : o_i \in I_{C_{\mathbb{c},i}}$$

An element $\mathfrak{c} \in \mathbb{c}$ is called a *condition element.* For a model tuple $\mathfrak{m} \in I_{\mathfrak{M}}$ of a metamodel tuple $\mathfrak{M}$ and a condition element $\mathfrak{c}$, we say that:

$$\mathfrak{m} \textit{ contains } \mathfrak{c} :\Leftrightarrow \exists\, m \in \mathfrak{m} : \mathfrak{c} \subseteq m$$

Conditions represent object tuples, called *condition elements*, that instantiate the same tuple of classes. They are supposed to occur in models that fulfill a certain condition regarding consistency and thus require elements in other models to exist, as subsequently defined by consistency relations. We say that a tuple of models contains a condition element if any of the models contains all the objects within the condition element. This implies that such a model's metamodel has to contain all the classes in the class tuple of the condition. We use conditions to define consistency relations as the co-occurrence of condition elements.

Definition 4.18 (Consistency Relation)
Let $\mathfrak{C}_{l,CR}$ and $\mathfrak{C}_{r,CR}$ be two class tuples. A consistency relation CR is a subset of pairs of condition elements in conditions $\mathbb{c}_{l,CR}, \mathbb{c}_{r,CR}$ with $\mathfrak{C}_{l,CR} = \mathfrak{C}_{\mathbb{c}_{l,CR}}$ and $\mathfrak{C}_{r,CR} = \mathfrak{C}_{\mathbb{c}_{r,CR}}$:

$$CR \subseteq \mathbb{c}_{l,CR} \times \mathbb{c}_{r,CR}$$

We call a pair of condition elements $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR$ a *consistency relation pair.* For a model tuple $\mathfrak{m}$ and a consistency relation pair $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle$, we say that:

$$\mathfrak{m} \textit{ contains } \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle :\Leftrightarrow \mathfrak{m} \textit{ contains } \mathfrak{c}_l \wedge \mathfrak{m} \textit{ contains } \mathfrak{c}_r$$

A consistency relation is a set of pairs of condition elements, which indicate the tuples of objects that are considered consistent with each other. This means that if a model contains one of the left condition elements that occurs in the relation, another model must contain one of the related right condition elements. It bases on two conditions that define relevant object tuples in instances of each of the two metamodels and defines the ones that are related to each other. Without loss of generality, we assume that each condition element of both conditions occurs in at least one consistency relation pair:

$$\begin{aligned}
&\forall \mathfrak{c} \in \mathbb{C}_{l,CR} : \exists \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR : \mathfrak{c} = \mathfrak{c}_l \\
&\land \forall \mathfrak{c} \in \mathbb{C}_{r,CR} : \exists \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR : \mathfrak{c} = \mathfrak{c}_r
\end{aligned}$$

Based on these consistency relations, we can define a fine-grained notion of consistency.

Definition 4.19 (Consistency)

Let CR be a consistency relation and let $\mathfrak{m} \in I_{\mathfrak{M}}$ be a tuple of models of the metamodels in $\mathfrak{M}$. We say that:

$$\begin{aligned}
&\mathfrak{m} \textit{ consistent to } CR :\Leftrightarrow \\
&\quad \exists W \subseteq CR : \big[\forall \langle \mathfrak{c}_{l,1}, \mathfrak{c}_{r,1} \rangle, \langle \mathfrak{c}_{l,2}, \mathfrak{c}_{r,2} \rangle \in W : \\
&\qquad\quad \big(\langle \mathfrak{c}_{l,1}, \mathfrak{c}_{r,1} \rangle = \langle \mathfrak{c}_{l,2}, \mathfrak{c}_{r,2} \rangle \lor (\mathfrak{c}_{l,1} \neq \mathfrak{c}_{l,2} \land \mathfrak{c}_{r,1} \neq \mathfrak{c}_{r,2}) \big) \\
&\qquad \land \forall \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in W : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}_l \land \mathfrak{m} \textit{ contains } \mathfrak{c}_r \big) \\
&\qquad \land \forall \mathfrak{c}'_l \in \mathbb{C}_{l,CR} : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}'_l \Rightarrow \mathfrak{c}'_l \in \mathbb{C}_{l,W} \big) \big]
\end{aligned}$$

We call such a W a *witness structure* for consistency of $\mathfrak{m}$ to CR, and for all pairs $\langle \mathfrak{w}_l, \mathfrak{w}_r \rangle \in W$, we call $\mathfrak{w}_l$ and $\mathfrak{w}_r$ *corresponding to* each other.

For a set of consistency relations $\mathbb{CR} = \{CR_1, CR_2, \ldots\}$, we say that:

$$\mathfrak{m} \textit{ consistent to } \mathbb{CR} :\Leftrightarrow \forall CR \in \mathbb{CR} : \mathfrak{m} \textit{ consistent to } CR$$

A consistency relation CR relates one condition element at the left side to one or more other condition elements at the right side of the relation. The definition of consistency ensures that if one condition element $\mathfrak{c} \in \mathbb{C}_{l,CR}$ at the left side of the relation occurs in a tuple of models, exactly one of the condition elements related to it by a consistency relation CR occurs in

Figure 4.8.: A consistency relation derived from Figure 3.3, which depicts the necessity of a witness structure to ensure that only one employee out of those with differently capitalized names is allowed to correspond to a resident with the same name.

another model to consider the tuple of models consistent. If another element that is related to $\mathfrak{c}$ occurs in the models, this one has to be, in turn, related to another condition element $\mathfrak{c}' \in \mathbb{C}_{l,CR}$ of the left side of condition elements by CR that also occurs in the models. This ensures that a condition element contained in a model uniquely corresponds to another element to which it is considered consistent according to CR.

Consider the exemplary consistency relation in Figure 4.8, which is derived from the one in our running example in Figure 3.3. The relation requires for each resident an employee with an appropriate name to exist and vice versa. It assumes that resident names are stored lowercase and allows the employee name to be written in arbitrary capitalization. Thus, for example, both the employees with names "Alice" and "alice" would be considered consistent to a resident with name "alice". Without the restriction defined by the auxiliary witness structure W, an employee model containing the employees with both capitalizations would be considered consistent to a resident model containing a corresponding resident with the same name written in lowercase. The witness structure, however, ensures that for each employee one corresponding resident exists, thus there can only exist one employee with one of the allowed capitalizations, as each of them is corresponding to the resident with the lowercase name. In general, the witness structure restriction ensures that if several alternatives for a corresponding element exists, only one is actually allowed to be present.

Example 4.1. *The definition of consistency is exemplified in Figure 4.9, which is an alternation of an extract of Figure 3.3 only considering employees and residents. Models with employees and residents are considered consistent if for each employee exactly one resident with the same name or the name in lowercase exists. The model pairs* 1–3 *are obviously consistent according to the definition, because there is always a pair of objects that fulfills the consistency relation. In model pair* 4*, there is a consistent resident for each employee, but there is no appropriate employee for the resident with name = "Bob". However,*

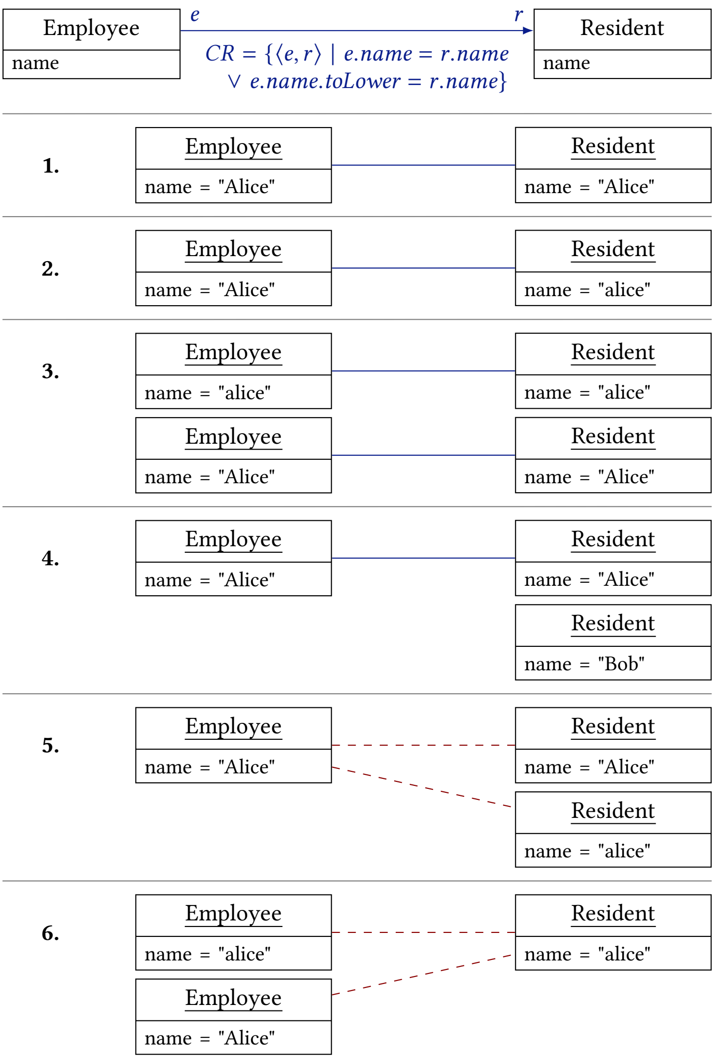


Figure 4.9.: A consistency relation between employee and resident and six example model pairs: pairs 1–4 consistent with an appropriate witness structure W shown in blue, solid lines, and pairs 5 and 6 inconsistent with an inappropriate mapping structure shown in red, dashed lines. Adapted from [Kla+20, Fig. 2].

our definition of consistency only requires that for each condition element at the left side of the relation that appears in the models, an appropriate right element occurs, but not vice versa. Thus, a relation is interpreted unidirectionally, which we subsequently discuss in more detail. In model pair 5*, there are two residents with names in different capitalizations, which would both be considered consistent to the employee according to the consistency relation. Comparably, in model pair* 6*, there is a resident that fulfills the consistency relations for both employees, each having a different but matching capitalization. However, the consistency definition requires that each model element for which consistency is defined by a consistency relation must only have one corresponding element. In this case, there are two residents or employees that could be considered consistent to the employee or resident, respectively, thus there is no witness structure with a unique mapping between the elements as required by the definition.*

As mentioned in the example, the definition considers consistency in a unidirectional way, which means that a consistency relation may define that some elements $\mathfrak{c}_r$ are required to occur in a tuple of models if some elements $\mathfrak{c}_l$ occur, but not vice versa. Such a unidirectional notion can also be reasonable in our example, as it could make sense to require a resident for each employee, but not every resident might be employed and thus also represent an employee. To achieve a bijective consistency definition, for each consistency relation CR its transposed relation $CR^T = \{\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \mid \langle \mathfrak{c}_r, \mathfrak{c}_l \rangle \in CR\}$ can be considered as well. Regarding Figure 4.9, if we consider the relation between employees and residents as well as its transposed, the model pair 4 would also be considered inconsistent, because an appropriate employee for each resident is required by the transposed relation. We call sets of consistency relations that contain only bijective definitions of consistency *symmetric*.

Definition 4.20 (Symmetric Consistency Relation Set)
Let $\mathbb{CR}$ be a set of consistency relations. We say that $\mathbb{CR}$ is *symmetric* if, and only if, for each contained relation its transposed one is also contained:

$$\mathbb{CR} \textit{ is symmetric } :\Leftrightarrow \forall CR \in \mathbb{CR} : CR^T \in \mathbb{CR}$$

Any description of bijective consistency relations can be defined with a symmetric consistency relation set. We have defined consistency in a unidirectional way for the two following reasons.

1. Some relevant consistency relations are actually not bijective. Apart from the simple example concerning residents and employees, this situation always occurs when objects at different levels of abstraction are related. Consider a relation between components and classes, requiring for each component an implementation class but not vice versa, or a relation between UML models and object-oriented code, requiring for each UML class an appropriate class in code but not vice versa. These relations could not be expressed if consistency relations were always considered bijective.
2. We consider networks of consistency relations, in which a combination of multiple bijective consistency relations does not necessarily imply a bijective consistency relation again. Thus, we need a unidirectional notion of consistency relations anyway.

One might argue that consistency is usually traced by means of a *trace model*, which stores the pairs of element tuples in models that fulfill a consistency relation. A trace model can be seen as an explicit representation of a witness structure as specified in Definition 4.19. We do, however, not explicitly consider such an explicit trace model in this formalism for two reasons also discussed in previous work [Kla+21]. First, a trace model is only necessary in practice if no identifying information for related elements is present, or if performance is to be improved. However, we assume such identifying information to exist without loss of generality, as introduced in Subsection 3.3.3. Second, a trace model can, from a theoretical perspective, be treated as a usual model by defining consistency between one concrete and one trace model. This conforms to the fact that each multiary relation can be expressed by binary relations to an additional model (in this case the trace model), as discussed in existing work [Ste20b; Cle+19]. We discuss practical benefits of having an explicit trace model for consistency preservation in Chapter 6 to distinguish modifications of elements from their removal and addition. But this does, as discussed, not restrict applicability of our formalism.

4.4.2. Expressiveness of Fine-Grained Relations

The model-level consistency notion of Definition 4.2 is established and based on notions used by several researchers. The fine-grained consistency notion according to Definition 4.19 is based on the insight that practical approaches to describe consistency and its preservation use fine-grained rules rather than enumerating consistent model pairs. We did, however, only provide examples that justify specific decisions in the definitions, such as the witness structure for corresponding elements, but we did not argue if and why fine-grained relations are an actual refinement, such that statements about model-level consistency relations also apply to fine-grained relations.

To show that every set of fine-grained consistency relations can be expressed by a single model-level consistency relation, we can use the same constructive approach that we have used to define consistency according to multiple consistency relations, be they at the model level or fine-grained. Given fine-grained consistency relations $\mathbb{CR} = \{CR_1, \dots, CR_k\}$, we can construct an equivalent model-level consistency relation CR as follows:

$$CR = \{\mathfrak{m} \mid \mathfrak{m} \textit{ consistent to } \mathbb{CR}\}$$

A model-level consistency relation can, however, not necessarily be expressed by fine-grained consistency relations. The most simple construction approach would define a single fine-grained consistency relation to express a model-level consistency relation, which contains the complete models instead of extracts of them. The definition of consistency is, however, different for the two types of relations. While at the model level consistency is defined as two (or more) models being in a relation (see Definition 4.2), fine-grained consistency relations do only describe that if an element at the left side of the relation occurs in a model, then any of the related elements at the right side has to occur in another. If two models are considered consistent by a model-level consistency relation, they are also consistent to the accordingly constructed fine-grained relation, because there is a witness structure that contains exactly the two consistent models. If there is a model that is not considered consistent to any other model in the model-level consistency relation, thus the model-level consistency relation does not contain any pair with that model, then there will also be no such pair in the fine-grained relation. According to Definition 4.19 of consistency for fine-grained relations, if there is no condition element in the relation, then consistency is not constrained for the contained model elements. In consequence, such a model would be considered consistent to every other model.

While, at first, this may seem inappropriate, it actually is appropriate for two reasons. First, the formalism can only express that for some elements other elements need to exist, but not that specific elements are not allowed to exist if other elements exist. This is reasonable, because consistency between models is supposed to ensure that the overlap of information is represented uniformly, thus to express that information in one model needs to be represented in another one as well. Expressing that some elements are not allowed to exist because of others, e.g., being an employee in one model, the same person cannot be a student in another model, is actually not a consistency constraint for information shared between models. This is actually additional information that should be stored in a specific model representing these semantics. Thus, we do not consider this case at all.

Second, the formalism for fine-grained consistency relations can not prevent specific elements from existing at all. For example, a consistency relation may define that for a component in an architecture model a corresponding class in the object-oriented design model has to exist, but it may not restrict that only components of specific names are allowed. Such restrictions should and actually are separate specifications not related to consistency between models but restricting a model on its own. Thus, the metamodel or some additional specification for it should provide such restrictions of valid models, which we have discussed as a restriction of I_M for a metamodel M in Section 3.3.

Summarizing, we found that we can express each set of fine-grained consistency relations by a model-level consistency relation. Additionally, we know that there are specific kinds of restrictions that can be encoded in model-level consistency relations but not in fine-grained consistency relations. We have, however, discussed why they are not relevant for the designated application area of consistency preservation. In consequence, all insights made for model-level consistency relations can also be applied to fine-grained consistency relations and, if specific restrictions are excluded, vice versa.

4.4.3. Application to Consistency Preservation Rules

As mentioned before, the fine-grained notion of consistency fits well to how transformation languages consider consistency. They allow to define rules that relate only some classes by relations, conforming to fine-grained consistency relations, from which fine-grained consistency preservation rules are derived. Alternatively, they directly allow to define rules to preserve consistency between specific classes. These rules are often called *transformation rules* and composed to a transformation that consists of multiple such rules, each encoding a consistency relation and a preservation rule.

It may easily happen that the execution of one transformation rule leads to the violation of the consistency relation of another, which induces dependencies between the individual transformation rules. Thus, a combination of transformation rules to a transformation has to ensure correctness, i.e., that the consecutive execution of the rules leads to a consistent state of the models. Languages such as QVT-R and QVT-O therefore specify that transformation rules may not be conflicting [QVT, Sec. 7.10.2.]. It is also a dedicated topic of research to ensure that the rules of a single transformation conform to each other, e.g. [CGL17; Cab+10], which is why we assume that transformations fulfill that property.

To avoid the necessity of specifying this conformance property for transformation rules, we stick to the existing notion of coarse-grained consistency preservation rules, as it is sufficient for our considerations. Still, consistency preservation rules were defined for model-level consistency relations in Definition 4.5. This can, however, be easily extended to fine-grained consistency relations, as we simply need to require the rule to consider consistency to a set of fine-grained relations according to Definition 4.19 rather than consistency to a single model-level consistency relation according to Definition 4.2.

A consistency preservation rule $\textsc{Cpr}_{\mathbb{CR}}$ for a set of consistency relations $\mathbb{CR}$ according to Definition 4.18 is thus still considered correct if it only returns changes when they yield models that are consistent to all consistency relations if applied to the input models, in accordance with Definition 4.6:

$$\begin{aligned}
&\forall\, m_1 \in I_{M_1}, m_2 \in I_{M_2}, \delta_{M_1} \in \Delta_{M_1}, \delta_{M_2} \in \Delta_{M_2} :\\
&\quad \forall\, \delta'_{M_1} \in \Delta_{M_1}, \delta'_{M_2} \in \Delta_{M_2} : \big(\textsc{Cpr}_{\mathbb{CR}}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) = \langle \delta'_{M_1}, \delta'_{M_2} \rangle\\
&\qquad \Rightarrow \langle \delta'_{M_1}(m_1), \delta'_{M_2}(m_2) \rangle \textit{ consistent to } \mathbb{CR}\big)
\end{aligned}$$

Note that being consistent to all fine-grained consistency relations is equivalent to being consistent to the model-level consistency relation induced by the fine-grained relations.

Likewise, we consider a synchronizing transformation according to Definition 4.7 as a pair of fine-grained consistency relations and a consistency preservation rule for them, thus $\mathfrak{t} = \langle \mathbb{CR}, \textsc{Cpr}_{\mathbb{CR}} \rangle$. Again, in conformance with Definition 4.8, we call such a transformation $\mathfrak{t}$ correct if, and only if, its consistency preservation rule is correct.

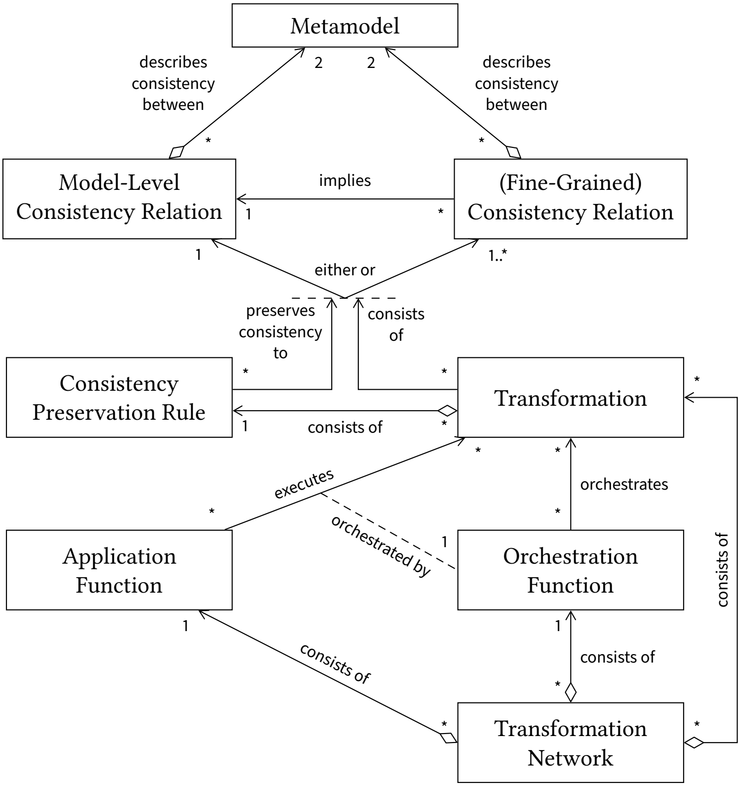


Figure 4.10.: A conceptual model for the terms and artifacts introduced for transformation networks and their relations. Adapted from [Kla+21, Fig. 5].

4.5. Summary

In this chapter, we have discussed notions of correctness for transformation networks and the artifacts they consist of, and we have precisely defined the notion that is relevant for the context of this thesis. We give an overview of the introduced concepts and their relations in the conceptual model in Figure 4.10. In summary, we provided the following insight in this chapter.

Insight II.1 (Correctness Notion)

A reasonable notion of correctness for networks of modular, independently developed transformations consists of correctness of the single transformations, which need to be synchronizing, and correctness of the application function that determines an execution order of the transformations. An application function may not be able to return a result for different reasons, such as transformations not being applicable to specific changes, the absence of an execution order of the transformations that leads to consistent models, or the inability to find such an order. Thus, in comparison to correctness, the degree of conservativeness is the more important property of an application function, which indicates how often the function does not deliver a result although there is an order of transformations that would restore consistency. Additionally, although theoretically not relevant for correctness, the relations defining when models are considered consistent must fulfill some notion of compatibility to be useful, as they can otherwise prevent transformations from finding consistent models.

In the following chapters, we thus define a notion of compatibility for consistency relations, discuss how correctness of the individual synchronizing transformations for achieving local consistency can be achieved, and finally how a correct and appropriate application function to perform the orchestration for achieving global consistency can be defined. In summary, these following contributions together allow to develop what we defined as a *correct* transformation network.

For visualizing examples of consistency relations, consistency preservation rules, and their execution throughout the next chapters, we use a notation according to the example depicted in Figure 4.11. We visualize consistency relations with blue arrows and a definition of the conditions for consistency relation pairs forming that relation. In the example, the consistency relation contains all pairs of employees and residents having the same name, except for those with an empty name. The arrows of such a relation indicate whether we only consider a directional consistency relation or also its transposed one. We depict consistency preservation rules with orange arrows and denote which changes it produces because of which input change. In the example, we denote that the addition of an employee ($+e$) leads to the addition of a resident with the same name, specified by the according property assignment

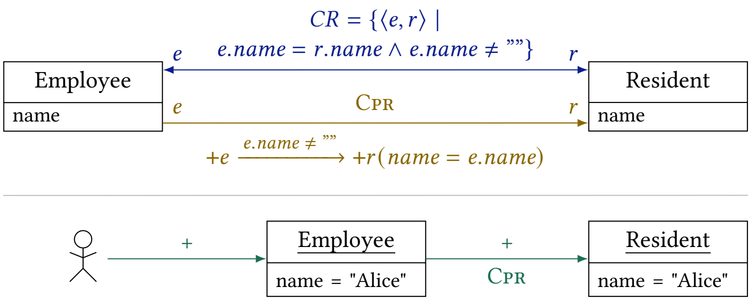


Figure 4.11.: Example for the visualization of consistency relations, consistency preservation rules, and the execution of changes by users or consistency preservation.

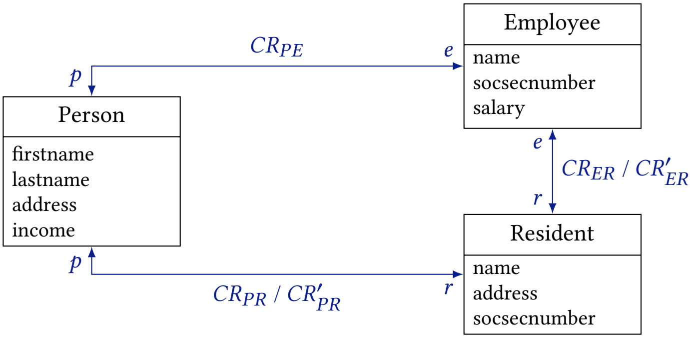
$r(name = e.name)$. In addition, we annotate conditions to the consistency preservation rules, such as $e.name \neq ""$ in the example, which restricts the resident creation to the case in which the employee name is not empty. We usually specify only parts of a consistency preservation rule if the other cases are not relevant in the specific context. In the example, we only specify the behavior for the case of adding an element but not of modifying or removing it. Finally, we denote the execution of any changes, including consistency preservation rules, with green arrows. In the example, we visualize the addition of an employee by a user, denoted with a "+", which leads to the addition of a resident because of the execution of the above introduced consistency preservation rule.

5. Proving Compatibility of Consistency Relations

Transformations, from which we construct transformation networks, are composed of consistency relations and consistency preservation rules that preserve them, as we have defined in Chapter 4. We focus on binary relations and according preservation rules, which relate two metamodels. While we have precisely defined correctness of transformations and their orchestration in a network, we found that the underlying consistency relations themselves can, from a theoretical perspective, be considered correct by construction, as there is no other artifact (be it explicit or only given implicitly) with respect to which it has to be correct. Since we assume transformations to be developed independently and reused in a modular way, we can especially not assume a monolithic consistency relation to which the modular consistency relations must be correct (see Subsection 4.2.3).

We have, however, already given examples for cases in which binary consistency relations are somehow contradictory. This is the case if the developers of individual transformations have different, conflicting notions of consistency between the metamodels. In the worst case, this can lead to the situation that no single tuple of models would be considered consistent to a set of binary consistency relations, which is obviously unwanted behavior. We have discussed an abstract example for that case already in Subsection 4.2.4.

We recapture the running example defined in Figure 3.3 and extend it with alternatives for two of the binary consistency relations in Figure 5.1. The example contains three pairwise consistency relations between persons, employees and residents. They are defined in a way such that none of them can be omitted, because each pair shares a unique overlap in their attributes. In the example, the consistency relations CR_{PE}, CR_{PR} and CR_{ER} (as well as their transposed ones) are fulfilled if for each person (and each employee and resident analogously) in the models exactly one employee and one resident



$$CR_{PE} = \{\langle p, e\rangle \mid p.\mathit{firstname} + \text{"}\textvisiblespace\text{"} + p.\mathit{lastname} = e.\mathit{name} \wedge p.\mathit{income} = e.\mathit{salary}\}$$

$$CR_{PR} = \{\langle p, r\rangle \mid p.\mathit{firstname} + \text{"}\textvisiblespace\text{"} + p.\mathit{lastname} = r.\mathit{name} \wedge p.\mathit{address} = r.\mathit{address}\}$$

$$CR'_{PR} = \{\langle p, r\rangle \mid p.\mathit{lastname} + \text{",}\textvisiblespace\text{"} + p.\mathit{firstname} = r.\mathit{name} \wedge p.\mathit{address} = r.\mathit{address}\}$$

$$CR_{ER} = \{\langle e, r\rangle \mid e.\mathit{name} = r.\mathit{name} \wedge e.\mathit{socsecnumber} = r.\mathit{socsecnumber}\}$$

$$CR'_{ER} = \{\langle e, r\rangle \mid e.\mathit{name}.\mathit{toLower} = r.\mathit{name} \wedge e.\mathit{socsecnumber} = r.\mathit{socsecnumber}\}$$

Figure 5.1.: Derivation of Figure 3.3: Three simple metamodels for persons, employees and residents, and three binary relations $CR_{PE}, CR_{PR}, CR_{ER}$ for each pair of them, with CR'_{PR} as an alternative for CR_{PR} and CR'_{ER} as an alternative for CR_{ER}. Adapted from [Kla+20, Fig. 1].

exist that fulfill the defined relations for names and other attributes. According to our notion of consistency relations (Definition 4.18), it is essential that always only one such corresponding element exists. Intuitively, these consistency relations are *compatible*, as they lead to a reasonable set of model tuples that are considered consistent.

In contrast, considering CR'_{PR} instead of CR_{PR}, the relations can never be fulfilled, because the concatenation of *firstname* and *lastname* from person to employee conflicts with the one from person to resident. The relation between employees and persons assumes *firstname* and *lastname* in that order, whereas the relation between residents and persons assumes them to be concatenated vice versa and to be separated by a comma. Fulfilling these relations would require an infinitely large model, as the cycle of the relations requires for each person, employee, and resident the existence of the others with *firstname* and *lastname* swapped and extended with a comma.

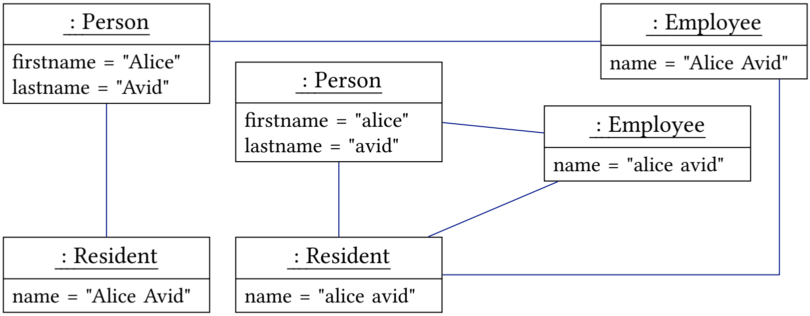


Figure 5.2.: Elements required by the consistency relations CR_{PE}, CR_{PR} and CR'_{ER} (as well as their transposed) in Figure 5.1 for a resident with the name "Alice Avid". Solid blue lines connect corresponding elements, which do not form a valid witness structure.

As finite models cannot fulfill this, the set of consistent model tuples would be empty.

In addition, considering consistency relation CR'_{ER} instead of CR_{ER}, no models containing residents with a *name* not written in lowercase can be consistent to all relations, as depicted in the example in Figure 5.2, which, for reasons of simplicity, omits all other attributes than the names. A resident with a non-lowercase *name* requires a person with equally capitalized *firstname* and *lastname* to exist. This requires an employee with an equally capitalized *name* to exist. The relation CR'_{ER} now requires a resident with the *name* written in lowercase to exist, which, again, requires a person with the lowercase *name* to exist. This, in turn, requires an employee with the lowercase *name* to exist as well. In consequence, the resident with the lowercase *name* would correspond to both the employee with the original and the lowercase *name*, whereas the resident with the original *name* does not correspond to any employee. Since there is no witness structure with a unique mapping of corresponding elements, as also reflected in Figure 4.9, such models cannot be consistent to the consistency relations. More intuitively speaking, it is impossible to find an employee that fulfills the consistency relation CR'_{ER} for a resident with a non-lowercase *name*. This is what we call and later precisely define as an *incompatibility* of the consistency relations, as they define constraints that cannot be fulfilled at the same time. This can always occur if there is a cycle in the graph induced by the consistency relations.

Such incompatibilities are unwanted, as they indicate that developers have different, contradictory notions of consistency. Additionally, they can easily result in transformations that do not yield consistent models or for which finding an orchestration that yields consistent models becomes unnecessarily difficult. For that reason, we first discuss scenarios to identify an intuitive notion of *compatibility*, which we then transfer into a formal notion. Afterwards, we develop a formal, inductive approach to prove compatibility of relations, for which we prove correctness. It is based on the insight that consistency relations forming a specific kind of tree structure are compatible and that removing a specific kind of redundant relations preserves compatibility. We then derive a practical approach for the transformation language QVT-R. This chapter thus constitutes our contribution **C 1.2**, which consists of four subordinate contributions: a discussion of compatibility notions; a formal definition of one such notion; a formal approach to prove compatibility; and finally a practical realization of that approach. It answers the following research question:

RQ 1.2: When are the constraints induced by transformations contradictory and how can that be analyzed?

We will see that it is, in general, not possible to prove that transformations are incompatible if the language, in which the relations are described, is undecidable, such as QVT-R. We can, however, at least conservatively validate compatibility of transformations. Thus, if our approach proves compatibility, the transformations are actually compatible, but not vice versa. This enables transformation developers to validate compatibility of their transformations on-the-fly during transformation development, if developed for a specific scenario, or a posteriori during their combination, according to the scenarios introduced in Section 3.2. In particular, in the first scenario developers can immediately react to the introduction of incompatibilities during transformation development.

We have published central contributions of this chapter, in particular the formal and the practical approach for validating compatibility, in previous work [Kla+20]. Parts of some sections of this chapter are also literally taken from that publication, which we further indicate in the respective sections. The practical approach has been developed in the Master's thesis of Pepin [Pep19], which was supervised by the author of this thesis.

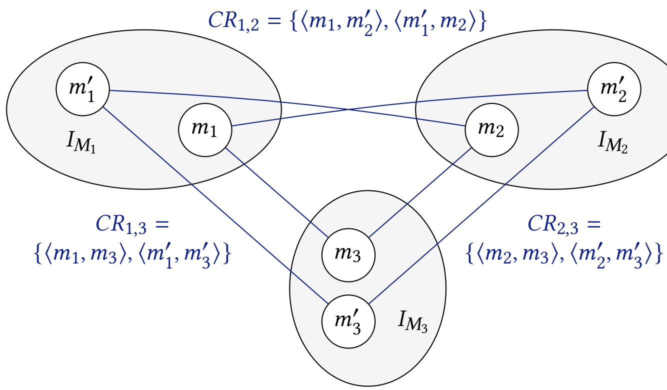


Figure 5.3.: Example for consistency relations that imply an empty global relation. Small circles denote models, and solid, blue lines relate consistent models.

5.1. Towards a Notion of Compatibility

We start with general considerations on model-level consistency relations, no matter whether they are specified explicitly or implied by sets of fine-grained consistency relations. A set of binary model-level consistency relations induces a monolithic, multiary relation, also called *global relation*, as discussed in Subsection 4.2.4. A monolithic relation CR for metamodels $M_1, \ldots, M_n$ and pairwise consistency relations $CR_{i,k}$ is defined by:

$$CR = \{\langle m_1, \ldots, m_n \rangle \mid \bigwedge_{1 \leq i < k \leq n} \langle m_i, m_k \rangle \in CR_{i,k}\}$$

As discussed before, the consistency relations are correct by definition and so is the induced global relation, even if it is empty. It is, however, unclear whether the relations are reasonable in combination.

In fact, if the relations induce an empty global relation, these relations do actually not properly fit to each other, because no single tuple of models would be considered consistent, thus no system could be consistently described. One may thus consider such relations incompatible. Figure 5.3 shows an extended version of the example already given in Subsection 4.2.4, which induces an empty global relation. This is an abstraction of the concrete examples that we have already discussed for our our running example, in which modified consistency relations lead to an empty set of consistent model tuples due to conflicting concatenations of names between persons, residents and employees.

There may, however, be more cases than empty induced global relations that we want to exclude by considering the relations incompatible. In general, the goal of finding incompatibilities and excluding them is twofold: First, we want to identify if different developers of modular relations have an incompatible notion of consistency, such that the results of preserving consistency would never be as expected. This is what we have seen in the examples with the name relations. We want to exclude these cases, because developers will not want to combine transformations based on relations that are contradicting. Second, incompatibilities may lead to transformations not being able to find consistent models, so the orchestration would not be able to execute transformations in an order that achieves a consistent state. If we, for example, encoded the relations from the running example with the inverse concatenation of *firstname* and *lastname* (CR'_{PR}) into transformations, each cycle in which the transformations are executed would produce one new person, employee, and resident, or it would change each of the existing ones, such that *firstname* and *lastname* are swapped and a comma is appended to *lastname*. In consequence, transformations would not be able to find a consistent state and, if not stopped preemptively, be executed endlessly. Thus, we also want to exclude such cases, because they can prevent the execution of transformations in a transformation network from terminating.

5.1.1. Necessity of Obsolete Relation Elements

A first intuitive option to define incompatibility is the presence of model pairs in the consistency relations, for which no globally consistent model tuple containing them can be found. This canonically covers the case in which the modular relations induce an empty global relation, because for none of the model pairs in each relation a globally consistent model tuple containing them can be found. An example for this case is depicted in Figure 5.4, in which the relation $CR_{1,2}$ contains the pairs $\langle m_1, m'_2 \rangle$ and $\langle m'_1, m_2 \rangle$, for which neither m_3 nor m'_3 is consistent to both other consistency relations, as the induced global relation is $CR = \{\langle m_1, m_2, m_3 \rangle, \langle m'_1, m'_2, m'_3 \rangle\}$. Thus, these model pairs may be denoted *obsolete* as they cannot occur in any globally consistent model tuple.

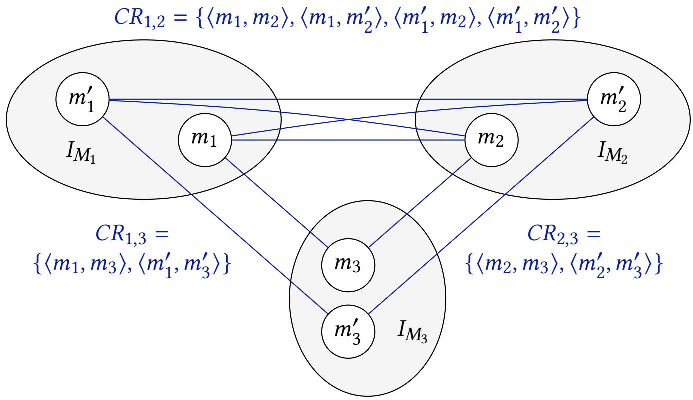


Figure 5.4.: Example for obsolete model pairs in consistency relation $CR_{1,2}$, which can never occur in a globally consistent model tuple. Small circles denote models, and solid, blue lines relate consistent models.

While this point of view may be reasonable when considering only the consistency relations, as we are finally just interested in results that are globally consistent, it induces problems to the process of achieving such a result by means of the execution of transformations or, more precisely, their consistency preservation rules. In fact, transformation networks need to allow intermediate states of models that are only locally consistent, although they can never occur in a globally consistent state. This is necessary, because otherwise each transformation would have to consider which model pairs are not only locally consistent but can be globally consistent as well. We, however, excluded such an alignment of the transformations by assumption of independent development and modular reuse and instead let the orchestration of transformations negotiate a consistent result.

Consider the following example, which is also exemplarily depicted in Figure 5.5. A UML class model and Java code are considered consistent when the same classes and interfaces with the same methods (in Java potentially with an empty body) are contained. Declaring the methods in a class when they are already declared in an implemented interface is optional in the UML. Then for each UML model a usually infinite number of consistent Java models exists, containing arbitrary implementations of the methods. PCM models and UML class models are consistent when components are realized as classes implementing the provided interfaces of the components. In this

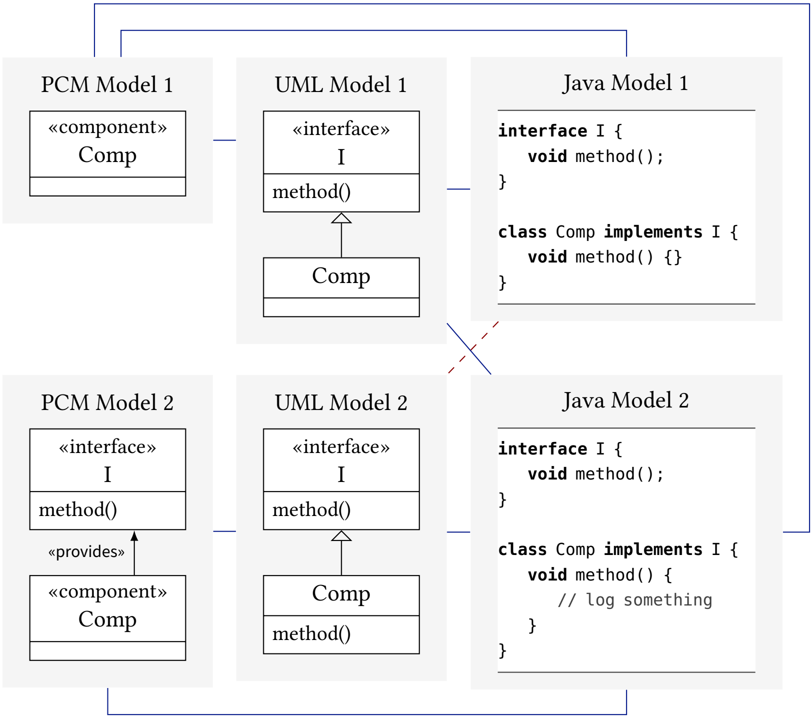


Figure 5.5.: Example for an obsolete model pair in consistency relations between PCM, UML and Java: The Java model with the empty method is locally consistent to the UML class model specifying the interface method also in the component implementation class. But these two models can never be globally consistent, because for the PCM component providing the interface, the consistency relation requires at least a default implementation of the method. Lines relate consistent models, whereof models related by dashed, red lines are never globally consistent.

case, the classes are required to declare the methods of provided interfaces again. Every class with "Comp" in its name is considered a component. Analogously, each component is represented by a Java class implementing the provided interfaces. The consistency relation between PCM and Java may, however, require that a method within a class that realizes a method of a provided interface of a component contains at least some default implementation, be it logging or something more component-specific. If we considered model pairs that can never occur in globally consistent model tuples as incompatible and thus forbid them, a UML model could not be considered consistent to a Java model if any method in a class with "Comp" in its name that is declared in one of its interfaces is realized by a Java method with an empty body. The transformation between UML and Java would thus not be allowed to create an empty Java method upon creation of such a UML method. This would, however, enforce the relation between UML and Java to encode information about components, which both breaks our assumption of independent development, as the developer of the transformation between UML and Java would need to know about components, and of modular reuse, because the transformation is then tied to the scenario in which the PCM is used as well.

In consequence of the given scenario and the according insight that transformations may need to produce transient states that are only locally consistent to ensure independence of the transformations and their reusability in different contexts, such obsolete consistency relations do not induce a proper notion of incompatibility.

5.1.2. Prevention from Finding Consistent Solutions

To identify a proper notion of incompatibility, we consider an exemplary transformation scenario from which we can derive such a notion. In the example depicted in Figure 5.6, we start with the models m_1, m_2 and m_3, which are consistent to all three defined consistency relations. If a user performs a change of m_2 to m_2', one possible execution of transformations can be as follows: The transformation for $CR_{2,3}$ changes m_3 to m_3', the one for $CR_{1,3}$ changes m_1 to m_1' and then the one for $CR_{1,2}$ changes m_2' back to m_2, as that is the only model consistent to m_1'. Now the transformation for $CR_{2,3}$ changes m_3' back to m_3, and finally the one for $CR_{1,3}$ restores m_1. As a result, the determined execution order yields the initial models before the user change, which are actually consistent but reject the user change.

Apart from the three given models, only m_1'', m_2', and m_3' are consistent. Upon the user change of m_2 to m_2', we would expect the transformations to find these models as a consistent result, as otherwise, like in the exemplary execution, the original models are returned, which actually rejects the user change. The issue results from model m_1' being present in the consistency relations but not being consistent in any globally consistent model tuple.

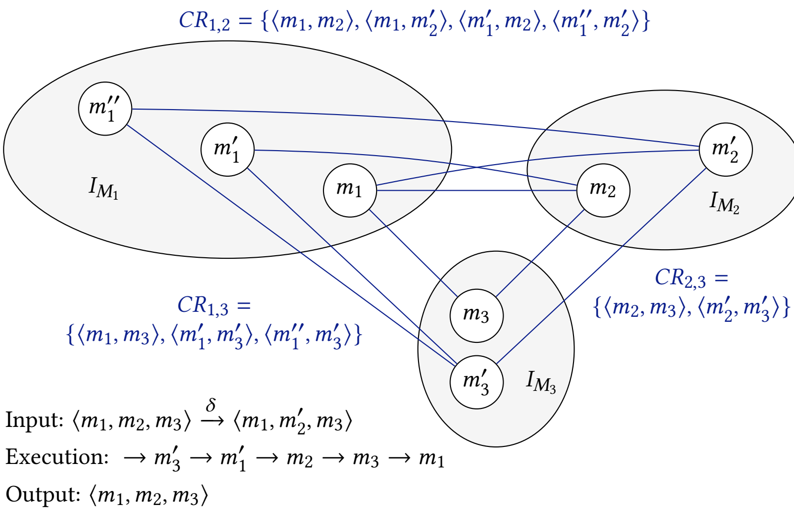


Figure 5.6.: Example for rejecting a user change because of consistency relations containing model pairs that are never globally consistent. Small circles denote models, and solid, blue lines relate consistent models.

Nevertheless, the selection of m_1' is valid and appropriate for each transformation locally, as there are models to which it is locally consistent according to each consistency relation on its own.

Note that this scenario is different from the case discussed for obsolete relation elements. In the scenarios discussed for obsolete relation elements, each model in such an obsolete pair occurs in a globally consistent model tuple but not both models in that pair together do. For example, the Java class with an empty method body actually occurs in a globally consistent model tuple but not together with the UML class model in which the method is declared in the class, although they are locally consistent.

We have seen that it is problematic when consistency relations define consistency of models that do not occur in any globally consistent model tuple. This can easily lead to transformations that do not find expected solutions and unnecessarily reject user changes. We did not define a requirement that user changes may not be reverted on purpose, as that behavior may also be expected to express that certain changes are not allowed to be made. However, if there was a reasonable sequence of transformations that returns a consistent tuple of models reflecting the user changes, it should be preferred over one that reverts the user change.

5.1.3. An Informal Notion of Compatibility

The discussed case that models do not occur in any globally consistent model tuple can be seen as a special case of obsolete relation elements, because it actually means that for each pair in a consistency relation in which a model occurs, the model pair cannot occur in a globally consistent model. In consequence, we found that in a combination of relations a model is problematic if

1. it is locally consistent to another model, i.e., it occurs in a consistency relation pair, and
2. it can never be globally consistent, i.e., it is not contained in any model tuple that is consistent to all consistency relations.

The model m_1' in Figure 5.6 is such a model, as it is locally consistent to m_2 and m_3', but those two are inconsistent. We can distinguish the following two cases that lead to the occurrence of such a model like m_1'.

User: The model was created by the user, thus adapting the model is unwanted as the user introduced it. Such a change should be rejected as the model cannot be globally consistent.

Transformation: The model was created by a transformation. In our example, this can either be the case because m_2 or m_3' was created. There is, however, at least m_1'' to which m_2 and m_3' are consistent as well, so the transformation should better select that one. If there was no such m_1'', then m_2 and m_3' would also not be in any globally consistent model tuple, thus the argumentation could be applied inductively.

In consequence, allowing such models during the process of describing a system and preserving consistency between the system models does not provide any benefits and thus should, in the best case, not occur. There is no reason to create such models, but it may prevent transformations from finding consistent states. In fact, disallowing the adaptation of the user change is even more reasonable when not concerning the complete model, like proposed with *authoritative models* by Stevens [Ste20b], but only the part considered by a specific rule that describes consistency. This can, for example, be a rule specifying the relation between classes and components rather than between the complete metamodels of the PCM and the UML. This is one of the reasons why we provided the formalization of fine-grained consistency relations in Definition 4.18 that relate extracts of models rather than complete ones. We use this fine-grained notion for formalizing and analyzing compatibility.

Transferred to our fine-grained notion of consistency relations, we consider consistency relations incompatible if there is a condition element (rather than a model) which does not occur in any tuple of models that is globally consistent to all consistency relations. We can thus formulate the following, for now informal notion of compatibility:

> For every condition element occurring in a consistency relation pair, a globally consistent model tuple containing it must exist.

This notion is especially reasonable when we consider the process of preserving consistency after user changes. We want to ensure that if consistency of the elements modified by the user is restricted by a consistency relation, there should at least be one consistent tuple of models that reflects the user change, i.e., contains the condition element he or she introduced or modified. If this is not the case, the transformations will not be able to produce a reasonable result, apart from reverting or adapting the user change.

Note that this notion of compatibility does only exclude combinations of relations according to the above made argumentation of being generally useless and potentially preventing transformations from finding consistency result. This does, however, not exclude further useless or unintended combinations of relations, for which the semantics of the relations would have to be known and analyzed. The already discussed example of the necessity to infinitely swap *firstname* and *lastname* and append a comma induced by the alternative consistency relation CR'_{PR} in Figure 5.1 leads to the situation that no tuple of models can fulfill those constraints, thus the global induced consistency relation is empty. If we, however, relax CR'_{PR} such that only *firstname* and *lastname* are swapped but no comma is appended, the relations can be fulfilled by models that contain each person twice, once with *firstname* and *lastname* assigned properly and once with them swapped. Although we might say that the relations are not intended that way, it is impossible for a generic approach to validate that without knowing about the semantics of the attributes *firstname*, *lastname* and their combination in *name*. In a different context, it may be desired that two attributes are concatenated in both orders, thus we cannot disallow that case in general.

Obviously, the given notion of compatibility is a property of a set of consistency relations and not of a single consistency relation. We may say that compatibility of a single relation is *context-dependent*. In consequence, that property can neither be analyzed nor systematically achieved for a single consistency relation. We can, by definition, not provide a construction approach for consistency relations to be compatible in each context. Compatibility can only be achieved by construction if all consistency relations to be used together are known and developed together, such that compatibility can be analyzed on-the-fly.

5.1.4. An Analysis for Compatibility of Relations

In the following sections, we define a formal notion of compatibility and derive a formal as well as a practical approach for analyzing or, more precisely, proving it. To give a first overview of this approach, we briefly introduce the central idea based on the informal notion of compatibility, which we first introduced in previous works [Kla18; Kla+19b].

We have seen that incompatibilities can arise whenever there are cycles in the graph induced by consistency relations. This means that the same models are related across two paths of relations, which may be contradictory. Thus, to avoid incompatibilities by construction, one could define a network of transformations and thus underlying consistency relations that does not contain any cycles. This situation is given when the network forms a tree. As we have already discussed, it is, however, in general not possible to define such a tree. First, it contradicts our assumption of independent development, as transformations would need to be aligned such that the missing direct relations between metamodels are expressed across other paths. Second, like we have seen in the running example in Figure 5.1, if three metamodels all share specific information only pairwise, there needs to be a cycle of transformations to keep that information consistent.

Even if we cannot construct a tree, we can use the insight that trees of transformations consist of inherently compatible consistency relations to analyze arbitrary topologies for compatibility. This bases on the following two techniques.

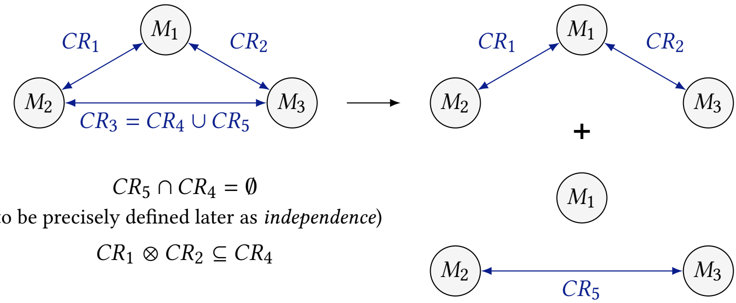


Figure 5.7.: Example for the decomposition of independent and removal of redundant consistency relations for analyzing compatibility. Adapted from [Kla18, Fig. 4].

Redundancy: If a consistency relation is redundant in a network, i.e., the same model tuples are considered consistent with or without that specific relation, we can remove it without affecting compatibility of the relations. More precisely, CR_1 is redundant in $\{CR_1, CR_2, CR_3\}$ if, and only if, a model tuple $\langle m_1, m_2, m_3 \rangle$ is consistent to $\{CR_1, CR_2, CR_3\}$ exactly when it is consistent to $\{CR_2, CR_3\}$. Iteratively identifying redundant relations and removing them until the remaining network is a tree, which is inherently compatible, we inductively know that the network with the redundant relations is compatible as well.

Independence: A second compatibility-preserving property of fine-grained consistency relations is independence. For example, if consistency between components and classes between PCM, UML and Java is expressed in one set of relations and consistency between different interface representations in another, they can be considered independently, because modifications in components and classes do not affect interfaces and vice versa. Proving compatibility for each independent set of consistency relations inductively proves compatibility of the union of all sets.

Finding independent subsets of relations and removing their redundancies until only trees remain proves compatibility. We call this approach *decomposition*, as we decompose the original relations into independent, essential relations, and we say that the resulting trees *witness* compatibility.

Figure 5.7 sketches the ideas for proving compatibility based on the given informal notion. We consider consistency relations CR_1, CR_2, and CR_3 between three metamodels, for which we know that CR_3 can be separated into disjoint CR_4 and CR_5, i.e., the relations are independent. Thus, one relation may relate components and classes and the other may relate different interface representations, as exemplarily explained before. Additionally, we know that the combination of CR_1 and CR_2 is a subset of CR_4, thus CR_4 is redundant as models are only considered consistent if they are consistent to CR_1 and CR_2 anyway. In other words, $CR_1 \otimes CR_2$ is more restrictive regarding consistency than CR_4. In consequence, we can, for the scope of the analysis, remove CR_4 and consider CR_1 and CR_2 independently from CR_5. This results in two independent trees of relations, which are inherently compatible. Since redundancy and independence are compatibility-preserving, this proves compatibility of the original relations.

5.2. A Formal Notion of Compatibility

In this section, we precisely define our up to now informally introduced notion of *compatibility*. For that, we use the fine-grained notion of consistency and its defining relations as proposed in Section 4.4. We discuss implicit relations, which are induced by a set of consistency relations, such as transitive relations, and, finally, derive a compatibility notion from the consistency formalization and its pursued perception. The contents of this and the remaining sections of this chapter are mostly, in parts literally, taken from our published article on proving compatibility [Kla+20].

5.2.1. Implicit Consistency Relations

Considering sets of consistency relations, as they are implicitly defined by the set of transformations in a transformation network, their combination is of especial interest. Each set of consistency relations defines relations between two sets of classes but also implies further *transitive* consistency relations. Having one relation between classes A and B and one between B and C implies an additional relation between A and C. We define a notion for the concatenation of relations that implies such transitive relations, which are supposed to reflect the consistency constraints introduced by the concatenated relations. Models should always be consistent to a concatenation of consistency relations if they are consistent to each of the concatenated relations, as otherwise the concatenation would introduce additional constraints. To achieve this, the following definition makes appropriate restrictions to the derived consistency relation pairs.

Definition 5.1 (Consistency Relations Concatenation)
Let CR_1 and CR_2 be consistency relations. We define their concatenation $CR_1 \otimes CR_2$ as:

$$\begin{aligned}
CR_1 \otimes CR_2 \coloneqq \{ & \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \mid \\
& \exists \langle \mathfrak{c}_l, \mathfrak{c}_{r,1} \rangle \in CR_1 : \exists \langle \mathfrak{c}_{l,2}, \mathfrak{c}_r \rangle \in CR_2 : \mathfrak{c}_{l,2} \subseteq \mathfrak{c}_{r,1} \\
& \wedge \forall \langle \mathfrak{c}_l, \mathfrak{c}'_{r,1} \rangle \in CR_1 : \exists \langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle \in CR_2 : \mathfrak{c}'_{l,2} \subseteq \mathfrak{c}'_{r,1} \}
\end{aligned}$$

with $\mathfrak{C}_{l,CR} = \mathfrak{C}_{l,CR_1}$ and $\mathfrak{C}_{r,CR} = \mathfrak{C}_{r,CR_2}$

The concatenation of two consistency relations contains pairs of object tuples that are related across common elements in the right respectively left side of the consistency relation pairs. Such a concatenation may be empty. Two requirements ensure that all models considered consistent to the concatenation are also consistent to the single relations: First, two consistency relation pairs of CR_1 and CR_2 are only combined if the left condition element of the consistency relation pair of CR_2 is a subset of the right condition element of the consistency relation pair of CR_1. Only in that case the existence of the right condition element of the pair of CR_1 in a model requires the existence of an according condition element in CR_2. Second, it is necessary that for all elements $\mathfrak{c}'_{r,1}$ in the right side of CR_1, which are considered consistent to a condition element $\mathfrak{c}_l$, there must be a matching condition element, i.e., a subset of $\mathfrak{c}'_{r,1}$, in the left condition of CR_2. If there was an element $\mathfrak{c}'_{r,1}$ in the right side of CR_1 for which the left-side condition of CR_2 does not contain a subset, the concatenation does not constrain consistency for the existence of $\mathfrak{c}_l$. Thus, without these restrictions the occurrence of $\mathfrak{c}_l$ in a model tuple would not necessarily impose any consistency constraint by CR_2. We explain these two restrictions at an example.

Example 5.1. *Figure 5.8 extends the example in Figure 5.1 with further classes in the consistency relations, such that they do not only relate single classes to each other. It defines an address for employees and, in the second example, also a*

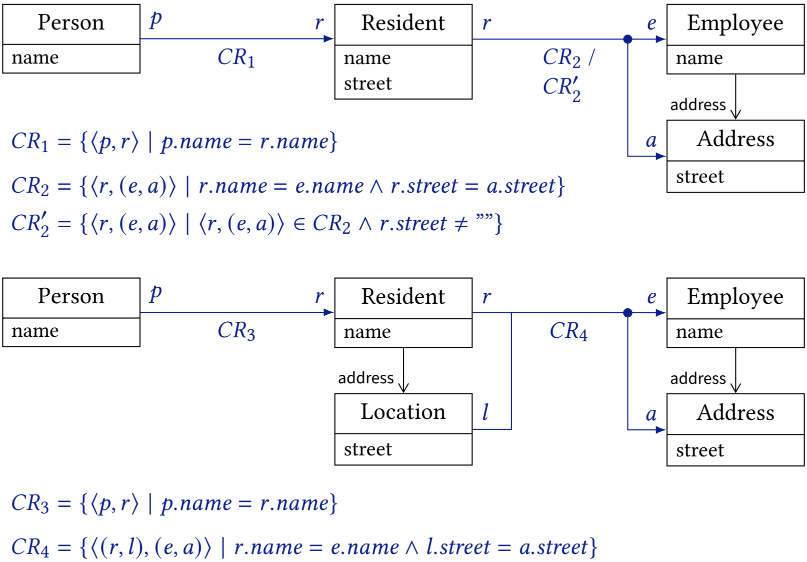


Figure 5.8.: Two scenarios, each with two consistency relations: Consistency relations CR_1 and two options CR_2, CR'_2 with $CR_1 \otimes CR_2 \neq \emptyset$ and $CR_1 \otimes CR'_2 = \emptyset$, and consistency relations CR_3 and CR_4 with $CR_3 \otimes CR_4 = \emptyset$ and $CR_4^T \otimes CR_3^T \neq \emptyset$. Taken from [Kla+20, Fig. 3].

location for the addresses of residents, which are represented in additional classes. Both examples contain consistency relations between persons and residents (CR_1 and CR_3), which define that for each person a resident with the same name has to exist. The examples provide different options for the consistency relation between residents (with locations) and employees with addresses (CR_2, CR'_2, and CR_4), which exemplify the necessity for the restrictions in Definition 5.1:

1. *$CR_1 \otimes CR_2$: CR_2 requires for each resident an employee with the same name and an address with the same street name. Because residents with arbitrary street names are consistent to a person with the same name, $CR_1 \otimes CR_2$ relates each person to an employee having the same name and addresses with all possible street names. All models that are consistent to the concatenation are also consistent to the single relations.*

2. *$CR_1 \otimes CR'_2$: CR'_2 is similar to CR_2 but additionally requires that the street of a resident must not be empty. In consequence, for a resident with an empty street name it is not required that an employee exists. This*

results in $CR_1 \otimes CR_2' = \emptyset$, because every person is consistent to a resident with an empty street name, thus not requiring a corresponding employee. This shows the necessity of the second restriction in the definition.

3. *$CR_3 \otimes CR_4$: The concatenation $CR_3 \otimes CR_4$ is obviously empty, because CR_3 requires a resident for each person, but CR_4 only requires an employee if there is also a location. Such a location does not necessarily exist if a person exists, thus if the models are consistent to CR_3 and CR_4, there does not have to be an employee for any contained person. This shows the necessity for the first restriction in Definition 5.1, which would require a left condition element from CR_4 (resident and location) to be a subset of a right condition element in CR_3 (resident).*
4. *$CR_4^T \otimes CR_3^T$: This concatenation of transposed relations contains all combinations of each possible employee with all addresses and relates them to a person with the same name. This is reasonable, because CR_4^T requires for all existing employees and addresses that an appropriate resident with the same name has to exist, which then requires a person with that name to exist due to CR_3^T. The definition does only cover that case due to its first restriction, because $\mathfrak{c}_{l,2}$, i.e., the resident related to a person by CR_3^T is a subset of $\mathfrak{c}_{r,1}$, i.e., a tuple of resident and location.*

We can formally show that the defined notion of concatenation does not lead to any restriction of consistency regarding the single relations:

Lemma 5.1 (Concatenation Consistency)

Let CR_1 and CR_2 be two consistency relations for a metamodel tuple $\mathfrak{M}$, and let $CR = CR_1 \otimes CR_2$ be their concatenation. Then it holds that:

$$\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ consistent to } \{CR_1, CR_2\} \Rightarrow \mathfrak{m} \textit{ consistent to } CR\big)$$

Proof. For any tuple of models $\mathfrak{m}$ that is consistent to CR_1 and CR_2, take a witness structure W_1 that witnesses consistency of $\mathfrak{m}$ to CR_1 and W_2 that witnesses consistency of $\mathfrak{m}$ to CR_2. Now consider the composed witness structure $W = W_1 \otimes W_2$. We show that W is a valid witness structure for CR.

Let us assume there were $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle, \langle \mathfrak{c}_l', \mathfrak{c}_r' \rangle \in W$ with $\mathfrak{c}_l = \mathfrak{c}_l'$ and $\mathfrak{c}_r \neq \mathfrak{c}_r'$, such that W is not a witness structure for CR. Per definition, $\mathfrak{c}_l$ only occurs in one $\langle \mathfrak{c}_l, \mathfrak{c}_{r,1} \rangle \in W_1$. So there must be two consistency relation pairs

$\langle \mathfrak{c}_{l,2}, \mathfrak{c}_r \rangle, \langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_r \rangle \in CR_2$ with $\mathfrak{c}_{l,2} \subseteq \mathfrak{c}_{r,1}$ and $\mathfrak{c}'_{l,2} \subseteq \mathfrak{c}_{r,1}$. However, since $\mathfrak{c}_{l,2}$ and $\mathfrak{c}'_{l,2}$ contain instances of the same classes and are both subsets of the same object tuple $\mathfrak{c}_{r,1}$, we have $\mathfrak{c}_{l,2} = \mathfrak{c}'_{l,2}$. So we know that W fulfills the first condition of a witness structure according to Definition 4.19 for consistency:

$$\forall \langle \mathfrak{c}_{l,1}, \mathfrak{c}_{r,1} \rangle, \langle \mathfrak{c}_{l,2}, \mathfrak{c}_{r,2} \rangle \in W : \big(\langle \mathfrak{c}_{l,1}, \mathfrak{c}_{r,1} \rangle = \langle \mathfrak{c}_{l,2}, \mathfrak{c}_{r,2} \rangle \vee \mathfrak{c}_{l,1} \neq \mathfrak{c}_{l,2} \wedge \mathfrak{c}_{r,1} \neq \mathfrak{c}_{r,2} \big)$$

Additionally, since W_1 and W_2 are witness structures for consistency of $\mathfrak{m}$ to CR_1 and CR_2, the model tuple contains all condition elements in W_1 and W_2. Consequentially, $\mathfrak{m}$ also contains the condition elements in W, as those in W are composed of the ones in W_1 and W_2. This implies that the second condition of Definition 4.19 is fulfilled:

$$\forall \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in W : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}_l \wedge \mathfrak{m} \textit{ contains } \mathfrak{c}_r \big)$$

Finally, we assume the third condition of Definition 4.19 was unfulfilled, i.e.:

$$\exists \, \mathfrak{c}'_l \in \mathbb{C}_{l,CR} : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}'_l \wedge \mathfrak{c}'_l \notin \mathbb{C}_{l,W} \big)$$

We know that $\mathbb{C}_{l,CR} \subseteq \mathbb{C}_{l,CR_1}$, because the left condition elements in CR are, per definition, taken from the left condition elements in CR_1 and thus also contained in CR_1. Since $\mathfrak{m}$ *contains* $\mathfrak{c}'_l$, there must be a consistency relation pair $\langle \mathfrak{c}'_l, \mathfrak{c}'_{r,1} \rangle \in W_1$ that witnesses consistency of $\mathfrak{c}'_l$ according to CR_1. There must be at least one consistency relation pair $\langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle \in CR_2$ with $\mathfrak{c}'_{l,2} \subseteq \mathfrak{c}'_{r,1}$, because otherwise $\mathfrak{c}'_l$ would, per definition, not occur in the left condition of CR. For all such tuples $\langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle$, we know that $\mathfrak{m}$ *contains* $\mathfrak{c}'_{l,2}$, because $\mathfrak{m}$ *contains* $\mathfrak{c}'_{r,1}$ due to its containment in W_1 and due to $\mathfrak{c}'_{l,2} \subseteq \mathfrak{c}'_{r,1}$. In consequence, consistency to CR_2 requires that for one of those $\mathfrak{c}'_{r,2}$ it holds that $\mathfrak{m}$ *contains* $\mathfrak{c}'_{r,2}$ and that there is $\langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle \in W_2$ that witnesses this consistency. Summarizing, due to $\langle \mathfrak{c}'_l, \mathfrak{c}'_{r,1} \rangle \in W_1$ and $\langle \mathfrak{c}'_{l,2}, \mathfrak{c}'_{r,2} \rangle \in W_2$ with $\mathfrak{c}'_{l,2} \subseteq \mathfrak{c}'_{r,1}$ and due to the definition of W as $W_1 \otimes W_2$, we know that $\langle \mathfrak{c}'_l, \mathfrak{c}'_{r,2} \rangle \in W$, which breaks our assumption. So we have shown that:

$$\forall \, \mathfrak{c}'_l \in \mathbb{C}_{l,CR} : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}'_l \Rightarrow \mathfrak{c}'_l \in \mathbb{C}_{l,W} \big)$$

Summarizing, we have shown that W fulfills all three requirements for a witness structure for consistency according to Definition 4.19, so we know that $\mathfrak{m}$ *consistent to* CR. $\square$

5.2.2. Transitive Closure of Consistency Relations

Based on the introduced notion of concatenation, we can define a transitive closure for a consistency relation set, which contains all relations in that set complemented by all possible concatenations of them, i.e., *implicit relations* of that set. Having shown that our definition of concatenation of consistency relations is well-defined in the sense that it does not introduce further restrictions for consistency, we can show that the transitive closure does not restrict consistency in comparison to the set of consistency relations itself.

Definition 5.2 (Consistency Relations Transitive Closure)
Let $\mathbb{CR}$ be a set of consistency relations. We define its transitive closure $\mathbb{CR}^+$ as:

$$\mathbb{CR}^+ := \{CR \mid \exists\, CR_1, \dots, CR_k \in \mathbb{CR} : CR = CR_1 \otimes \dots \otimes CR_k\}$$

The transitive closure of a set of consistency relations $\mathbb{CR}$ contains all consistency relations of $\mathbb{CR}$ and all their concatenations. Thus, the transitive closure contains consistency relations that relate all elements that are directly or indirectly related due to $\mathbb{CR}$. Due to cycles in the concatenation of relations, this closure can, in general, be of infinite size.

The transitive closure of a consistency relation set does not restrict consistency in comparison to the original set, i.e., if a model tuple is consistent to a set of consistency relations, it is also consistent to their transitive closure. We show that by first extending the argument of Lemma 5.1, which shows that concatenation does not further restrict consistency, to the transitive closure, which is only a set of concatenations of consistency relations.

Lemma 5.2 (Relation Set Consistency)
Let $\mathbb{CR}$ be a set of consistency relations for a tuple of metamodels $\mathfrak{M}$. Then it holds that:

$$\forall\, CR \in \mathbb{CR}^+ \setminus \mathbb{CR} : \exists\, CR_1, \dots, CR_k \in \mathbb{CR} : \forall\, \mathfrak{m} \in I_{\mathfrak{M}} :$$
$$\big(\mathfrak{m} \textit{ consistent to } \{CR_1, \dots CR_k\} \Rightarrow \mathfrak{m} \textit{ consistent to } CR\big)$$

Proof. Per definition, every $CR \in \mathbb{CR}^+$ is a concatenation of relations in $\mathbb{CR}$, i.e.:

$$\forall\, CR \in \mathbb{CR}^+ : \exists\, CR_1, \ldots, CR_k \in \mathbb{CR} : CR = CR_1 \otimes \ldots \otimes CR_k$$

We know for every two consistency relations CR_1 and CR_2 and all model tuples $\mathfrak{m}$ that if $\mathfrak{m}$ *consistent to* $\{CR_1, CR_2\}$, then $\mathfrak{m}$ *consistent to* $CR_1 \otimes CR_2$ (Lemma 5.1). Inductively applying that argument to $CR_1, \ldots, CR_k$ shows that $\mathfrak{m}$ *consistent to* CR whenever $\mathfrak{m}$ *consistent to* $\{CR_1, \ldots, CR_k\}$. □

As a result of this lemma, we can show that the transitive closure of a consistency relation set considers the same tuples of models consistent as the consistency relation set itself.

Lemma 5.3 (Transitive Closure Consistency)
Let $\mathbb{CR}$ be a consistency relation set for a metamodel tuple $\mathfrak{M}$. Then it holds that:

$$\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \left(\mathfrak{m} \textit{ consistent to } \mathbb{CR} \Leftrightarrow \mathfrak{m} \textit{ consistent to } \mathbb{CR}^+\right)$$

Proof. Adding a consistency relation to a set of consistency relations can never relax consistency, i.e., it cannot lead to models being consistent that were not considered consistent before. Definition 4.19 for consistency defines models as consistent when they are consistent to all consistency relations in a set. Thus, only adding relations can further restrict the set of consistent model tuples. In consequence, it holds that:

$$\mathfrak{m} \textit{ consistent to } \mathbb{CR}^+ \Rightarrow \mathfrak{m} \textit{ consistent to } \mathbb{CR}$$

According to Lemma 5.3, a tuple of models that is consistent to $\mathbb{CR}$ is always consistent to all transitive relations in $\mathbb{CR}^+$ as well. Thus, we know that:

$$\mathfrak{m} \textit{ consistent to } \mathbb{CR} \Rightarrow \mathfrak{m} \textit{ consistent to } \mathbb{CR}^+$$

In consequence, the same models are consistent to $\mathbb{CR}$ and its transitive closure $\mathbb{CR}^+$. □

5.2.3. Compatibility of Consistency Relations

Based on the notion of fine-grained consistency relations and their concatenation, we can precisely formulate our initially informal notion of *compatibility* of consistency relations. We have stated that we consider consistency relation incompatible if they are contradictory, like the relation between names in our initial example in Figure 5.1. In that example, for residents with non-lowercase names no consistent tuple of models could be derived. We formalize this notion of *non-contradictory* relations by requiring that relations may not restrict that an object tuple, for which consistency is defined in any consistency relation, cannot occur in a consistent model tuple anymore.

Definition 5.3 (Compatibility)

Let $\mathbb{CR}$ be a set of consistency relations for a tuple of metamodels $\mathfrak{M}$. We say that:

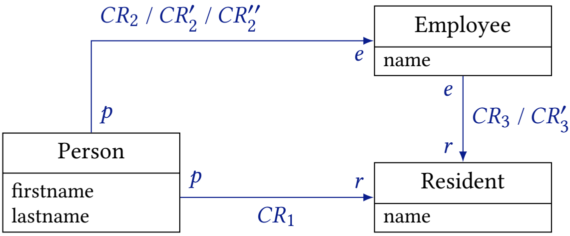
$$\mathbb{CR} \textit{ compatible } :\Leftrightarrow \forall CR \in \mathbb{CR} : \forall \mathfrak{c} \in \mathbb{c}_{l,CR} : \exists \mathfrak{m} \in I_{\mathfrak{M}} : \\ (\mathfrak{m} \textit{ contains } \mathfrak{c} \wedge \mathfrak{m} \textit{ consistent to } \mathbb{CR})$$

We call a set of consistency relation $\mathbb{CR}$ *incompatible* if it is not compatible.

We exemplify this notion of compatibility at an extract of the initial example with different consistency relations.

Example 5.2. *Figure 5.9 shows an extract of the three metamodels from Figure 5.1 and several consistency relations, of which different combinations are compatible or incompatible according to the previous definition. We always consider the actual relations together with their transposed ones to have a symmetric set of consistency relations.*

$\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3, CR_3^T\}$**:** *These relations are obviously compatible, because they relate firstname, lastname and name in the same way. Thus, for each object with any name, and thus any condition element in all of the consistency relations, a consistent model tuple can be found by adding instances of the other classes with appropriate names.*



$$CR_1 = \{\langle p, r\rangle \mid r.name = p.\mathit{firstname} + \text{"␣"} + p.lastname\}$$

$$CR_2 = \{\langle p, e\rangle \mid e.name = p.\mathit{firstname} + \text{"␣"} + p.lastname\}$$
$$CR_2' = \{\langle p, e\rangle \mid e.name = p.\mathit{firstname} + \text{",␣"} + p.lastname\}$$
$$CR_2'' = \{\langle p, e\rangle \mid e.name = p.lastname + \text{"␣"} + p.\mathit{firstname}\}$$

$$CR_3 = \{\langle r, e\rangle \mid r.name = e.name\}$$
$$CR_3' = \{\langle r, e\rangle \mid r.name = e.name.toLower\}$$

Figure 5.9.: Three metamodels with different options for consistency relations. The relation sets $\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3, CR_3^T\}$ and $\{CR_1, CR_1^T, CR_2'', CR_2''^T, CR_3, CR_3^T\}$ are compatible, whereas the sets $\{CR_1, CR_1^T, CR_2', CR_2'^T, CR_3, CR_3^T\}$ and $\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3', CR_3'^T\}$ are not. Taken from [Kla+20, Fig. 4].

$\{CR_1, CR_1^T, CR_2', CR_2'^T, CR_3, CR_3^T\}$**:** *These relations are incompatible, because each person p1 requires the existence of an additional person p2 with p2.firstname = p1.firstname + ", " and p2.lastname = p1.lastname due to CR_2' and the transitive relations requiring the addition of a comma. Thus, each person would require an infinite number of further persons to exist in a consistent tuple of models. Models are, however, finite, so there is no such model tuple and the relations are incompatible.*

$\{CR_1, CR_1^T, CR_2', CR_2'^T, CR_3, CR_3^T\}$**:** *These relations are compatible. The relations define that for a person p1, another person p2 with p2.firstname = p1.lastname and p2.lastname = p1.firstname has to exist, so that the tuple of models is consistent. Although that behavior may not be desired, it does not violate the definition of compatibility, because for every object in the relations, a consistent model tuple can be constructed. In general, it can even be necessary that consistency relations require the same elements with swapped attribute values to exist, such that this behavior can and should not be forbidden. Finally, such a relation does also not prevent a consistency* *preservation rule from finding a consistent model tuple. In consequence, such behavior may be undesired due to the specific semantics of a metamodel's domain, but it can neither be detected automatically nor does it lead to problems when executing transformations.*

$\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3', CR_3'^T\}$**:** *These consistency relations reflect the ones of our motivational example in Figure 5.2 for an intuitive notion of incompatibility. The formal definition of compatibility also considers these relations as incompatible, because it is not possible to create a resident with an uppercase name, such that the containing tuple of models is consistent. For a resident with name = "A␣B", a person with firstname = "A" and lastname = "B" has to exist, which requires the existence of an employee with name = "A␣B". Now CR_3' requires a resident with name = "a␣b" to exist, which in turn requires a person with firstname = "a" and lastname = "b" and an employee with name = "a␣b" to exist. In consequence, there are two employees, one with the uppercase and one with the lowercase name, for which a resident with the lowercase name has to exist according to the relation CR_3'. So there is no witness structure with a unique mapping between the elements that is required to fulfill Definition 4.19 for consistency.*

To summarize, compatibility is supposed to ensure that consistency relations do not impose restrictions on other relations such that their condition elements, for which consistency is defined, can never occur in consistent models. The goal of ensuring compatibility is especially to prevent the execution of consistency preservation rules in transformation networks from non-termination, as it may occur in the second example scenario, in which an infinitely large model would be required to fulfill the consistency relations.

Analogously to the equivalence of a set of consistency relations $\mathbb{CR}$ and its transitive closure $\mathbb{CR}^+$ in regards to consistency of a model tuple, we can show that a set of consistency relations and its transitive closure are always equal with regards to compatibility.

Lemma 5.4 (Transitive Closure Compatibility)
Let $\mathbb{CR}$ be a set of consistency relations. Then it holds that:

$$\mathbb{CR} \textit{ compatible } \Leftrightarrow \mathbb{CR}^+ \textit{ compatible}$$

Proof. The reverse direction of the equivalence is given by definition, since compatibility of a set of consistency relations implies compatibility of every subset by definition. So we have to show the forward direction by considering the compatibility definition for all $CR \in \mathbb{CR}^+$. We partition $\mathbb{CR}^+$ into $\mathbb{CR}$ and $\mathbb{CR}^+ \setminus \mathbb{CR}$ and consider their consistency relations independently.

First, we consider $CR \in \mathbb{CR}^+ \setminus \mathbb{CR}$. According to Definition 5.2 for the transitive closure, each $CR \in \mathbb{CR}^+ \setminus \mathbb{CR}$ is a concatenation of consistency relations $CR_1, \dots, CR_k \in \mathbb{CR}$. In consequence of that definition, we know that $\mathbb{c}_{l,CR} \subseteq \mathbb{c}_{l,CR_1}$, so it is given that:

$$\forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR_1} : \forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \\ \big(\mathfrak{m} \textit{ contains } \mathfrak{c}_l \Rightarrow \mathfrak{m} \textit{ contains } \mathfrak{c}'_l\big) \tag{1}$$

Since $\mathbb{CR}$ is compatible, we know from Definition 5.3 for compatibility that:

$$\forall\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR_1} : \exists\, \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}'_l \wedge \mathfrak{m} \textit{ consistent to } \mathbb{CR}\big) \tag{2}$$

Because of Equation 1 and Equation 2, we know that:

$$\forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}_l \wedge \mathfrak{m} \textit{ consistent to } \mathbb{CR}\big) \tag{3}$$

Furthermore, Lemma 5.3 states that:

$$\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ consistent to } \mathbb{CR} \Leftrightarrow \mathfrak{m} \textit{ consistent to } \mathbb{CR}^+\big) \tag{4}$$

In consequence of Equations 3 and 4, we know that:

$$\forall\, CR \in \mathbb{CR}^+ \setminus \mathbb{CR} : \forall\, \mathfrak{c} \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{m} \in I_{\mathfrak{M}} : \\ \big(\mathfrak{m} \textit{ contains } \mathfrak{c} \wedge \mathfrak{m} \textit{ consistent to } \mathbb{CR}^+\big) \tag{5}$$

Second, we consider $CR \in \mathbb{CR}$. Due to compatibility of $\mathbb{CR}$ and Lemma 5.3 showing equality of consistency of $\mathfrak{m}$ regarding $\mathbb{CR}$ and $\mathbb{CR}^+$, it holds that:

$$\forall\, CR \in \mathbb{CR} : \forall\, \mathfrak{c} \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{m} \in I_{\mathfrak{M}} : \\ \big(\mathfrak{m} \textit{ contains } \mathfrak{c} \wedge \mathfrak{m} \textit{ consistent to } \mathbb{CR}^+\big) \tag{6}$$

Equations 5 and 6 show compatibility of $\mathbb{CR}^+$ if $\mathbb{CR}$ is compatible. □

5.3. A Formal Approach to Prove Compatibility

In this section, we derive a formal approach for proving compatibility of consistency relations from the given definition. It bases on two ideas:

1. A set of consistency relations in which each pair of classes is only related across one concatenation of relations is inherently compatible, because there cannot be any contradictory relations. We precisely define this in a specific notion of *consistency relation trees.*
2. A consistency relation that is redundant in a set of relations, i.e., a relation that does not alter the notion of consistency for models regarding the other relations in that set, does not affect compatibility and can thus be removed from that set of relations.

Given a set of consistency relations, compatibility can be proven inductively by finding a consistency relation tree (or multiple such trees) that is equivalent to the set of relations by removing redundant relations from that set. Such an equivalent consistency relation tree serves as a *witness* for compatibility of a set of relations. In the following, we formalize and prove this inductive approach to check compatibility of a set of consistency relations.

The sketched approach is essentially based on a notion of equivalence for sets of consistency relations. We consider two sets of consistency relations equivalent if they consider the same model tuples consistent.

Definition 5.4 (Consistency Relations Equivalence)
Let $\mathbb{CR}_1$ and $\mathbb{CR}_2$ be two sets of consistency relations defined for a tuple of metamodels $\mathfrak{M}$. We say that:

$$\begin{aligned}&\mathbb{CR}_1 \textit{ equivalent to } \mathbb{CR}_2 :\Leftrightarrow\\&\quad \forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ consistent to } \mathbb{CR}_1 \Leftrightarrow \mathfrak{m} \textit{ consistent to } \mathbb{CR}_2\big)\end{aligned}$$

We later use the notion of equivalence to introduce a notion of redundancy that is compatibility-preserving. In the following, we first consider structures of consistency relation sets that are inherently compatible and afterwards discuss redundancy as a means to reduce and decompose a relation set into an equivalent composition of such inherently compatible structures.

We consider the following two properties of a consistency relation set that lead to its inherent compatibility.

Composability: The union of independent, compatible sets of relations is compatible.

Trees: Relations fulfilling a special notion of *consistency relation trees* are compatible.

Showing that these properties imply compatibility, we know that a consistency relation set of independent subsets of consistency relation trees is inherently compatible.

5.3.1. Independence of Consistency Relations

We consider two consistency relation sets to be independent if the tuples of classes they put into relation are disjoint.

Definition 5.5 (Consistency Relation Sets Independence)
Let $\mathbb{CR}_1$ and $\mathbb{CR}_2$ be two sets of consistency relations. We say that:

$$
\begin{aligned}
&\mathbb{CR}_1 \textit{ and } \mathbb{CR}_2 \textit{ are independent } :\Leftrightarrow \\
&\quad \bigcup_{CR \in \mathbb{CR}_1} \mathfrak{C}_{r,CR} \cap \bigcup_{CR \in \mathbb{CR}_2} \mathfrak{C}_{l,CR} = \emptyset \\
&\quad \wedge \bigcup_{CR \in \mathbb{CR}_2} \mathfrak{C}_{r,CR} \cap \bigcup_{CR \in \mathbb{CR}_1} \mathfrak{C}_{l,CR} = \emptyset
\end{aligned}
$$

We call $\mathbb{CR}$ *connected* if there is no partition of $\mathbb{CR}$ into two subsets that are independent, i.e.:

$$
\begin{aligned}
&\forall \mathbb{CR}_1, \mathbb{CR}_2 \subseteq \mathbb{CR} : \big(\mathbb{CR}_1 \cap \mathbb{CR}_2 = \emptyset \wedge \mathbb{CR}_1 \cup \mathbb{CR}_2 = \mathbb{CR} \\
&\quad \Rightarrow \neg(\mathbb{CR}_1 \textit{ and } \mathbb{CR}_2 \textit{ are independent })\big)
\end{aligned}
$$

In fact, this notion of independence is not the most general one that ensures preservation of compatibility. Such a notion would only require that for each condition element in each of the consistency relation sets still a consistent model tuple can be found when both consistency relation sets are considered

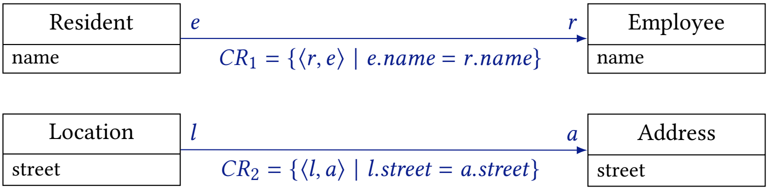


Figure 5.10.: Two independent (sets of) consistency relations. Taken from [Kla+20, Fig. 5].

together. This means that it is only necessary that for all instances of class tuples that may be required by one of the consistency relation sets to produce a consistent model tuple for each of the condition elements, there is no condition element containing these instances within the consistency relations of the other set. Such a notion does, however, become complicated to validate and the given one already reflects a reasonable notion of independence, which is sufficient for all cases that we consider in our evaluation, which indicates general adequacy. Thus, we stick to the given notion of independence.

Example 5.3. *Figure 5.10 depicts a simple example with two consistency relations CR_1 and CR_2, each relating instances of two disjoint classes with each other. Since there is no overlap in the classes that are related by the consistency relations, they are considered independent according to Definition 5.5.*

An important property of independent consistency relation sets is that computing their union is compatibility-preserving, i.e., the union of compatible, independent consistency relation sets is compatible as well.

Theorem 5.5 (Independent Relation Sets Compatibility)
Let $\mathbb{CR}_1$ and $\mathbb{CR}_2$ be two independent sets of consistency relations. Then it holds that:

$$\mathbb{CR}_1 \cup \mathbb{CR}_2 \textit{ compatible } \Leftrightarrow \mathbb{CR}_1 \textit{ compatible } \wedge \mathbb{CR}_2 \textit{ compatible}$$

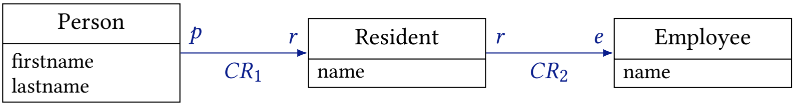
Proof. The forward direction is trivially given. Compatibility of the union of the consistency relation sets means that for every condition element in the consistency relations of the union, a model tuple containing that condition element and being consistent to the union of the consistency relation sets can be found. Then the same model tuple is consistent to each of the consistency relation sets and, in particular, the one containing the condition element.

The backward direction of the equivalence can be seen by construction. Since $\mathbb{CR}_1$ is compatible, per definition there is a model tuple $\mathfrak{m}$ for each condition element $\mathfrak{c}$ of the left condition of each consistency relation in $\mathbb{CR}_1$ that contains $\mathfrak{c}$ and that is consistent to $\mathbb{CR}_1$. Taking any such $\mathfrak{m}$, we create $\mathfrak{m}'$ by removing all elements from $\mathfrak{m}$ that are contained in any condition elements of the left conditions in every consistency relation $CR \in \mathbb{CR}_2$ and thus potentially require other elements to occur to be considered consistent to that consistency relation. The classes of these elements are thus in $\mathfrak{C}_{l,CR}$. In consequence, $\mathfrak{m}'$ does not contain any condition elements in left conditions of consistency relations in $\mathbb{CR}_2$ and is thus consistent to $\mathbb{CR}_2$ by definition. Additionally, $\mathfrak{m}'$ is still consistent to $\mathbb{CR}_1$, because due to the independence of $\mathbb{CR}_1$ and $\mathbb{CR}_2$, there cannot be any consistency relation $CR' \in \mathbb{CR}_1$ that requires the existence of the removed elements. Otherwise, the classes of these elements would be in $\mathfrak{C}_{r,CR'}$. Per definition of independence, however, $\mathfrak{C}_{l,CR} \cap \mathfrak{C}_{r,CR'} = \emptyset$, which is a contradiction. In consequence, for each condition element $\mathfrak{c}$ of each consistency relation in $\mathbb{CR}_1$, a model tuple that contains $\mathfrak{c}$ and is consistent to $\mathbb{CR}_1 \cup \mathbb{CR}_2$ exists. The argumentation applies to $\mathbb{CR}_2$ analogously, so the definition of compatibility is fulfilled for all condition elements of all consistency relations in $\mathbb{CR}_1 \cup \mathbb{CR}_2$. $\square$

The constructive proof can also be reflected exemplarily in Figure 5.10: Take any tuple of models that, for example, contains a resident with an arbitrary name and is consistent to CR_1, i.e., that also contains an employee with the same name. If that tuple of models contains any addresses or locations, they can be removed without violating consistency to CR_1, because addresses and locations are independently related by CR_2.

5.3.2. Consistency Relation Trees

In addition to independence of consistency relation sets as a property that inherently implies compatibility, we aim at finding a specific structure of a connected consistency relation set that leads to inherent compatibility of the contained relations. In consequence, if we can reduce sets of consistency relations to independent sets of such a structure in a compatibility-preserving



$$CR_1 = \{\langle p, r\rangle \mid r.name = p.firstname + \text{"␣"} + p.lastname\}$$
$$CR_2 = \{\langle r, e\rangle \mid r.name = e.name\}$$

Figure 5.11.: A consistency relation tree $\{CR_1, CR_1^T, CR_2, CR_2^T\}$. Adapted from [Kla+20, Fig. 6].

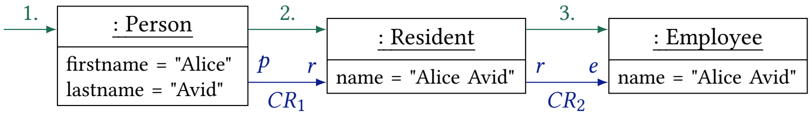
way, we know that the relations are compatible. Intuitively, such a structure can be expected from a kind of trees, because then there are no two concatenations of relations that can relate elements in a contradictory way.

Definition 5.6 (Consistency Relation Tree)
Let $\mathbb{CR}$ be a symmetric, connected consistency relation set. We say:

$$\begin{aligned}
&\mathbb{CR} \textit{ is a consistency relation tree } :\Leftrightarrow\\
&\quad \forall\, CR = CR_1 \otimes \ldots \otimes CR_m \in \mathbb{CR}^+ :\\
&\quad \forall\, CR' = CR'_1 \otimes \ldots \otimes CR'_n \in \mathbb{CR}^+ \setminus CR :\\
&\qquad \Big[\forall\, s, t \mid s \neq t : \big(CR_s \neq CR_t^T \wedge CR'_s \neq CR'^T_t\big)\\
&\qquad \Rightarrow \mathfrak{C}_{l,CR} \cap \mathfrak{C}_{l,CR'} = \emptyset \vee \mathfrak{C}_{r,CR} \cap \mathfrak{C}_{r,CR'} = \emptyset\Big]
\end{aligned}$$

The definition of a consistency relation tree requires that there are no two sequences of consistency relations that put the same classes into relation, i.e., all pairs of classes are only put into relation by a single concatenation of consistency relations. Since we assume a symmetric set of consistency relations, we exclude the symmetric relations from that argument. Otherwise, there would always be two such concatenations by adding a consistency relation and its transposed relation to any other concatenation.

Example 5.4. *Figure 5.11 depicts a rather simple consistency relation tree. Persons are related to residents and residents are related to employees, all having the same names or a concatenation of firstname and lastname, respectively, by the relations CR_1 and CR_2, as well as their transposed relations CR_1^T and CR_2^T. There are no classes that are put into relation across different paths of consistency relations, thus the definition for a consistency relation tree is fulfilled.*



$$CR_1 = \{\langle p, r\rangle \mid r.name = p.firstname + \text{"␣"} + p.lastname\}$$

$$CR_2 = \{\langle r, e\rangle \mid r.name = e.name\}$$

Figure 5.12.: Model tuple construction with condition element of CR_1 containing person "Alice Avid" for a consistency relation tree of relations in Figure 5.11. Adapted from [Kla+20, Fig. 7]. Arrows with numbers indicate the order in which elements are created.

If an additional relation between persons and employees was specified, like in Figure 5.1, the tree definition would not be fulfilled.

The definition also covers the more complicated case in which multiple classes are put into relation by consistency relations, but only a subset of them that is put into relation by different consistency relations. We can now prove that a consistency relation tree is always compatible. To preserve the reading flow, we only provide a proof sketch in the following and refer for the complete proof with an auxiliary lemma to Appendix A.

Theorem 5.6 (Consistency Relation Tree Compatibility)
Let $\mathbb{CR}$ be a consistency relation tree. Then $\mathbb{CR}$ is compatible.

Proof Sketch. The complete proof is given in Appendix A. It is based on a proven lemma stating that starting with any of the consistency relations of a consistency relation tree, there is a sequence of the consistency relations such that there is no overlap in the classes of the conditions at the right sides of these relations and that for each relation there is no overlap in the classes of the condition at the left side with the ones at the right side of any subsequent relation in the sequence. More informally speaking, the relations do not induce a cycle between any of the classes in the metamodels. We use this insight to define a construction approach for such sequences given a set of consistency relations. For proving compatibility, we need to show that for each condition element in a consistency relation, we find a consistent model tuple containing it. Thus, we start with each condition element of each relation and add a corresponding element according to that relation. We then inductively add further elements required by other consistency relations due to the just added elements. Based on the properties of consistency relation trees, we can show that this construction is always possible and terminates with a consistent model tuple.

A simple example for that construction is depicted in Figure 5.12, which is based on the relations in the consistency relation tree in Figure 5.11 and more precisely explained in the complete proof. The example shows the construction for the condition element with the person named "Alice Avid", consecutively selecting consistency relations for whose fulfillment further elements, namely an appropriate resident and employee, are added. □

Summarizing, Theorem 5.5 and Theorem 5.6 show that consistency relation sets fulfilling the notion of consistency relation trees are compatible and that combining compatible independent sets of relations is compatibility-preserving. In consequence, having a consistency relation set that consists of independent subsets that are consistency relation trees, this set of relations is inherently compatible. An approach that evaluates whether a given set of consistency relations fulfills Definition 5.5 and Definition 5.6 for independence and trees can be used to prove compatibility of those relations.

5.3.3. Redundancy of Consistency Relations

We have introduced specific structures of consistency relation sets that are inherently compatible. However, actual consistency relation sets have such a structure only in specific cases. In general, like in our initial example in Figure 5.1, multiple consistency relations may put the same classes into relation, such that the definition for consistency relation trees is not fulfilled.

In the following, we present an approach to reduce the relations in a set of consistency relations to, in the best case, an equivalent set of independent consistency relation trees. The essential idea is to identify relations within a set, such that whether or not they are contained in the set does not change its compatibility. An approach that finds such relations and, for the scope of the analysis, removes them from the set until the remaining relations represent independent consistency relation trees proves compatibility of the given set of relations. We first define the term of a *compatibility-preserving* relation.

Definition 5.7 (Compatibility-Preserving Consistency Relation)
Let $\mathbb{CR}$ be a compatible set of consistency relations and let CR be a consistency relation. We say that:

$$CR \textit{ compatibility-preserving to } \mathbb{CR} :\Leftrightarrow \mathbb{CR} \cup \{CR\} \textit{ compatible}$$

To find such compatibility-preserving relations, we introduce the notion of *redundant* relations and prove the property of being compatibility preserving. Informally speaking, a relation is redundant if it is expressed transitively across others, i.e., if it does not restrict or relax consistency compared to a combination of other relations. We precisely define redundancy as follows.

Definition 5.8 (Redundant Consistency Relation)
Let $\mathbb{CR}$ be a set of consistency relations for a tuple of metamodels $\mathfrak{M}$. For a consistency relation $CR \in \mathbb{CR}$, we say that:

$$CR \textit{ redundant in } \mathbb{CR} :\Leftrightarrow \exists CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall \mathfrak{m} \in I_{\mathfrak{M}} :$$
$$\big(\mathfrak{m} \textit{ consistent to } CR' \Rightarrow \mathfrak{m} \textit{ consistent to } CR\big)$$

The definition of redundancy of a consistency relation *CR* ensures that there is another consistency relation, possibly transitively expressed across others, such that if a model is consistent to that other relation, it is also consistent to *CR*. This means that there are no model tuples that are considered inconsistent to *CR* but not to another relation, thus *CR* does not restrict consistency. Actually, the definition of redundancy implies that the set of consistency relations with and without the redundant one are equivalent according to Definition 5.4, thus both consider the same model tuples to be consistent.

Lemma 5.7 (Redundant Relations Equivalence)
Let $CR \in \mathbb{CR}$ be a redundant consistency relation in a relation set $\mathbb{CR}$. Then $\mathbb{CR}$ is equivalent to $\mathbb{CR} \setminus \{CR\}$.

Proof. As discussed in Lemma 5.3, adding a consistency relation to a set of consistency relations can never lead to a relaxation of consistency, i.e., models becoming consistent that were not considered consistent before. This is a direct consequence of Definition 4.19 for consistency, which requires models to be consistent to all consistency relations in a set to be considered consistent and thus restricts the set of consistent model tuples by adding further consistency relations. In consequence, it holds that:

$$\mathfrak{m} \textit{ consistent to } \mathbb{CR} \Rightarrow \mathfrak{m} \textit{ consistent to } \mathbb{CR} \setminus \{CR\}$$

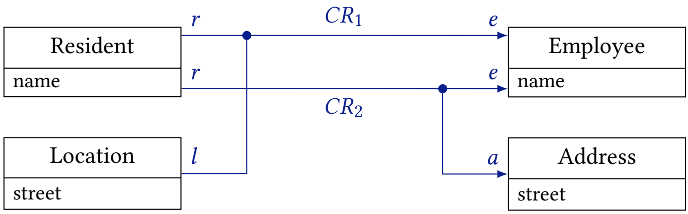
Additionally, a direct consequence of Definition 5.8 for redundancy is that a redundant consistency relation does not restrict consistency, as it considers all models consistent that are also considered consistent to another consistency relation in the transitive closure of the consistency relation set. Thus, all models that are considered consistent to the transitive closure of $\mathbb{CR} \setminus \{CR\}$ are also consistent to CR and thus to all relations in $\mathbb{CR}$:

$$\mathfrak{m} \textit{ consistent to } (\mathbb{CR} \setminus \{CR\})^+ \Rightarrow \mathfrak{m} \textit{ consistent to } \mathbb{CR}$$

According to Lemma 5.3, each tuple of models that is consistent to a consistency relation set is also consistent to its transitive closure and vice versa. So the latter implication is also true for $\mathbb{CR} \setminus \{CR\}$. Summarizing, $\mathbb{CR}$ and $\mathbb{CR} \setminus \{CR\}$ are equivalent. □

In general, to consider a consistency relation redundant in $\mathbb{CR}$, it has to define equal or weaker requirements for consistency than one of the other relations in $\mathbb{CR}$. Informally speaking, such weaker requirements mean that the redundant relation must have weaker conditions, i.e., it must require consistency for less objects and consider the same or more objects consistent to each of the left condition elements.

Example 5.5. *Such weaker consistency requirements are exemplified in the example in Figure 5.13, which shows a consistency relation CR_1 that is redundant in $\{CR_1, CR_2\}$. A redundant consistency relation, such as CR_1, must have weaker requirements in the left condition, such that it requires consistent elements to exist in less cases. This means that it may have a larger set of classes that are matched and that there may be less condition elements for which consistency is required. In case of CR_1, the left condition contains both a resident and a location, whereas the left condition of CR_2 only contains residents. Thus, CR_1 requires consistent elements, i.e., employees, only if a resident and a location exist, whereas CR_2 already requires that for an existing resident. Furthermore,*



$CR_1 = \{\langle (r,l), e\rangle \mid r.name \neq \text{""} \wedge (r.name = e.name \vee r.name = e.name.toLower)\}$

$CR_2 = \{\langle r, (e,a)\rangle \mid r.name = e.name \wedge a.street \neq \text{""}\}$

Figure 5.13.: Redundant consistency relation CR_1 in $\{CR_1, CR_2\}$. Taken from [Kla+20, Fig. 8].

the residents for which CR_1 constrains consistency are a subset of those for which CR_2 constrains consistency, as CR_1 does not constrain consistency for residents with an empty name. Thus, the left condition elements of CR_1 are a subset of those of CR_2. In consequence, if CR_1 constrains consistency for a resident and a location, CR_2 constrains it for the contained resident anyway.

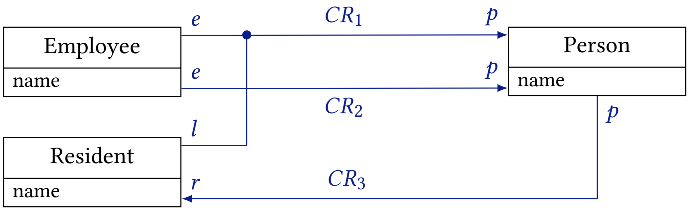
Additionally, a redundant consistency relation, such as CR_1, must have weaker requirements for the elements at the right side, such that one of the consistent right condition elements is contained anyway, because another relation already required them. This means that the relation may have a smaller set of classes, of whom instances are required to consider the models consistent. In addition, there may be more condition elements at the right side considered consistent to condition elements at the left side to not restrict the elements considered consistent. CR_1 only requires an employee to exist for a resident, whereas CR_2 also requires a non-empty address to exist. Additionally, CR_1 does not restrict the employees that are considered consistent to residents in comparison CR_2, as it also considers employees with the same name as consistent, but additionally those having the name of the resident in lowercase.

Our goal is to have a compatibility-preserving notion of redundancy, i.e., adding a redundant relation to a compatible relation set should preserve compatibility. Unfortunately, the up to now given intuitive redundancy definition is not compatibility-preserving.

Proposition 5.8 (Redundant Relations Non-Compatibility)
Let $\mathbb{CR}$ *be a compatible consistency relation set and let* CR *be a consistency relation that is redundant in* $\mathbb{CR} \cup \{CR\}$*. Then* CR *is in general not compatibility-preserving to* $\mathbb{CR}$*.*

Proof. We prove the proposition by providing a counterexample. Consider the example in Figure 5.14. CR_2 relates each employee to a person with the same *name* and CR_3 relates each person to a resident with the same *name* in lowercase. The consistency relation set $\{CR_2, CR_3\}$ is obviously compatible, because for each employee and each person, which constitute the left condition elements of the consistency relations, a consistent model tuple containing the person and employee, respectively, can be created by adding the appropriate person or employee with the same *name* and a resident with the *name* in lowercase. Furthermore, CR_1 is redundant in $\{CR_1, CR_2, CR_3\}$. If a model is consistent to CR_2, it is also consistent to CR_1, since CR_1 also requires persons with the same *name* as an employee to be contained in a model tuple but in less cases, precisely those in which the models also contain a resident such that the employee's *name* is the resident's *name* written in uppercase.

$\{CR_1, CR_2, CR_3\}$ is, however, not compatible. Intuitively, this is because CR_1 and CR_3 define an incompatible mapping between the names of residents and persons. Consider a model with an employee and a resident with $name = {}$"A". This is a condition element in $\mathbb{c}_{l,CR_1}$. Consequentially, CR_1 requires a person with $name = {}$"A" to exist. Then CR_3 requires a resident with $name = {}$"a" to exist. Thus, there are two tuples of employees and residents, both with the employee named "A" and one with resident "A" as well as one with resident "a", for which a person with $name = {}$"A" is required by CR_1. However, CR_1 forbids to have two residents with one having the lowercase *name* of the other, because both are condition elements in CR_1 requiring an appropriate person to occur in a consistent model, but both can only be mapped to the same person with the uppercase *name*. In consequence, there is no witness structure with a unique mapping as required by Definition 4.19 for consistency. This example shows that adding a redundant consistency relation to a compatible consistency relation set does not necessarily preserve its compatibility. □



$$CR_1 = \{\langle(e, r), p\rangle \mid e.name = r.name.toUpper \wedge e.name = p.name\}$$

$$CR_2 = \{\langle e, p\rangle \mid e.name = p.name\}$$

$$CR_3 = \{\langle p, r\rangle \mid r.name = p.name.toLower\}$$

Figure 5.14.: A consistency relation CR_1 being redundant in $\{CR_1, CR_2, CR_3\}$ with $\{CR_2, CR_3\}$ being compatible and $\{CR_1, CR_2, CR_3\}$ being incompatible. Taken from [Kla+20, Fig. 9].

5.3.4. Compatibility-Preserving Redundancy

In consequence of Proposition 5.8, we need a stronger definition of redundancy, which is compatibility-preserving. The counterexample in Figure 5.14 shows that it is problematic if a redundant relation considers more classes in its left condition than the relation it is redundant to. Therefore, we define a stronger notion that restricts the left class tuple.

Definition 5.9 (Left-Equal Redundant Consistency Relation)
Let $\mathbb{CR}$ be a set of consistency relations for a metamodel tuple $\mathfrak{M}$. For a consistency relation $CR \in \mathbb{CR}$, we say:

$$\begin{aligned}
&CR \textit{ left-equal redundant in } \mathbb{CR} :\Leftrightarrow \\
&\quad \exists\, CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \\
&\qquad \big((\mathfrak{m} \textit{ consistent to } CR' \Rightarrow \mathfrak{m} \textit{ consistent to } CR) \wedge \mathfrak{C}_{l,CR} = \mathfrak{C}_{l,CR'}\big)
\end{aligned}$$

The definition of left-equal redundancy restricts the notion of redundancy to cases in which the left condition of the redundant consistency relation CR considers the same classes as the other relation in the set of consistency relations that induces consistency of a model tuple to CR. As discussed before, redundancy in general allows that the left condition of a redundant consistency relation can consider a superset of those classes.

Lemma 5.9 (Left-Equal Redundancy to Redundancy)

Let CR be a consistency relation that is left-equal redundant in a set of consistency relations $\mathbb{CR}$. Then CR is redundant in $\mathbb{CR}$.

Proof. Since the definition of left-equal redundancy is equal to the one for redundancy except for the additional class tuple restriction, redundancy of a left-equal redundant relation is a direct implication of the definition. □

We prove an auxiliary lemma to show that left-equal redundancy preserves compatibility. It shows that if a model tuple contains a left condition element of a left-equal redundant relation, i.e., if that relation requires the model tuple to contain corresponding elements for that object tuple, there is another relation that requires corresponding elements for that object tuple.

Lemma 5.10 (Left-Equal Redundancy Containment)

Let CR be a consistency relation that is left-equal redundant in a consistency relation set $\mathbb{CR}$ for a metamodel tuple $\mathfrak{M}$. Then it holds that:

$$\exists\, CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR'} :$$
$$\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}'_l \Rightarrow \mathfrak{m} \textit{ contains } \mathfrak{c}_l\big)$$

Proof. Due to left-equal redundancy of CR in $\mathbb{CR}$, we know per definition:

$$\exists\, CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall\, \mathfrak{m} \in I_{\mathfrak{M}} :$$
$$\big((\mathfrak{m} \textit{ consistent to } CR' \Rightarrow \mathfrak{m} \textit{ consistent to } CR) \land \mathfrak{C}_{l,CR} = \mathfrak{C}_{l,CR'}\big)$$

This implies that:

$$\exists\, CR' \in (\mathbb{CR} \setminus \{CR\})^+ : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \mathfrak{c}_l \in \mathbb{c}_{l,CR'}$$

Because if there was a $\mathfrak{c}_l \in \mathbb{c}_{l,CR}$ so that $\mathfrak{c}_l \notin \mathbb{c}_{l,CR'}$, then the model tuple $\mathfrak{m}$ only consisting of $\mathfrak{c}_l$ would be consistent to CR'. In contrast, there is at least one $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR$, so that $\mathfrak{m}$ needs to contain $\mathfrak{c}_r$ to be consistent to CR, which is not given by construction. This shows that $\mathbb{c}_{l,CR'}$ contains all elements in $\mathbb{c}_{l,CR}$, so there is always at least one element in $\mathbb{c}_{l,CR'}$ that a model tuple contains if it contains an element in $\mathbb{c}_{l,CR}$, which proves the lemma. □

Theorem 5.11 (Left-Equal Redundancy Compatibility)
Let $\mathbb{CR}$ be a compatible set of consistency relations and let CR be left-equal redundant in $\mathbb{CR} \cup \{CR\}$. Then $\mathbb{CR} \cup \{CR\}$ is compatible.

Proof. Left-equal redundancy of CR in $\mathbb{CR} \cup \{CR\}$ implies general redundancy according to Definition 5.8. In consequence, $\mathbb{CR}$ and $\mathbb{CR} \cup \{CR\}$ are equivalent, as shown in Lemma 5.7. Because of this equivalence, we know that:

$$\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ consistent to } \mathbb{CR} \Leftrightarrow \mathfrak{m} \textit{ consistent to } \mathbb{CR} \cup \{CR\}\big) \tag{1}$$

It follows from Definition 5.3 for compatibility and Equation 1:

$$\begin{aligned}&\forall\, CR' \in \mathbb{CR} : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR'} : \exists\, \mathfrak{m} \in I_{\mathfrak{M}} :\\ &\quad \big(\mathfrak{m} \textit{ contains } \mathfrak{c}_l \wedge \mathfrak{m} \textit{ consistent to } \mathbb{CR} \cup \{CR\}\big)\end{aligned} \tag{2}$$

This already shows that for $\mathbb{CR}$ the compatibility definition is fulfilled, so we need to prove that the compatibility definition is fulfilled for CR as well. Due to compatibility of $\mathbb{CR}$ and Lemma 5.4 showing equality of compatibility for a consistency relation set and its transitive closure, we know that:

$$\begin{aligned}&\forall\, CR' \in \mathbb{CR}^+ : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR'} : \exists\, \mathfrak{m} \in I_{\mathfrak{M}} :\\ &\quad \big(\mathfrak{m} \textit{ contains } \mathfrak{c}_l \wedge \mathfrak{m} \textit{ consistent to } \mathbb{CR}^+\big)\end{aligned} \tag{3}$$

Due to left-equal redundancy of CR in $\mathbb{CR} \cup \{CR\}$, we have shown in Lemma 5.10 that the following is true:

$$\begin{aligned}&\exists\, CR' \in \mathbb{CR}^+ : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR'} : \forall\, \mathfrak{m} \in I_{\mathfrak{M}} :\\ &\quad \big(\mathfrak{m} \textit{ contains } \mathfrak{c}'_l \Rightarrow \mathfrak{m} \textit{ contains } \mathfrak{c}_l\big)\end{aligned} \tag{4}$$

The combination of Equation 3 and Equation 4 gives:

$$\begin{aligned}&\exists\, CR' \in \mathbb{CR}^+ : \forall\, \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists\, \mathfrak{c}'_l \in \mathbb{c}_{l,CR'} :\\ &\quad \big[\forall\, \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}'_l \Rightarrow \mathfrak{m} \textit{ contains } \mathfrak{c}_l\big)\\ &\quad \wedge \exists\, \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}'_l \wedge \mathfrak{m} \textit{ consistent to } \mathbb{CR}^+\big)\big]\end{aligned}$$

A simplification by combining the two last lines of that statement leads to:

$$\forall \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists \mathfrak{m} \in I_{\mathfrak{M}} : \big(\mathfrak{m} \textit{ contains } \mathfrak{c}_l \land \mathfrak{m} \textit{ consistent to } \mathbb{CR}^+\big)$$

Due to Equation 1 and Lemma 5.3, which shows equality of consistency for a consistency relation set and its transitive closure, this is equivalent to:

$$\begin{aligned}&\forall \mathfrak{c}_l \in \mathbb{c}_{l,CR} : \exists \mathfrak{m} \in I_{\mathfrak{M}} :\\ &\quad \big(\mathfrak{m} \textit{ contains } \mathfrak{c}_l \land \mathfrak{m} \textit{ consistent to } \mathbb{CR} \cup \{CR\}\big)\end{aligned} \tag{5}$$

The combination of Equation 2 and Equation 5 shows that $\mathbb{CR} \cup \{CR\}$ fulfills Definition 5.3 for compatibility. □

Corollary 5.12 (Transitive Redundancy Compatibility)
Let $\mathbb{CR}$ be a compatible set of consistency relations and let $CR_1, \ldots, CR_k$ be consistency relations with:

$$\forall\, i \in \{1, \ldots, k\} : CR_i \textit{ left-equal redundant in } \mathbb{CR} \cup \{CR_1, \ldots, CR_i\}$$

Then $\mathbb{CR} \cup \{CR_1, \ldots, CR_k\}$ is compatible.

Proof. $\mathbb{CR}$ is compatible. Sequentially adding CR_i to $\mathbb{CR} \cup \{CR_1, \ldots, CR_{i-1}\}$ inductively ensures compatibility of $\mathbb{CR} \cup \{CR_1, \ldots, CR_i\}$ due to Theorem 5.11, which shows compatibility of $\mathbb{CR} \cup \{CR_1, \ldots, CR_{i-1}\}$. □

With Corollary 5.12, we have shown that if we have a set of consistency relations $\mathbb{CR}$ and are able to find a sequence of redundant consistency relations $CR_1, \ldots, CR_k$ according to Corollary 5.12 such that we know that $\mathbb{CR} \setminus \{CR_1, \ldots, CR_k\}$ is compatible, then it is proven that $\mathbb{CR}$ is compatible.

5.3.5. An Algorithm to Prove Compatibility

In the previous subsections, we have proven the following three central insights.

1. Compatibility is composable: If independent sets of consistency relations are compatible, their union is compatible as well (Theorem 5.5).

Algorithm 5.1 Proof for compatibility of consistency relations.

procedure ProveCompatibility($\mathbb{CR}$)
    if IsRelationTree($\mathbb{CR}$) **then**
        return TRUE
    end if
    if HasIndependentSubsets($\mathbb{CR}$) **then**
        $\{\mathbb{CR}_1, \mathbb{CR}_2\} \leftarrow$ FindIndependentSubsets($\mathbb{CR}$)
        isFirstSetCompatible $\leftarrow$ ProveCompatibility($\mathbb{CR}_1$)
        isSecondSetCompatible $\leftarrow$ ProveCompatibility($\mathbb{CR}_2$)
        return *isFirstSetCompatible* $\wedge$ *isSecondSetCompatible*
    end if
    $CR_{redundant} \leftarrow$ FindRedundantRelation($\mathbb{CR}$)
    if $CR_{redundant} \neq \emptyset$ **then**
        return ProveCompatibility($\mathbb{CR} \setminus \{CR_{redundant}\}$)
    end if
    return FALSE
end procedure

2. Consistency relation trees are compatible: If there are no two concatenations of consistency relations in a consistency relation set that relate the same classes, that set is compatible (Theorem 5.6).
3. Left-equal redundancy is compatibility-preserving: Adding a left-equal redundant consistency relation to a compatible consistency relation set, the union of that set with the redundant relation is compatible (Corollary 5.12).

These insights enable us to define a formal approach for proving compatibility of a set of consistency relations. Given a set of relations for which compatibility shall be proven, we search for consistency relations in that set that are left-equal redundant to it. If iteratively removing such redundant relations from the set leads to a set of independent consistency relation trees, it is proven that the initial set of consistency relations is compatible.

Algorithm 5.1 realizes this procedure. It executes the described steps and assumes appropriate procedures to find out whether the given set of relations is a relation tree, whether it consists of independent subsets, and whether it contains a redundant relation. It is easy to see that this algorithm is correct, as is implements the proven findings summarized before. This does, however, not mean that implementing the sub-procedures is trivial. We provide a practical approach to realize them in the subsequent section.

> **Theorem 5.13** (Compatibility Algorithm Correctness)
> *Algorithm 5.1 is correct, i.e., it only returns* TRUE *if the given consistency relation set* $\mathbb{CR}$ *is compatible.*

Proof. We make a case distinction for the returning statements of the algorithm.

1. If the consistency relation set is a tree, the algorithm directly returns TRUE (Lines 2–4), which is correct according to Theorem 5.6.
2. If the consistency relation set can be split into independent sets, the algorithm returns TRUE when both independent sets are identified as compatible by recursive application of the algorithm (Lines 5–10), which is correct according to Theorem 5.5.
3. If the consistency relation set contains a redundant relation, the algorithm returns TRUE when the set without the redundant relation is identified as compatible by recursive application of the algorithm (Lines 11–14), which is correct according to Corollary 5.12.
4. In all other cases, the algorithm returns FALSE (Line 15). □

The algorithm, however, also operates *conservatively*. If the approach finds redundant relations, such that a consistency relation set can be reduced to a set of independent consistency relation trees, the set is proven compatible, as we have shown by proof. If the approach is not able to find such relations, the set may still be compatible, but the approach is not able to prove that. Conceptually, this can be due to the fact that there are compatibility-preserving relations that do not fulfill the definition of left-equal redundancy, or because our independence definition is too restrictive. Furthermore, an actual technique to identify left-equal redundant relations may not be able to find all of them automatically for undecidability reasons, as we see later at the practical approach.

> **Theorem 5.14** (Compatibility Algorithm Conservativeness)
> *Algorithm 5.1 operates conservatively, which means that it is correct but the given consistency relation set $\mathbb{CR}$ is not necessarily incompatible if it returns* FALSE.

Proof. We know that the algorithm is correct due to Theorem 5.13. Additionally, it is easy to find examples for which the algorithm cannot prove compatibility, although the relations are compatible. Let us assume a consistency relation CR. Then we construct a consistency relation CR' by taking CR, adding an arbitrary class C to the left-hand side class tuple of the relation, and constructing the relation elements by taking the ones in CR, each complemented by all instances of C. Then $\{CR, CR'\}$ is, by construction, compatible, but the two relations are neither independent or a consistency relation tree, as they relate the same classes, nor are they redundant according to Definition 5.9, because the left-side class tuples are not equal. □

The example given in the proof for conservativeness shows that the strictness of our definition for left-equal redundancy (Definition 5.9) can prevent the algorithm from proving compatibility. We will, however, see in the evaluation in Section 9.1 that it is still sufficient in realistic cases, although such special cases as discussed in the proof are not supported.

In the following, we discuss how such an approach can be operationalized. First, we discuss at the example of QVT-R how transformations can be represented in a graph-based structure, which conforms to our formal notion and allows to check whether the structure is an independent set of consistency relation trees. Second, we present an approach for finding consistency relations that are left-equal redundant by means of a Satisfiability Modulo Theories (SMT) solver applied to the constraints defined in QVT-R transformations.

5.4. A Practical Approach to Prove Compatibility

We have presented a formal and proven correct approach for validating compatibility of consistency relations in the previous section. It comprises the reduction of a given set of consistency relations by removing redundant relations to result in independent consistency relation trees. In this section, we propose an algorithm that turns the formal approach into an operational procedure. For the most part, this approach is based on results developed and described in detail in the Master's thesis of Pepin [Pep19], which was supervised by the author of this thesis, and published in a report [Kla+20].

We call the process of removing redundant relations from a consistency relation set to generate independent consistency relation trees *decomposition*. A decomposition procedure requires a representation of consistency relations present in actual model transformations that allows to validate their redundancy, more specifically the property of left-equal redundancy (see Definition 5.9). We have decided to employ the transformation language QVT-R for the operationalization. First, QVT-R is standardized [QVT] and well researched. Second, it provides a level of abstraction at which consistency relations are explicitly represented. In contrast, imperative languages would first require to extract consistency relations from their implicit specification as the image of the transformation rules.

In the following, we first present a mapping between the formalization of the previous sections to the QVT-R transformation language through the use of *predicates*. We then propose a fully automated decomposition procedure that takes a set of QVT-R transformations, called a *consistency specification*, as an input and removes redundant consistency relations as far as possible. To find a redundant relation, the procedure identifies an alternative concatenation of consistency relations relating the same classes, according to Definition 5.9, and performs a *redundancy test* with respect to that alternative concatenation. We explicitly separate the identification of candidates for the alternative concatenation from the redundancy test to assert exchangeability of the redundancy test approach.

5.4.1. Practical Specification of Consistency Relations

In Subsection 4.1.1, we have discussed the distinction of intensional and extensional specifications of consistency. We have used an extensional specification for formalizing consistency relations in Definition 4.18. Developers, however, define transformations with intensional specifications of the constraints that have to hold. In relational transformation languages, such as QVT-R, they define consistency as a set of conditions that models must fulfill. Such conditions are expressed with metamodel elements, like attributes and references. For example, an employee and a resident are considered consistent if their *name* values are equal.

Conditions represent predicates, i.e., Boolean-valued filter functions. Consistency relations are then defined as sets of condition element pairs for which the predicate evaluates to TRUE. In Subsection 4.1.1, we have already shown that this type of specification has equal expressiveness and can be transformed into an extensional specification. We define such a predicate based on combinations of properties, selected from each metamodel, which we introduce in the following.

Definition 5.10 (Property Set)
A property set $\mathbb{P}_C$ for a class C is a subset of properties of C, i.e., $\mathbb{P}_C = \{P_{C,1}, \ldots, P_{C,n}\}$ such that $P_{C,i} \in C$.

A property set represents a selection of properties of a class that are relevant for the definition of a predicate in order to distinguish consistent and non-consistent condition elements. For a consistency relation, not all properties of a class may be relevant and thus need to be considered. In a case of an extensional specification at the level of classes rather than properties, such as the one defined in Definition 4.18, this is expressed by enumerating all objects with all possible values of the irrelevant properties. Thus, expressing the relations at the level of classes or properties have equal expressiveness.

Definition 5.11 (Tuple of Property Sets)
For a class tuple $\mathfrak{C} = \langle C_1, \ldots, C_n \rangle$, we denote a tuple of property sets for every class as a property tuple $\mathfrak{P}_{\mathfrak{C}}$:

$$\mathfrak{P}_{\mathfrak{C}} = \langle \mathbb{P}_{C_1}, \ldots, \mathbb{P}_{C_n} \rangle = \langle \{P_{C_1,1}, \ldots, P_{C_1,m}\}, \ldots, \{P_{C_n,1}, \ldots, P_{C_n,k}\} \rangle$$

Since condition elements in consistency relations consist of objects that instantiate multiple classes, property set tuples generalize the use of property sets to class tuples.

Definition 5.12 (Property Value Set)

A property value set $\mathbb{p}_C$ for a property set $\mathbb{P}_C = \{P_{C,1}, \ldots, P_{C,n}\}$ is a set in which each property in $\mathbb{P}_C$ is instantiated, i.e., $\mathbb{p}_C = \{p_{C,1}, \ldots, p_{C,n}\}$ with $p_{C,i} \in I_{P_{C,i}}$. Analogously, a tuple of property value sets is built from a tuple of property sets by instantiating each property set in it.

A property value set is a subset of property values of an object o that instantiates C, like a property set is a subset of properties of a class C. Such a property value set represents the information of an object o that is relevant for consistency according to a specific consistency relation.

Definition 5.13 (Predicate)

A predicate for two class tuples $\mathfrak{C}_l = \langle C_{l,1}, \ldots, C_{l,n} \rangle$ and $\mathfrak{C}_r$ is a triple $\pi = \langle \mathfrak{P}_{\mathfrak{C}_l}, \mathfrak{P}_{\mathfrak{C}_r}, \text{F}_\pi \rangle$ where $\mathfrak{P}_{\mathfrak{C}_l} = \langle \mathbb{P}_{C_{l,1}}, \ldots, \mathbb{P}_{C_{l,n}} \rangle$ (resp. $\mathfrak{P}_{\mathfrak{C}_r}$) is a tuple of property sets of $\mathfrak{C}_l$ (resp. $\mathfrak{C}_r$) and F_π is a Boolean-valued function for instances of $\mathfrak{P}_{\mathfrak{C}_l}$ and $\mathfrak{P}_{\mathfrak{C}_r}$, i.e.,

$$\text{F}_\pi : I_{\mathfrak{P}_{\mathfrak{C}_l}} \times I_{\mathfrak{P}_{\mathfrak{C}_r}} \rightarrow \{\text{TRUE}, \text{FALSE}\}$$

For better readability, we define the property collection $\mathbb{P}_\pi$ of a predicate $\pi = (\mathfrak{P}_{\mathfrak{C}_l}, \mathfrak{P}_{\mathfrak{C}_r}, \text{F}_\pi)$ as the union of all properties in that predicate:

$$\mathbb{P}_\pi := \bigcup_j \mathbb{P}_{C_{l,j}} \cup \bigcup_k \mathbb{P}_{C_{r,k}}$$

The definition of a predicate requires the selection of properties of the classes within the class tuples related by a consistency relation CR and the definition of a function F_π that defines whether two instances of these properties are considered consistent. If F_π evaluates to TRUE for given property values of two object tuples, they match the predicate and are considered consistent, i.e., they represent the condition elements of a consistency relation pair according to Definition 4.18.

Predicates thus model how consistency relations are defined in model transformation languages in terms of conditions to evaluate for object tuples, i.e., condition elements, rather than enumerating all consistent pairs of condition elements. We define when we consider property values to match objects and then derive how consistency relations can be defined by predicates.

Definition 5.14 (Property Matching)
Let $\mathbb{p}_C = \{p_{C,1}, \ldots, p_{C,n}\}$ be a property value set. We say that:

$$\mathbb{p}_C \textit{ matches } o :\Leftrightarrow o \in I_C \wedge \forall\, p_{C,i} \in \mathbb{p}_C : p_{C,i} \in o$$

Similarly, let $\mathfrak{p}_{\mathfrak{C}} = \langle \mathbb{p}_{C_1}, \ldots, \mathbb{p}_{C_n} \rangle$ be a tuple of property value sets and $\mathfrak{o} = \langle o_1, \ldots, o_n \rangle$ a tuple of objects. We say that:

$$\mathfrak{p}_{\mathfrak{C}} \textit{ matches } \mathfrak{o} :\Leftrightarrow \forall\, \mathbb{p}_{C_i} \in \mathfrak{p}_{\mathfrak{C}} : \mathbb{p}_{C_i} \textit{ matches } o_i$$

Definition 5.15 (Predicate-Based Consistency Relation)
Let $\mathbb{c}_l$ and $\mathbb{c}_r$ be two conditions for two class tuples $\mathfrak{C}_{\mathbb{c}_l}$ and $\mathfrak{C}_{\mathbb{c}_r}$. Let Π be a set of predicates for $\mathfrak{C}_{\mathbb{c}_l}$ and $\mathfrak{C}_{\mathbb{c}_r}$. A Π-based consistency relation CR_Π is a subset of pairs of condition elements:

$$\begin{aligned} CR_\Pi :=\ & \{\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in \mathbb{c}_l \times \mathbb{c}_r \mid \forall \langle \mathfrak{P}_{\mathfrak{C}_{\mathbb{c}_l}}, \mathfrak{P}_{\mathfrak{C}_{\mathbb{c}_r}}, \mathrm{F}_\pi \rangle \in \Pi : \\ & \quad \exists\, \mathbb{p}_{\mathfrak{C}_{\mathbb{c}_l}} \in I_{\mathfrak{P}_{\mathfrak{C}_{\mathbb{c}_l}}}, \mathbb{p}_{\mathfrak{C}_{\mathbb{c}_r}} \in I_{\mathfrak{P}_{\mathfrak{C}_{\mathbb{c}_r}}} : \\ & \big(\mathfrak{p}_{\mathfrak{C}_{\mathbb{c}_l}} \textit{ matches } \mathfrak{c}_l \wedge \mathfrak{p}_{\mathfrak{C}_{\mathbb{c}_r}} \textit{ matches } \mathfrak{c}_r \wedge \mathrm{F}_\pi(\mathfrak{p}_{\mathfrak{C}_{\mathbb{c}_l}}, \mathfrak{p}_{\mathfrak{C}_{\mathbb{c}_r}}) = \textsc{true}\big)\} \end{aligned}$$

The construction of consistency relations by means of predicates is comparable to the one discussed in Subsection 4.1.1 at the level of models. Definition 5.15 extends that construction to fine-grained consistency relations. It expresses how consistency relations enumerating consistent object tuples are defined by means of predicates. The construction of the consistency relation fully amounts to the evaluation of the predicate function.

Example 5.6. *We construct a consistency relation CR_{PR} based on predicates between persons and residents, according to Figure 5.1. CR_{PR} ensures that the name of a resident concatenates the firstname and lastname of a person and that both have the same address. CR_{PR} involves one class in each metamodel, resulting in two class tuples $\mathfrak{C}_P = \langle C_{Person} \rangle$ and $\mathfrak{C}_R = \langle C_{Resident} \rangle$. Two predicates need to represent consistency conditions, which are equal names and equal*

```
import M1 : 'path_m1.ecore';
import M2 : 'path_m2.ecore';

transformation T(M1, M2) {
   [top] relation R1 {
      [variable declarations]
      domain M a : A { πM }
      domain N b : B { πN }
      [when { PRECONDITION }] [where { INVARIANT }]
   }

   [top] relation R2 { ... }
}
```

Listing 5.1: Simplified structure of a QVT-R transformation. Adapted from [Kla+20, Lst. 1].

addresses. The first predicate considers firstname and lastname of a person and name of a resident, so $\mathfrak{P}_{\mathfrak{C}_P,1} = \langle\{\mathit{firstname}, \mathit{lastname}\}\rangle$ *and* $\mathfrak{P}_{\mathfrak{C}_R,1} = \langle\{\mathit{name}\}\rangle$*. Similarly,* $\mathfrak{P}_{\mathfrak{C}_P,2} = \langle\{\mathit{address}\}\rangle$ *and* $\mathfrak{P}_{\mathfrak{C}_R,2} = \langle\{\mathit{address}\}\rangle$*. The functions of the predicate, shortly denoting name as n, firstname as fn, lastname as ln, as well as address of a person as* a_P *and of a resident as* a_R*, are:*

$$F_{\pi,1}(\langle\{fn, ln\}\rangle, \langle\{n\}\rangle) = \begin{cases} \mathit{TRUE}, & \text{if } n = fn + \text{"␣"} + ln \\ \mathit{FALSE}, & \text{otherwise} \end{cases}$$

$$F_{\pi,2}(\langle\{a_P\}\rangle, \langle\{a_R\}\rangle) = \begin{cases} \mathit{TRUE}, & \text{if } a_P = a_R \\ \mathit{FALSE}, & \text{otherwise} \end{cases}$$

CR_{PR} *is a* Π*-based consistency relation where* Π *is the set of the two predicates for names and addresses* $\{\langle\mathfrak{P}_{\mathfrak{C}_P,1}, \mathfrak{P}_{\mathfrak{C}_R,1}, F_{\pi,1}\rangle, \langle\mathfrak{P}_{\mathfrak{C}_P,2}, \mathfrak{P}_{\mathfrak{C}_R,2}, F_{\pi,2}\rangle\}$.

We have decided to use QVT-R as the relational language of the QVT standard [QVT] for implementing the formal approach for validating compatibility. The defined relations can be interpreted as predicates defining Π-based consistency relations. The language can be executed in *checkonly* mode to check models for fulfillment of consistency relations, or in *enforce* mode to repair consistency in a specified direction if not all relations are fulfilled. The relevant parts of the structure of a QVT-R transformation are as follows and also exemplified in Listing 5.1.

```
fstn:String; lstn:String; inc:Integer;

domain person p:Person {
   firstname = fstn, lastname = lstn, income = inc
};
domain employee e:Employee {
   name = fstn + '␣' + lstn, salary = inc
};
```

Listing 5.2: Two QVT-R domains, each with one domain pattern. Adapted from [Kla+20, Lst. 2].

A QVT-R `transformation` receives models, which conform to defined meta-models, and checks or repairs their consistency. Each `transformation` is composed of `relations`, which define when objects of both models are considered consistent. These relations are only invoked if they are prefixed by the `top` keyword, if they belong to the precondition (`when`) of a relation to be invoked, or if they belong to the invariant (`where`) of a relation that was already invoked. The QVT-R mechanism for checking consistency is based on pattern matching. The relations between information in the different models are represented by variables assigned to class properties. These variables contain values that must remain consistent from one object to another. To consider models consistent, there must exist some assignment that matches all patterns at the same time.

More precisely, each QVT-R `relation` contains two `domains`, which in turn contain *domain patterns*. In QVT terminology, a domain pattern is a variable instantiating a class. This variable can take values that are constrained by conditions on its properties, known as *property template items*. These conditions are expressed by OCL constraints [OCL]. We give an example for the domains of persons and employees according to the running example in Listing 5.2, in which each domain has one pattern. These patterns filter *Person* objects with three property template items for *firstname*, *lastname*, and *income*, and *Employee* objects with two property template items for *name* and *salary*, respectively. For two objects to be consistent, there must exist values of *fstn*, *lstn*, and *inc* that match property values of these objects, thus ensuring that the employee *name* equals the concatenation of the *firstname* and the *lastname* of the person and that both have the same *income*. If objects are inconsistent, e.g., if the person and the employee have different incomes, then there is no such variable assignment. The QVT-R transformations for all three relations of the running example introduced with Figure 5.1 are depicted in Listing 5.3.

In *checkonly* mode, QVT-R evaluates the existence of a value that fulfills all property template items in domain patterns. These patterns can be regarded as predicates. To transfer QVT-R relations into our formalism, each relation is translated into one or multiple predicates. A predicate is formed by the properties that are bound to the same QVT-R variables, because having QVT-R variables in common means that values of these properties are interrelated and thus need to fulfill some consistency constraints. The properties of each domain form one of the property sets of a predicate. Extracting the OCL constraints of the property template items generates the predicate function. The property sets together with the predicate function represent a predicate. We subsequently present a formal construction of predicates from QVT-R and their representation in a graph.

5.4.2. Consistency Relations Represented as Graphs

In the following, we introduce the decomposition procedure for proving compatibility, which relies on an algorithmic way to detect redundant consistency relations. We have defined the notion of left-equal redundancy for extensionally specified consistency relations in Definition 5.9. That notion is based on classes, whereas predicate-based consistency relations are defined for properties. We have, however, already discussed that both kinds of specification have equal expressiveness. Comparing predicate-based consistency relations of different transformations to evaluate redundancy is what we call a *redundancy test*.

Consistency specifications induce a graph of class properties, which are related by edges that are labeled with the predicates that define the consistency relations. Such a graph representation enables the application of graph algorithms to identify independent and redundant consistency relations. The decomposition procedure thus creates such a graph, denoted as a *property graph*, out of QVT-R transformations and detects redundant relations in that graph. It represents properties and predicates as a hypergraph with labeling.

```
import personMM : 'personmm.ecore';
import employeeMM : 'employeemm.ecore';
import residentMM : 'residentmm.ecore';

transformation PersonEmployee(person: personMM, employee: employeeMM) {
   top relation PE {
      fstn:String; lstn:String; inc:Integer;

      domain person p:Person {
         firstname = fstn, lastname = lstn, income = inc
      };
      domain employee e:Employee {
         name = fstn + '␣' + lstn, salary = inc
      };
   }
}

transformation PersonResident(person: personMM, resident: residentMM) {
   top relation PR {
      fstn:String; lstn:String; addr:String;

      domain person p:Person {
         firstname = fstn, lastname = lstn, address = addr
      };
      domain resident r:Resident {
         name = fstn + '␣' + lstn, address = addr
      };
   }
}

transformation EmployeeResident(employee: employeeMM, resident: residentMM) {
   top relation ER {
      n:String; ssn:Integer;

      domain employee e:Employee {
         name = n, socsecnumber = ssn
      };
      domain resident r:Resident {
         name = n, socsecnumber = ssn
      };
   }
}
```

Listing 5.3: Three binary QVT-R transformations forming a consistency specification, based on the relations in Figure 5.1. Adapted from [Kla+20, Fig. 10].

Definition 5.16 (Property Graph)

Let $\mathbb{CR} = \{CR_1, \dots, CR_n\}$ be a set of consistency relations where each consistency relation CR_i is based on a set of predicates Π_i. A property graph is a couple $\mathcal{M} = \langle \mathcal{H}, \textsc{l} \rangle$, such that $\mathcal{H} = \langle V_{\mathcal{H}}, E_{\mathcal{H}} \rangle$ is a hypergraph and $\textsc{l}$ is a hyperedge labeling: $V_{\mathcal{H}}$ is the set of vertices, i.e., the union of properties in all predicates:

$$V_{\mathcal{H}} := \bigcup_{i=1}^{n} \bigcup_{\pi \in \Pi_i} \mathbb{P}_\pi$$

$E_{\mathcal{H}}$ is the set of hyperedges, i.e., $E_{\mathcal{H}} \subseteq \mathcal{P}(V_{\mathcal{H}}) \setminus \{\emptyset\}$. Each hyperedge consists of the properties of one predicate:

$$E_{\mathcal{H}} := \bigcup_{i=1}^{n} \bigcup_{\pi \in \Pi_i} \{\mathbb{P}_\pi\}$$

$\textsc{l}$ labels each hyperedge with its corresponding predicate function:

$$\forall\, i \in \{1, \dots, n\} : \forall\, \pi = \langle \mathfrak{P}_{\mathfrak{C}_l}, \mathfrak{P}_{\mathfrak{C}_r}, \textsc{f}_\pi \rangle \in \Pi_i : \textsc{l}(\mathbb{P}_\pi) = \textsc{f}_\pi$$

A property graph groups properties that are used in the same predicate. Each hyperedge with its labeling represents a predicate, which, in turn, represents a consistency relation. Thus, such a graph is useful for detecting independent sets of consistency relations and potential redundancies. When there are sets of hyperedges that do not share any vertices, they relate independent sets of properties. According to Definition 5.5, the consistency relations represented by the hyperedges are independent. On the contrary, if multiple sequences of hyperedges relate the same properties, the represented consistency relations form a cycle and may thus be incompatible or redundant.

A property graph needs to be a hypergraph, because a predicate can relate more than two properties, so an edge must be able to relate more than two vertices. The consistency relation CR_{PE} of the running example relates an employee's *name* to the concatenation of the *firstname* and *lastname* of a person and thus contains three properties. We depict the hypergraph for the running example in Figure 5.15. In the following, we discuss the construction of a property graph from QVT-R transformations. The identification of redundancies in the represented relations is part of the subsequent subsection.

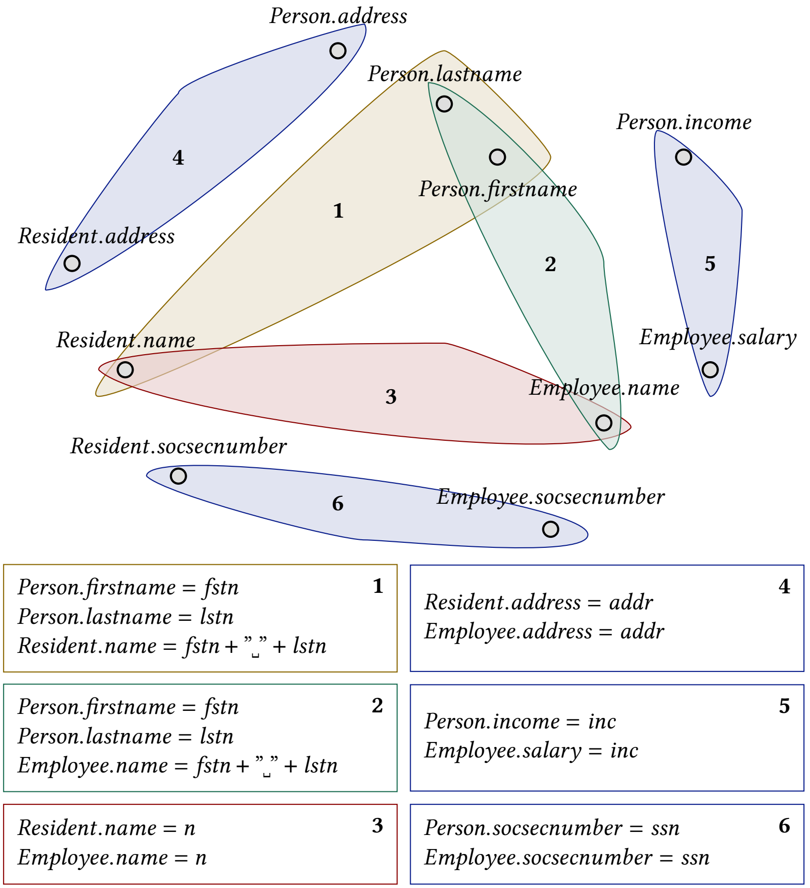


Figure 5.15.: Property graph for the QVT-R example in Listing 5.3 based on relations in Figure 5.1. Hyperedges are represented as shaded areas. Constraints for the predicate functions are annotated in boxes. Adapted from [Kla+20, Fig. 11].

The construction of the property graph for a given set of QVT-R transformations requires each of them to be processed. Since transformations are not executed but only transformed into a property graph, the processing order is not relevant. Each transformation consists of a set of QVT-R relations, of which each usually only defines consistency for small parts of the metamodels. Those relations depend on each other and can thus not be processed in arbitrary order. Only those relations that may be invoked during the execution of transformations need to be considered, which could be derived from a call graph. While top-level relations are always invoked during execution, other relations are only invoked in `where` or `when` clauses of other relations similar to function calls. Since `when` and `where` clauses are dual to each other, we restrict ourselves to relations that are invoked in `where` clauses. Then, starting from top-level relations, relevant relations can simply be identified by a depth-first traversal.

The property graph construction starts with an empty graph. For every processed QVT-R relation, vertices and a hyperedge may be added. Each QVT-R relation needs to be translated into a set of predicates, which are represented by labeled hyperedges, in accordance with Definition 5.16. As an example, we consider the relation *PE* of our running example in Listing 5.3, which relates the domains for persons and employees. For each domain of a relation, the class tuples of the predicates are specified in the domain patterns. In the example, these class tuples are $\mathfrak{C}_{person} = \langle C_{Person} \rangle$ and $\mathfrak{C}_{employee} = \langle C_{Employee} \rangle$. Each class in each class tuple is associated with a set of property template items. A property template item relates a property to an OCL expression. For example, the property template item $name = fstn + "␣" + lstn$ defines that the property *name* must match the OCL expression $fstn + "␣" + lstn$. The OCL expressions, in turn, contain QVT-R variables, such as *fstn* and *lstn*. Predicates are supposed to relate those properties that actually share a consistency relation, i.e., that are actually put into relation by the QVT-R relation. Such a relation is only given if two properties are related by the same QVT-R variables, because in such a case a value assignment to that variable must satisfy the property template items of both properties. In such a case, a hyperedge is created and labeled with a function that realizes the conditions of the property template item. For example, *Person.firstname*, *Person.lastname* and *Employee.name* are related by the QVT-R variables *fstn* and *lstn*, thus a hyperedge is generated between them. In contrast, constraints on *Employee.salary* and *Employee.name* are independent, because the property template items relate them to disjoint sets of QVT-R variables. Thus consistency of one does not depend on consistency of the other. In addition to property template items, OCL expressions relating properties occur in `when` and `where` clauses, of which we, again, focus on invariants of `where` clauses. Like for property template items, properties related by shared QVT-R variables in these clauses have to be grouped into a hyperedge.

Algorithm 5.2 Merge to predicates. Adapted from [Kla+20, Alg. 1].

procedure $\textsc{MergeProperties}(\{\langle\{p\}, V_{\{p\}}\rangle\})$
  $\mathit{stopMerge} \leftarrow \textsc{true}$
  $\mathit{entries}[] \leftarrow [\langle\{p\}, V_{\{p\}}\rangle]$ ▹ Convert input set to sequence
  do
    $\mathit{stopMerge} \leftarrow \textsc{true}$
    $\mathit{results} \leftarrow \emptyset$
    while $\mathit{entries}[] \neq []$ **do**
      $\mathit{ref} := \langle\mathbb{P}_{\mathit{ref}}, V_{\mathbb{P}_{\mathit{ref}}}\rangle \leftarrow \mathit{entries}[0]$
      $\mathit{others} \leftarrow \mathit{entries}[1:]$
      $\mathit{entries}[] \leftarrow []$
      for $\langle\mathbb{P}, V_{\mathbb{P}}\rangle \in \mathit{others}$ **do**
        if $V_{\mathbb{P}} \cap V_{\mathbb{P}_{\mathit{ref}}} = \varnothing$ **then**
          $\mathit{entries}[] \leftarrow \mathit{entries}[] + \langle\mathbb{P}, V_{\mathbb{P}}\rangle$
        else
          $\mathit{stopMerge} \leftarrow \textsc{false}$
          $\mathit{ref} \leftarrow \langle\mathbb{P} \cup \mathbb{P}_{\mathit{ref}}, V_{\mathbb{P}} \cup V_{\mathbb{P}_{\mathit{ref}}}\rangle$
        end if
      end for
      $\mathit{results} \leftarrow \mathit{results} \cup \{\mathit{ref}\}$
    end while
    $\mathit{entries}[] \leftarrow [\mathit{results}]$ ▹ Convert results to sequence
  while $\neg\mathit{stopMerge}$
  return $\mathit{set}(\mathit{entries}[])$ ▹ Convert sequence to output set
end procedure

Algorithm 5.2 expresses the sketched procedure of merging properties to predicates that finally represent hyperedges of the property graph. It manages couples, called *entries*, of properties and QVT-R variables. These entries denote that a set of properties is related by the according set of QVT-R variables. The algorithm starts with a set of couples, of which each couple $\langle\{p\}, V_{\{p\}}\rangle$ consists of a singleton $\{p\}$ that presents a property p and the QVT-R variables $V_{\{p\}}$ it is related to by its property template item. In each iteration, the algorithm chooses one reference entry and merges it with all other entries to which the intersection of their QVT-R variables is not empty.

The algorithm terminates when all sets of QVT-R variables are pairwise disjoint.

Example 5.7. *The relation PE of the QVT-R transformation PersonEmployee in Listing 5.3 contains five properties, which can be described with these entries:*

$\langle\{\mathit{firstname}\}, \{\mathit{fstn}\}\rangle, \langle\{\mathit{lastname}\}, \{\mathit{lstn}\}\rangle, \langle\{\mathit{income}\}, \{\mathit{inc}\}\rangle,$
$\langle\{\mathit{name}\}, \{\mathit{fstn}, \mathit{lstn}\}\rangle, \langle\{\mathit{salary}\}, \{\mathit{inc}\}\rangle$

After the execution of the algorithm, properties are merged into two sets:

$\langle\{\mathit{firstname}, \mathit{lastname}, \mathit{name}\}, \{\mathit{fstn}, \mathit{lstn}\}\rangle, \langle\{\mathit{income}, \mathit{salary}\}, \{\mathit{inc}\}\rangle$

Each entry delivered by the algorithm can be transformed into a hyperedge. To this end, the properties are grouped into two tuples according to the domains they originally belonged to. The predicate function is given by the conjunction of all OCL expressions associated with properties of the entry, i.e., property template items and invariants. For the subsequent identification of redundant relations, we only need to operate on this hypergraph rather than the original metamodels or QVT-R transformations.

5.4.3. Decomposition of Consistency Relations

The decomposition procedure for proving compatibility of consistency relations aims at removing redundant relations until, in case of success, the remaining relations form sets of independent consistency relation trees. For a property graph $\mathcal{M} = \langle \mathcal{H}, \textsc{l} \rangle$, this is achieved by removing the hyperedges of $\mathcal{H}$ that represent redundant consistency relations until no further redundant relations can be found. Redundancy according to Definition 5.9 is given if for a consistency relation an alternative concatenation of consistency relations that relates the at least partly same classes does not restrict consistency. In terms of a graph, this means that there must be two paths between the same properties. Independence of consistency relations is then given by connected components of the hypergraph, because they represent the properties that are related by constraints involving the same QVT-R variables. According to Theorem 5.5, consistency relations are compatible if they are composed of independent, compatible subsets. Thus if compatibility can be shown for the relations in each connected component of the hypergraph, their union is also compatible.

While the hypergraph representation of predicates in consistency relations is well suited for reasons of expressiveness, the drawback of hypergraphs is the increased complexity of graph algorithms, such as graph traversal. We therefore replace the property graph with its dual, i.e., an equivalent simple graph, for the realization of the redundancy test. This dual graph contains the hyperedges of the property graph as vertices and edges between two vertices when their hyperedges in the property graph share at least one property. Figure 5.16 shows the dual of a property graph of the running example.

Definition 5.17 (Dual of a Property Graph)

Let $\mathcal{M} = (\mathcal{H}, \textsc{l})$ be a property graph. The dual of the property graph $\mathcal{M}$, denoted $\mathcal{M}^*$, is a tuple $\langle \mathcal{G}, \textsc{v}, \textsc{l} \rangle$ with a simple graph $\mathcal{G}$ and two functions $\textsc{v}$ and $\textsc{l}$ such that:

- $V_{\mathcal{G}} := E_{\mathcal{H}}$
- $E_{\mathcal{G}} := \{\{E_1, E_2\} \mid \forall \langle E_1, E_2 \rangle \in E^2_{\mathcal{H}} : E_1 \cap E_2 \neq \emptyset\}$
- $\forall \langle E_1, E_2 \rangle \in E_{\mathcal{G}} : \textsc{v}(\{E_1, E_2\}) = E_1 \cap E_2$

The function $\textsc{v}$ labels each edge $\{E_1, E_2\}$ in the dual with the set of properties that occur both in E_1 and E_2. Since the property graph and its dual have equal expressiveness, the property graph can be constructed out of the dual again. Given a dual $\mathcal{M}^* = \langle \mathcal{G}, \textsc{v}, \textsc{l} \rangle$, the property graph $\mathcal{M} = \langle \mathcal{H}, \textsc{l} \rangle$ can be built by defining $V_{\mathcal{H}} = \bigcup_{V \in V_{\mathcal{G}}} V$ and $E_{\mathcal{H}} = V_{\mathcal{G}}$.

Independence of consistency relations in the property graph is characterized by the existence of two (or more) subhypergraphs[1] such that there is no path (i.e., sequence of incident hyperedges) from one to the other. In the dual of the property graph, such a situation is represented by two subgraphs that are not connected to each other. This conforms to the notion of connected components, which are maximal subgraphs such that any two vertices are connected by a path of edges and reflects the notion of independence given in

[1] A subhypergraph of a hypergraph $\mathcal{H} = (V_{\mathcal{H}}, E_{\mathcal{H}})$ is a hypergraph $\mathcal{S} = \langle V_{\mathcal{S}}, E_{\mathcal{S}} \rangle$ such that $E_{\mathcal{S}} \subseteq E_{\mathcal{H}}$ and $V_{\mathcal{S}} = \bigcup_{E \in E_{\mathcal{S}}} E$

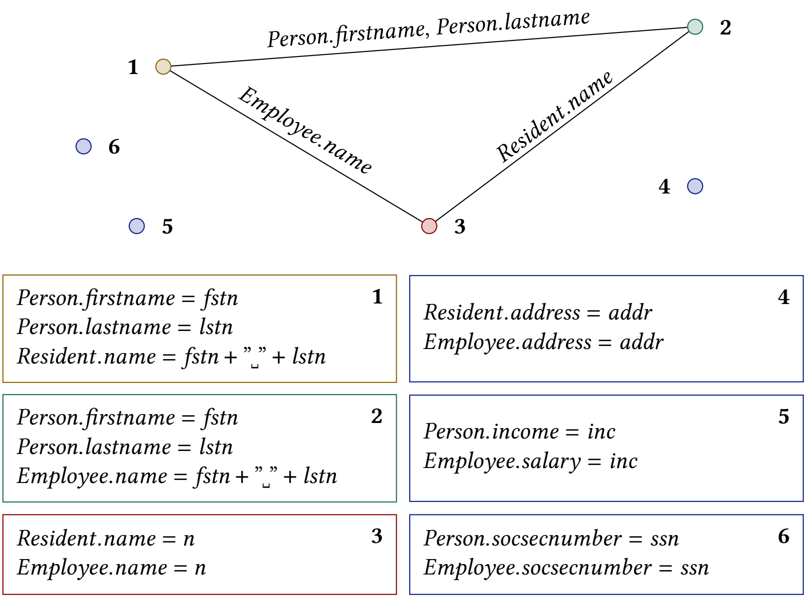


Figure 5.16.: Dual of the property graph for the QVT-R example in Listing 5.3 based on the relations in Figure 5.1. Adapted from [Kla+20, Fig. 12].

Definition 5.5. Per definition, each subgraph relates disjoint sets of properties, as otherwise an edge between two vertices that contain an intersection of these properties would exist. These property sets occur in independent sets of consistency relations, as otherwise there would be a vertex in the dual of the property graph for a hyperedge of the property graph that relates the properties that are linked by an OCL expression and according QVT-R variables. We use Tarjan's algorithm to compute the connected components of the dual of the property graph in linear time [Tar71]. These independent subgraphs can be processed independently, since their compatibility composes (see Theorem 5.5).

In addition to independence, Theorem 5.6 stating that consistency relation trees are compatible also applies to the dual of the property graph. When there are no two paths relating the same classes or properties, respectively, the notion of a consistency relation tree is fulfilled, thus the represented consistency relations are inherently compatible. Consequently, if the dual of the property graph is only composed of independent trees, i.e., if it is a *forest*, it is inherently compatible.

Finally, Corollary 5.12 has shown that adding left-equal redundant consistency relations to a compatible consistency relation set preserves its compatibility. According to Definition 5.9 for redundancy, we consider a predicate and its representing hyperedge, respectively, redundant if there is another concatenation of predicates that are always fulfilled if the redundant one is fulfilled. In the hypergraph, this conforms to an alternative sequence of hyperedges that represent those predicates, which relates the same properties as the possibly redundant one. In our operationalization, we only consider the case when the exact same classes are related by both the possibly redundant and the alternative concatenation of predicates, although the definition only requires the classes at the left side to be equal. The existence of such an alternative path is, however, only a necessary but not a sufficient condition. The predicates must also relate the properties in the same way, as, for example, one predicate may ensure that two string attributes are equal, whereas an alternative sequence of predicates only ensures that they have the same length. This is the reason why we perform a redundancy test for redundancy candidates given by such an alternative path, which we explain in the subsequent subsection.

An alternative path for a hyperedge E, which represents a predicate in the property graph, is a sequence of pairwise incident hyperedges, of which the first and last edge are incident to E. In the dual of the property graph, these hyperedges are represented by vertices. Thus, in the dual such an alternative sequence is given by a cycle including the vertex E. Let $[E, E_1, \ldots, E_n, E]$ be the vertex sequence of such a cycle, then $[E_1, \ldots, E_n]$ is the alternative path. The generation of redundant paths amounts to the enumeration of pairs $\langle E, E[]_i \rangle$, where E is a possibly redundant predicate, i.e., a vertex in the dual of the property graph, and $E[]_i$ is an alternative sequence of predicates that may replace E. There may be multiple such possible alternative paths for a single predicate, thus all simple cycles in the dual of the property graph need to be considered. The problem of finding all simple cycles in an undirected graph is called *cycle enumeration*.

Algorithm 5.3 implements the enumeration of alternative paths for predicates and their removal in case they are redundant. The implementation of identifying a candidate predicate as actually redundant within a cycle is assumed to be available as an external function IsRedundant. As discussed

Algorithm 5.3 Predicates removal. Adapted from [Kla+20, Alg. 2].

```
procedure RemoveRedundantPredicates(Dual 𝓜*, pred ∈ V_𝓜*)
    {base_1, ..., base_n} ← PatonAlgorithm(𝓜*)
    foundCycles ← {base_1}
    currentCycles ← ∅, currentCycles* ← ∅
    for base ∈ {base_2, ..., base_n} do
        for foundCycle ∈ foundCycles do
            newCycle ← foundCycle ⊕ base
            if foundCycle ∩ base ≠ ∅ then
                currentCycles ← currentCycles ∪ {newCycle}
            else
                currentCycles* ← currentCycles* ∪ {newCycle}
            end if
        end for
        // Remove non-simple cycles from currentCycles
        for cycle_1, cycle_2 ∈ currentCycles do
            if cycle_2 ⊂ cycle_1 then
                currentCycles ← currentCycles \ {cycle_1}
                currentCycles* ← currentCycles* ∪ {cycle_1}
            end if
        end for
        // New valid cycles are in currentCycles ∪ {base}
        for cand ∈ currentCycles ∪ {base} do
            if pred ∈ cand ∧ IsRedundant(pred, cand) then
                remove pred and its incident edges from 𝓜*
                break
            end if
        end for
        foundCycles ← foundCycles ∪ currentCycles
        foundCycles ← foundCycles ∪ currentCycles* ∪ {base}
        currentCycles ← ∅, currentCycles* ← ∅
    end for
end procedure
```

before, this allows us to plug in different implementations for the redundancy test, of which we depict one in the subsequent subsection. The algorithm is mainly concerned with the enumeration of alternative paths.

The algorithm relies on the computation of a *cycles basis*, which is a set of simple cycles from which all other simple cycles of the graph can be derived by combination. This cycle basis is computed using Paton's algorithm [Pat69]. For a given predicate, the enumeration processes each cycle from the cycle basis and merges it with all cycles that have been processed so far. Every cycle is represented as a set of edges. We denote the symmetric difference with the $\oplus$ sign, i.e., $A \oplus B$ is the set of edges that are in A or in B but not in both. The set *foundCycles* contains all linear combinations of cycles that have been processed so far. Merged with cycles of the basis $base_1, \ldots, base_n$, these linear combinations are used to merge more than two cycles of the basis. In each iteration of Algorithm 5.3, processing a new cycle *base* from the cycle basis, new simple cycles are in $currentCycles \cup \{base\}$. Edge-disjoint or non-simple cycles are stored in $currentCycles^*$.

The redundancy test is performed in Line 21 whenever new cycles are generated. It checks for the given predicate *pred* whether one of the newly generated cycles is redundant, i.e., whether it contains *pred* and whether *pred* can be replaced by the concatenation of other predicates. If the test succeeds for an alternative sequence of predicates, the candidate can be removed. The algorithm then proceeds with further possibly redundant predicates. It terminates as soon as all predicates have been tested. If the connected component of the graph becomes a tree after a predicate removal, the dual of the connected component does not contain cycles anymore, thus no further redundancy tests have to be performed. In the following, we discuss how such a redundancy test can be realized.

5.4.4. Redundancy Check for Consistency Relations

We have so far considered the redundancy test of predicates in the decomposition procedure as a black box, which can be realized by any approach that is able to prove redundancy of predicates. This fosters independent reuse of the proposed decomposition procedure and the redundancy test to be presented in the following. Algorithm 5.3 contains the function IsRedundant that needs to realize this check.

Since OCL expressions have equal expressiveness than fist-order logic, reasoning about satisfiability of OCL constraints is undecidable [BKS02]. Deciding whether a predicate is redundant reduces to deciding satisfiability, which is why no strategy that always decides redundancy can be defined. In the following, we first discuss how predicates can be generally compared to prove compatibility. We then present an approach that translates OCL constraints of the predicates into first-order formulae and applies a theorem prover. Finally, we discuss the limitations of the approach especially arising from the translation to first-order logic and the use of a theorem prover.

A redundancy test takes a couple $\langle E, [E_1, \ldots, E_n] \rangle$ and returns TRUE whenever the predicate E is proven to be redundant to the sequence of predicates $[E_1, \ldots, E_n]$. Redundancy as defined in Definition 5.9 requires the set of consistency relations, which are defined by the predicates, to be equivalent with and without the redundant relation. This especially means that removing the redundant relation must not weaken consistency, i.e., it must not lead to models being considered consistent without that relation that are not considered consistent with that relation. This is equivalent for a property graph, in which a redundant predicate may not restrict consistency by considering a model with specific property values inconsistent that are considered consistent by an alternative sequence of predicates. A predicate E can thus only be removed if all instances matching the predicate also match predicates $[E_1, \ldots, E_n]$. In fact, Definition 5.9 limits redundancy to relations in which the left-side classes are equal. We do, however, only consider relations between the same sets of properties, thus being restricted to relations between the same sets of classes anyway.

In consequence, a redundancy test realizes the comparison of two sets of instances of models or, in particular, property values. A predicate can, however, be fulfilled by an infinite number of property values, i.e., condition elements in terminology of consistency relations, such as consistency of person incomes and employee salaries by an infinite number of integer pairs. An extensional element-wise comparison is thus generally impossible.

For that reason, we consider the intensional specification of consistency relations by means of OCL constraints. These constraints are annotated to the property graph as hyperedge labels. The redundancy test can thus be realized by a static analysis of these labels and QVT-R relation conditions in `when` and `where` clauses. One such strategy is the transformation of OCL expressions into first-order logic and the reasoning about the resulting first-order formulae [BKS02; BCD05]. We set up the first-order formulae such that they are valid, i.e., TRUE under every possible interpretation, whenever the redundancy test is positive. This transformation benefits from the availability of theorem proving tools for reasoning about first-order formulae.

Since first-order logic is generally undecidable, redundancy of a relation cannot be proven for every derived formula. Thus, the result quality of the decomposition procedure depends on the quality of the theorem prover. The transformation of OCL to logic formulae requires a representation of all constructs, such as arithmetic operations, strings, arrays, etc., in formulae. Objects, such as strings, floats, sequences, and others can be represented by theories of theorem provers. With theories, the satisfiability problem equates to assigning values to variables in first-order logic sentences such that their evaluation returns TRUE. For example, the formula $(a \times b = 6) \wedge (a + b > 0)$ is satisfiable given the assignment $\{a = 2, b = 3\}$. This extension is known as Satisfiability Modulo Theories (SMT). Formulae for the SMT problem are called *SMT instances*. *Theory-based theorem provers* provide built-in theories, to which we translate OCL constraints for our redundancy test.

The information that is necessary for a redundancy test is given by the predicates passed to the test. Let $E = \langle \mathfrak{P}_{\mathfrak{C}_l}, \mathfrak{P}_{\mathfrak{C}_r}, \text{F}_E \rangle$ be a predicate for two class tuples $\mathfrak{C}_l$ and $\mathfrak{C}_r$. During the construction of the property graph, a hyperedge composed of all properties in $\mathfrak{P}_{\mathfrak{C}_l}$ and $\mathfrak{P}_{\mathfrak{C}_r}$ is labeled with the description of the predicate function F_E. Such a predicate E can be replaced by a sequence of other predicates $[E_1, \ldots, E_n]$ if F_E evaluates to TRUE whenever $\text{F}_{E_1} \wedge \cdots \wedge \text{F}_{E_n}$ evaluates to TRUE. In that case, the removal of the consistency relation given by E does not weaken consistency, because it is fulfilled only when the relation given by the concatenation of $[E_1, \ldots, E_n]$ is fulfilled anyway. In consequence, E is redundant. This redundancy test can be encoded as a formula in the following way:

$$(\text{F}_{E_1} \wedge \cdots \wedge \text{F}_{E_n}) \Rightarrow \text{F}_E$$

This formula is a *Horn clause*. According to common terminology, we call terms at the left-hand side of the clause *facts* and the term at the right-hand side *goal*. The assignment of values to variables in the Horn clause also models the instantiation of properties, i.e., the assignment of property values. If the Horn clause is valid, the alternative sequence of predicates can replace the other predicate for every instance. Variables in Horn clauses are usually implicitly quantified universally. Predicate functions of OCL expressions, however, need to contain existentially quantified QVT-R variables, as the pattern matching of the expressions requires the existence of values for these variables.

Example 5.8. *Figure 5.16 depicts the dual of the property graph for the motivational example in Listing 5.3. It contains four connected components, of which three contain only one predicate. These three components are trivial trees, so compatibility for them is proven. The other component consists of three predicates and contains a cycle (*$[1, 2, 3]$*). Let* 3 *be the possibly redundant predicate. Then, the alternative combination of predicates is composed of* 1 *and* 2*. This leads to the following formula with facts* 1 *and* 2 *and goal* 3*:*

$$\begin{aligned}
&\big(\mathit{Person.firstname} = \mathit{fstn1} \land \mathit{Person.lastname} = \mathit{lstn1} \\
&\quad \land \mathit{Resident.name} = \mathit{fstn1} + \text{"␣"} + \mathit{lstn1} \\
&\land \mathit{Person.firstname} = \mathit{fstn2} \land \mathit{Person.lastname} = \mathit{lstn2} \\
&\quad \land \mathit{Employee.name} = \mathit{fstn2} + \text{"␣"} + \mathit{lstn2}\big) \\
&\Rightarrow \big(\exists\, n : (\mathit{Resident.name} = n \land \mathit{Employee.name} = n)\big)
\end{aligned}$$

QVT-R variables have been renamed to avoid conflicts, because they are no longer isolated as they were before in distinct QVT-R relations. The formula is valid and will be identified as such by an SMT solver. For that reason, predicate 3 *can be removed from the property graph and its dual. Since the component then only consists of two predicates and thus forms a tree, the represented consistency relations are compatible. Since all independent consistency relation sets, represented by the independent connected components of predicates, are compatible, the complete consistency specification is compatible.*

Whenever such a Horn clause is valid, i.e., TRUE under every interpretation, redundancy of the consistency relation represented by the predicate given as the clause goal is proven. The SMT solver takes the clause as an SMT instance and verifies its satisfiability whenever possible. Proving that a Horn clause H is valid is equivalent to proving that its negation $\neg H$ is unsatisfiable. Therefore, we actually let the SMT solver prove that the SMT instance $\mathrm{F}_{E_1} \land \cdots \land \mathrm{F}_{E_n} \land \neg \mathrm{F}_E$ is unsatisfiable. The complete process of the redundancy test is depicted in Figure 5.17. The solver can provide the following three results.

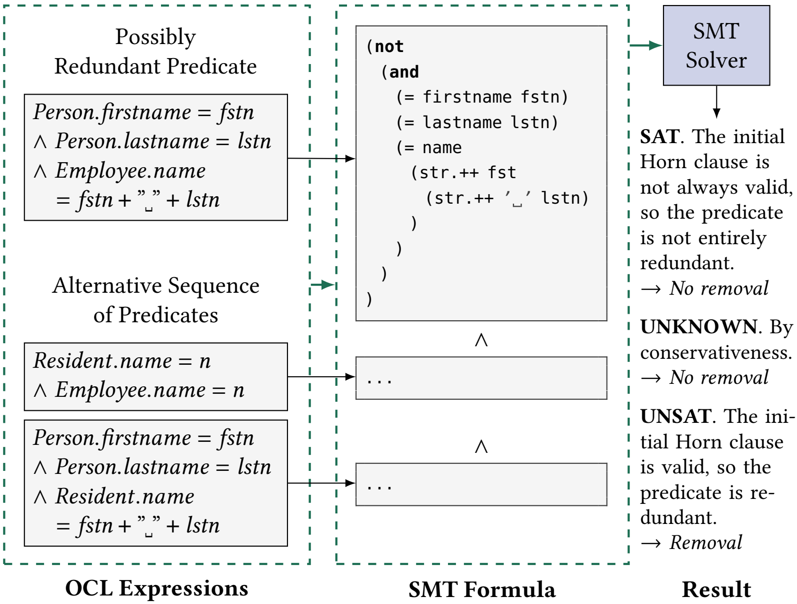


Figure 5.17.: Overview of the redundancy test from OCL expressions to the SMT solver results. Adapted from [Kla+20, Fig. 14].

Satisfiable: If $\neg H$ is satisfiable, H is not valid. An interpretation exists, i.e., an instantiation of properties, that fulfills the possibly redundant predicate but not the alternative sequence of predicates. Thus, the predicate is not redundant and cannot be removed.

Unsatisfiable: If $\neg H$ is unsatisfiable, H is valid. Thus, when the alternative sequence is fulfilled, the predicate is fulfilled as well. It is redundant and can be removed.

Unknown: First-order logic being undecidable, a theorem prover cannot evaluate satisfiability of all formulae, thus also returning *Unknown*. To ensure conservativeness, the redundancy test is considered negative. As a result, the predicate is not removed.

For the actual translation of OCL expressions in QVT-R relations into SMT instances, we refer to existing work on translating OCL to first-order formulae [BKS02] and, in particular, to our work presenting the specific translation for proving compatibility [Kla+20]. QVT-R uses a subset of OCL called *EssentialOCL* [QVT], which is a side-effect-free sublanguage that provides primitive data types, data structures and operations to express constraints on models. Several OCL constructs have a direct equivalent in theories of the theorem prover or can be mapped to a combination of primitive constructs. We employ the SMT-LIB specification, which is a standard that provides an input language for SMT solvers [BFT17], and the Z3 theorem prover [dB08] to realize the redundancy test. A complete reference of translated constructs has been developed in the Master's thesis of Pepin [Pep19].

In addition to undecidability of OCL, some OCL operations are said to be *untranslatable*, because no mapping to features of SMT solvers were found yet. Thus, some QVT-R relations cannot be processed automatically by the proposed decomposition procedure. For example, string operations like *toLower* and *toUpper* cannot be easily translated into logic formulae for SMT solvers without several used-defined axioms. Although decision procedures for such a case exist [Vea+12], they are not yet integrated into solvers.

In this section, we have discussed how the formal approach for proving compatibility as depicted in Algorithm 5.1 can be realized for QVT-R. We have defined a representation of consistency relations in graphs and explained how they can be derived from QVT-R transformations. We have discussed how a consistency relation tree and independent relation sets manifest in such a graph and how candidates for redundancies can be found in it. Finally, we have presented a redundancy test based on transforming OCL expressions of potentially redundant relations into Horn clauses that are validated by SMT solvers.

5.5. Summary

In this chapter, we have discussed the challenge regarding compatibility of consistency relations, which are encoded in transformations. We have derived and precisely defined a notion of compatibility and presented a formal approach that is able to validate it for given relations. The approach is proven to be correct. Based on the formal approach, we have developed a practical approach that validates compatibility of relations defined with QVT-R and OCL. We conclude this chapter with the following central insight.

Insight II.2 (Compatibility)

Transformations that are supposed to preserve contradictory consistency relations easily lead to problems when combining them to a network, because their relations cannot be fulfilled at the same time. The relations preserved by transformations should thus be *compatible*, i.e., they should not restrict consistency for elements such that no consistent set of models can be found by the transformation network. That notion of compatibility can be proven for given transformations by considering their preserved consistency relations, finding redundant relations, and removing them until only a tree of relations remains. Since we were able to prove that consistency relation trees are inherently compatible and removing redundant relations is compatibility-preserving, this approach is proven correct. Compatibility is a property of the network and not a single transformation, thus it cannot be achieved by construction of the individual transformations, but it can only be analyzed for a given transformation network.

6. Constructing Synchronizing Transformations

Transformations are the central artifacts of which a transformation network is composed. In Definition 4.7 and Subsection 4.4.3, we have introduced them as *synchronizing transformations*, which are combinations of consistency relations with a consistency preservation rule that preserves them. Correctness of such a transformation was then defined as the property of the consistency preservation rule to preserve consistency of given models according to the consistency relations (see Definition 4.8). In theory, a correct transformation can simply be achieved by adhering to that definition.

Using existing transformation languages, the defined transformations will, however, not follow the definition of a synchronizing transformation. Transformation languages usually allow the specification of unidirectional consistency preservation rules, i.e., rules that restore consistency by updating one model if the other was modified, such as QVT-O and QVT-R [QVT], ATL [Jou+06], or VIATRA [Ber+15]. Even if transformation languages allow bidirectional specifications, they still derive unidirectional consistency preservation rules from such a specification, such as forward and backward transformations (which may be incremental or not) derived from TGG rules [Leb+14]. In the following, we refer to such transformations as *ordinary transformations* and give a more precise definition of them later on. Synchronizing transformations, as we assume in transformation networks, are able to process changes made in both models and, in turn, also produce changes for both models. This is an inevitable property in transformation networks, because both models involved in a transformation may have been modified due to different sequences of transformations having modified both of them. The case that developers modify multiple models concurrently is sometimes also referred to as *synchronization*, although the term is sometimes even used for the simple case of incremental updates. If we consider that scenario, we will refer to it as *concurrent editing* to avoid confusion.

In this chapter, we aim to close this gap between synchronizing transformations as required in transformation networks and ordinary transformations with unidirectional consistency preservation rules used by transformation languages. We investigate which requirements such an ordinary transformation has to fulfill to emulate a synchronizing transformation and thus to be used in a transformation network. This chapter constitutes our contribution **C 1.3**, which consists of four subordinate contributions: a discussion of the formal basis for the gap between synchronizing and ordinary transformations; a discussion of different strategies to combine unidirectional consistency preservation rules of ordinary transformations to emulate a synchronizing transformation; a derivation of requirements for ordinary transformations to be used as synchronizing ones; and finally techniques to ensure that ordinary transformations fulfill these requirements. It answers the following research question:

RQ 1.3: Which requirements must a transformation fulfill for being used in a network in comparison to using it on its own?

The benefit of enabling the definition of ordinary transformations that can be used as synchronizing ones instead of providing an approach or language for the specification of synchronizing transformations is that existing and well-researched transformation languages and knowledge about them can be reused. Additionally, it is expected to reduce complexity, because the definition of two unidirectional consistency preservation rules is likely to be less cumbersome than the definition of a single synchronizing transformation, which has to consider all possible combinations of changes in two models. We will see that this is founded by the insight that only few combinations of changes are problematic and have to be considered explicitly.

We have published parts of the contributions in this chapter in previous work [Kla18; Kla+19b]. We have discussed the identification of essential issues when constructing synchronizing transformations from ordinary transformations that are defined in existing transformation languages [Kla18]. In the Master's thesis of Syma [Sym18], which was supervised by the author of this thesis, several issues in transformation networks have been identified, and for the category of changes arising from the combination of unidirectional transformation specifications a constructive solution has been proposed. We have also published that approach [Kla+19b] and present the results especially in Section 6.4.

6.1. Deriving the Gap to Ordinary Transformation

We have introduced that there is both a formal and a practical gap between synchronizing transformations, which we have defined as a component of transformation networks, and ordinary transformations, which are unidirectional and non-synchronizing as used by many transformation languages. In the following, we first give an example for faulty behavior if we simply used ordinary transformations in a transformation network. Afterwards, we give a formal definition of unidirectional preservation rules and ordinary transformations, then defined as *bidirectional transformations*. Finally, we discuss the relation between unidirectional consistency preservation rules and unidirectional consistency relations, as introduced in Section 4.4.

We have already sketched the example of creating a class in UML and Java after adding a component to a PCM model in Section 1.2.1. In that scenario, it was possible that for a created PCM component first a UML class is generated, which is then transformed into a Java class. Additionally, the transformation between PCM and Java creates another Java class, as it does not consider that there may be another transformation that has already created that class. Such scenarios can lead to the duplication of elements, as an already existing element is inserted again, or to an overwrite of an already existing element. Overwriting a previously created element may also remove information that was already added to it, like the transformation across UML may have added information to the Java class which is overwritten by the class creation of the transformation from PCM to Java.

An analogous example can be given for the running example of persons, employees, and residents depicted in Figure 3.3. We consider the consistency relations CR_{PE}, CR_{ER}, and CR_{PR}. As discussed in Chapter 5, these relations are compatible, thus for any given person, employee, or resident, there is a consistent tuple of models containing it. Thus, the relations do not prevent transformations from finding consistent models whenever a person, employee or resident is added. Ordinary transformations with unidirectional consistency preservation rules react to the changes in one model and update another accordingly. In case of adding a person, this may look as depicted in Figure 6.1. For each of the given consistency relations, we assume unidirectional consistency preservation rules that preserve consistency according to them. They especially create an employee for each added person and a resident for each created employee and person, respectively. Since the trans-

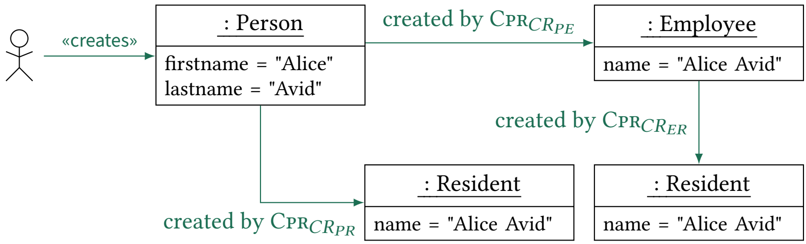


Figure 6.1.: Duplicate creation of a resident by two sequences of consistency preservation rules.

formations assume the models to be consistent before applying the changes, they always add a corresponding element when one of the elements is added. This leads to the situation that the consistency preservation rules for both CR_{PR} as well as CR_{ER}, namely $\textsc{Cpr}_{CR_{PR}}$ and $\textsc{Cpr}_{CR_{ER}}$, create a resident upon creation of a person. In consequence, there exist two residents with the same *name*, which does not fulfill the consistency relations.

It is our goal to find out how such a situation can be avoided by proper definition of consistency preservation rules in existing transformation languages. A simple solution in this example would have been to first check whether the elements to create already exist. This can either be done by using a trace model, which many transformation language use to store corresponding elements, or by searching for an appropriate element in the other model, using some key information like its name. Using a trace model, however, has some drawbacks and pitfalls, which we investigate in Subsection 6.4.2.

6.1.1. Unidirectional Consistency Preservation Rules

Before we can discuss options how unidirectional consistency preservation rules can be used to emulate the behavior of synchronizing consistency preservation rules, we first need to define them to be able to formally compare the two of them. In contrast to a synchronizing consistency preservation rule as defined in Definition 4.5, a unidirectional consistency preservation rule only receives changes made to one of the two models and returns changes to the other model instead of receiving and returning changes to both.

Definition 6.1 (Unidirectional Consistency Preservation Rule)
Let $\mathbb{CR}$ be a set of consistency relations between elements of two metamodels M_1 and M_2. A *unidirectional consistency preservation rule* $\textsc{Cpr}_{\mathbb{CR}}$ for the relation set $\mathbb{CR}$ is a partial function:

$$\textsc{Cpr}_{\mathbb{CR}} : (I_{M_1}, I_{M_2}, \Delta_{M_1}) \rightarrow \Delta_{M_2} \cup \{\perp\}$$

This is how the consistency preservation rules defined in or derived from many existing transformation languages operate. They take two models and changes to one of them and generate changes for the other. Most of them even directly apply the changes instead of returning a dedicated change artifact. The rule is partial to indicate inputs of models and changes that it is not able to handle. In these cases, the function returns $\perp$.

In addition, these rules usually expect the input models to be consistent and then ensure that after applying the input and the output changes to the models, the resulting models are consistent again. Their behavior for inconsistent input models is undefined, such that they either return $\perp$ or a change that does not necessarily guarantee that the models are consistent after applying the input and output changes. This conforms to the common notion of *correctness* for consistency preservation rules, like for the state-based (rather than our delta-based) notion of consistency preservation rules defined by Stevens [Ste10]. This is even compliant to the correctness notion that we have defined for synchronizing consistency preservation rules in Definition 4.6. Thus, we define correctness of a unidirectional consistency preservation rule as follows.

Definition 6.2 (Unidirectional Preservation Rule Correctness)
Let $\textsc{Cpr}_{\mathbb{CR}}$ be a unidirectional consistency preservation rule. We call $\textsc{Cpr}_{\mathbb{CR}}$ *correct* if, and only if, the resulting models when applying the input and output changes are consistent to $\mathbb{CR}$ again:

$$\begin{aligned}
&\textsc{Cpr}_{\mathbb{CR}} \textit{ correct } :\Leftrightarrow \forall\, m_1 \in I_{M_1}, m_2 \in I_{M_2}, \delta_{M_1} \in \Delta_{M_1} : \\
&\quad \big[\langle m_1, m_2 \rangle \textit{ consistent to } \mathbb{CR} \Rightarrow \\
&\qquad \forall\, \delta_{M_2} \in \Delta_{M_2} : \big(\textsc{Cpr}_{CR}(m_1, m_2, \delta_{M_1}) = \delta_{M_2} \\
&\qquad\quad \Rightarrow \langle \delta_{M_1}(m_1), \delta_{M_2}(m_2) \rangle \textit{ consistent to } \mathbb{CR}\big)\big]
\end{aligned}$$

In Definition 6.1, we explicitly allow consistency preservation rules to be partial. This was only an optional requirement for synchronizing consistency preservation rules defined in Definition 4.5, because there may be changes to both models that cannot be processed reasonably as one of the changes may need to be reverted to achieve consistency. Ignoring this practical requirement, it is theoretically possible to always return changes that, if applied to the input models, produce consistent models. These changes may perform arbitrarily unreasonable modifications but still restore consistency.

For unidirectional consistency preservation rules, partiality is not only a practical requirement. They must be partial, because there can be models for which no other models can be generated such that they are consistent to a consistency relation set. Consider the consistency relation $CR = \{\langle a, z\rangle, \langle b, z\rangle\}$ and its transposed $CR^T = \{\langle z, a\rangle, \langle z, b\rangle\}$. If a change led to the model $m = \{a, b\}$, then no second model to which it is consistent can be generated. A consistent model would have to contain z, because CR requires for a and b an element z to exist in another model. CR^T, however, requires that for a z only either a or b exists in the other model, as otherwise no witness structure with unique corresponding elements can be found (see Definition 4.19). In consequence, a unidirectional consistency preservation rule cannot produce a result for such an input without violating the correctness definition.

In fact, the definition does not specify for which inputs a unidirectional consistency preservation rule is allowed to be undefined. One could restrict this behavior to cases in which there is no $\delta_{M_2} \in \Delta_{M_2}$ for given models m_1 and m_2 and a given change $\delta_{M_1} \in \Delta_{M_1}$ such that $\langle \delta_{M_1}(m_1), \delta_{M_2}(m_2)\rangle$ *consistent to* $\mathbb{CR}$ for consistency relations $\mathbb{CR}$. We, however, leave it up to the developer to decide for which inputs a consistency preservation rule is undefined, as there might be cases in which a change restoring consistency can theoretically be generated but does semantically not make sense. This was also the reason for allowing a synchronizing consistency preservation rule to be partial, which is why we have already discussed the scenario in Subsection 4.3.2.

6.1.2. Unidirectional Relations and Preservation

Because of the definition of unidirectional consistency preservation rules based on a unidirectional notion of consistency relations, it seems reasonable to have a unidirectional consistency preservation rule associated with the unidirectional consistency relations for one direction between two metamodels.

For each pair of metamodels, this would result in two sets of unidirectional consistency relations and a consistency preservation rule for each of them.

It is, however, easy to see that a unidirectional consistency preservation rule cannot only consider one direction of consistency relations but needs to consider both. Consider the example in Figure 6.2, which contains an extract of the consistency relations of the running example. We assume the consistency relations CR_{ER} and CR_{ER}^{T} describing that for each employee a single corresponding resident must exist and vice versa. As discussed before, only considering CR_{ER} would realize the notion of not requiring an employee for every resident. If we define a unidirectional consistency preservation rule $\textsc{Cpr}_{CR_{ER}}$ only for the consistency relation CR_{ER} with the goal to always preserve consistency according to that relation after changes to the employee model, the example scenario 1 in Figure 6.2 shows that this is not the case. The rule properly propagates the change of adding an employee by adding a resident and thus restores consistency. Removing an employee, however, leads to a violation of consistency. The removal does not require the consistency preservation rule to perform any changes in the resident model, because CR_{ER} only requires a unique resident to exist for every employee, but does not forbid that there is a resident for which no employee exists. This is defined by the inverse relation CR_{ER}^{T}. In consequence, after removing an employee the consistency preservation rule does not perform any changes, as consistency to CR_{ER} is given, but the models are then inconsistent to CR_{ER}^{T}. $\textsc{Cpr}_{CR_{ER}}$ must, however, also be responsible for restoring consistency to CR_{ER}^{T} in case of an element removal, because the consistency preservation rule $\textsc{Cpr}_{CR_{ER}^{T}}$ for the inverse direction can not restore consistency to CR_{ER}^{T}, as the resident model was not changed.

The given scenario exemplifies the general case that consistency according to a consistency relation cannot only be violated by performing changes to the model containing the left condition elements of the relations, but also by changes to the model containing the right condition elements of the relation. In general, consistency of models to a consistency relation is affected by the presence of condition elements in the models. Consistency is defined as the ability to define a witness structure, i.e., a unique mapping between condition elements of the consistency relations that occur in the models. Thus, adding, changing, or removing elements in a model that constitute a condition element of the consistency relations can lead to inconsistencies.

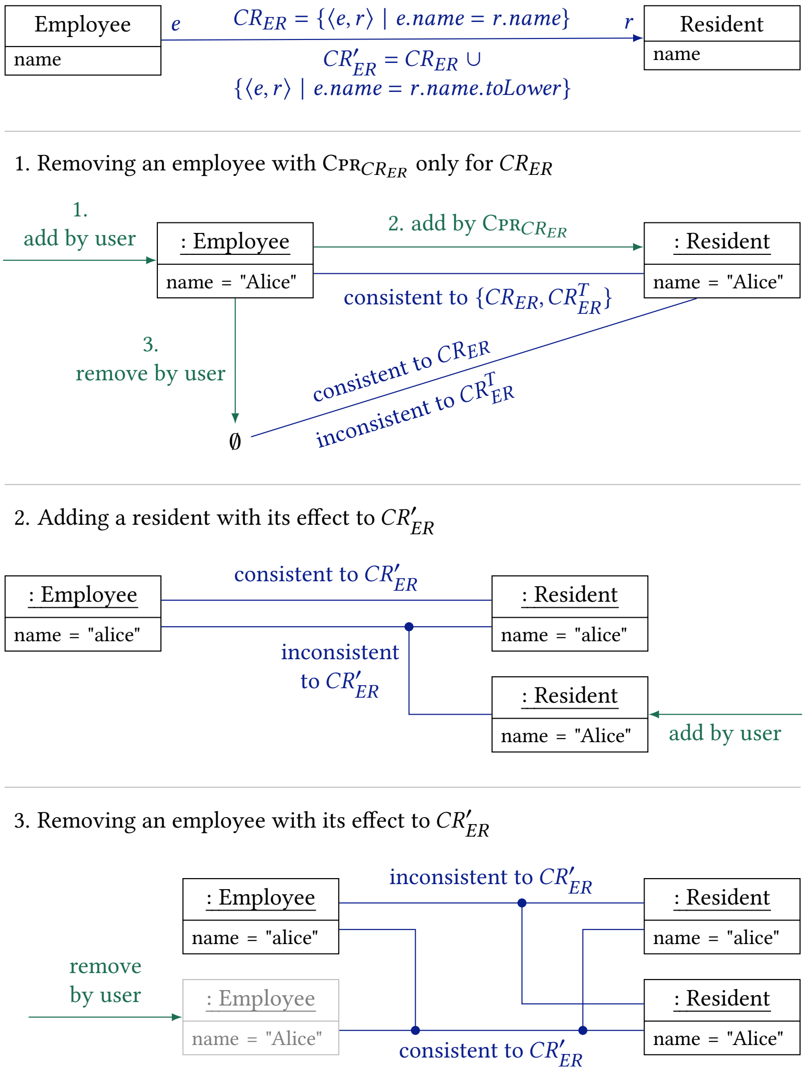


Figure 6.2.: Non-alignment of unidirectional relations and preservation rules. Blue lines without arrowheads connect elements that together are consistent or inconsistent to the noted relations. Green lines with arrowheads indicate changes by users or consistency preservation.

We can see that every type of change can lead to the violation of a consistency relation in either direction:

Addition: Whenever a condition element of the left side of a consistency relation is added to a model, a corresponding condition element needs to exist in another model. If it does not exist yet, the models are not consistent to that relation. When a condition element of the right side of a consistency relation is added to a model, this does, according to the definition of consistency, not require another condition element to exist in another model. It can, however, lead to the situation that no witness structure with a unique mapping between the elements exists anymore. Consider the exemplary relation CR'_{ER} in Figure 6.2 and the example scenario 2. Having an employee with the name “alice” and a corresponding resident with the same name, the models are consistent to that relation. Adding a resident with the name “Alice” violates CR'_{ER}, because the employee “alice” corresponds to both residents, so there is no mapping inducing a witness structure for consistency. In consequence, adding a condition element of the right side of the consistency relation to the models can also violate consistency to a consistency relation.

Removal: Whenever a condition element of the right side of a consistency relation is removed from a model, the corresponding condition element in the other model still exists. Because this element does not necessarily have a corresponding one anymore, there may not be a valid witness structure and thus the models may not be consistent anymore. When a condition element of the left side of a consistency relation is removed from a model, the originally corresponding element is not connected to the removed element in the witness structure anymore. If there is another element that occurs in a consistency relation pair with that corresponding element, there is no unique mapping of elements anymore. Consider again the relation CR'_{ER} in Figure 6.2 and the example scenario 3. Having two employees and residents with the names “alice” and “Alice”, the models are consistent, because each employee has a corresponding resident and vice versa. If we remove the employee “Alice”, the models are not consistent to CR'_{ER} anymore, because the remaining employee corresponds to both residents, so there is no unique mapping between condition elements representing a witness structure.

Change: We do not have a precise notion of when a condition element can be considered changed, as elements do not have an identity. Additionally, consistency in terms of being able to find a witness structure is only based on the existence or non-existence of condition elements, thus whether an element was changed or whether it was removed and created makes no difference. We might say that a condition element can be considered changed when the change describes modifications of the model elements in the condition element that lead to a new condition element within the same condition. This does, conceptually, not differ from the removal of one and the addition of another condition element. Thus, the same situations as discussed for addition and removal above can occur.

It is also easy to see that there is no trivial way of specifying a unidirectional consistency preservation rule that is synchronizing. It may seem natural to define a consistency preservation rule that is able to process changes in both models and then return only changes in one of them to restore consistency to close the gap between synchronizing and ordinary transformations. Consider the situation that we have two residents and employees named “Alice” and “Bob”. If one of them is removed in the residents model and the other in the employees model, then a proper synchronizing transformation should remove both corresponding elements such that the models are empty. This requires changes to both models. With a unidirectional consistency preservation rule for each direction, neither of them can produce changes in one of the models that reasonably restore consistency. Such a rule would necessarily revert one removal to restore consistency, which is not the intended behavior and would probably not be specified by a developer that way. In consequence, the consistency preservation rule would be undefined for that input, although a synchronization transformation would be able to resolve those changes. In fact, we would expect to have two unidirectional consistency preservation rules of which each removes one of the elements. This does, however, violate our existing notion of correctness for a single consistency preservation rule. In the subsequent sections, we therefore discuss relaxed requirements to unidirectional consistency preservation rules to be able to act like a synchronizing transformation.

6.1.3. Bidirectional Transformations

A unidirectional consistency preservation rule usually appears in combination with another rule for the opposite direction. We have seen that even a single unidirectional consistency relation between two metamodels requires unidirectional consistency preservation rules for both directions to preserve consistency according to that relation after changes to instances of either of the metamodels. Many transformation languages allow the specification of *bidirectional transformations*, which means that they derive unidirectional consistency preservation rules for both directions (see Section 2.4).

In general, it is reasonable to consider two unidirectional consistency preservation rules between two metamodels together, such that after changes in instances of any of the two metamodels, the other can be updated to restore consistency. A synchronizing transformation according to Definition 4.7 is also able to process changes in any of the two models, thus such a notion fits to our goal of emulating synchronizing transformations. According to common terminology, we define this as a *bidirectional transformation.*

Definition 6.3 (Bidirectional Transformation)

Let M_1 and M_2 be two metamodels, and let $\mathbb{CR}$ be a set of consistency relations between them. Additionally, let $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ and $\textsc{Cpr}_{\mathbb{CR}}^{\leftarrow}$ be unidirectional consistency preservation rules with:

$$\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow} : (I_{M_1}, I_{M_2}, \Delta_{M_1}) \rightarrow \Delta_{M_2} \cup \{\perp\}$$
$$\textsc{Cpr}_{\mathbb{CR}}^{\leftarrow} : (I_{M_2}, I_{M_1}, \Delta_{M_2}) \rightarrow \Delta_{M_1} \cup \{\perp\}$$

A *bidirectional transformation* is a triple $\mathfrak{t} = \langle \mathbb{CR}, \textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}, \textsc{Cpr}_{\mathbb{CR}}^{\leftarrow} \rangle$.

We call such a bidirectional transformation correct if both consistency preservation rules are correct according to Definition 6.2.

Definition 6.4 (Bidirectional Transformation Correctness)

Let $\mathfrak{t} = \langle \mathbb{CR}, \textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}, \textsc{Cpr}_{\mathbb{CR}}^{\leftarrow} \rangle$ be a bidirectional transformation. We call $\mathfrak{t}$ correct if, and only if, $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ and $\textsc{Cpr}_{\mathbb{CR}}^{\leftarrow}$ are both correct.

Such bidirectional transformations ensure that if any of two models is changed, a change for the other is generated such that both changed models are consistent again, or it may fail returning $\perp$. This does, however, not reflect the case that both models have been modified concurrently, as it is the case in transformation networks and thus supported by our initial definition of synchronizing transformations. We therefore discuss in the following sections how we can combine the unidirectional consistency preservation rules of a bidirectional transformation and which requirements we have to make to them such that the bidirectional transformation behaves like a synchronizing one.

6.2. Combining Unidirectional Consistency Preservation Rules

We have introduced that bidirectional transformations, as we assume to be the notion for practically usable transformation specifications, can only be applied after changes to one model and update the other to restore consistency. This induces a gap to synchronizing transformations, as required in transformation networks, which are able to accept changes made in both models and update both models to restore consistency. To close this gap, we discuss options to combine the unidirectional consistency preservation rules of a bidirectional transformation, such that it considers changes made to both models and thus acts like a synchronizing transformation.

6.2.1. Options for Combination

Existing work already proposed strategies to synchronize concurrent changes between two models. This includes techniques for processing concurrent changes with TGGs [Her+12; OPN20] and specific algorithms for a general notion of synchronizing transformations according to our definition [Xio+13; Xio+09]. All these approaches, however, deal with the more general case that arbitrary changes may have been made. This especially includes conflicting updates by one or more users, which need to be resolved and potentially require one of the changes to be reverted.

We are, however, in the situation that transformations do not perform arbitrary changes and that changes of other transformations may need to be revised but not reverted. For example, it may be necessary to update an attribute value again, because the interval of consistent values of the currently executed transformation is smaller than the one of a transformation executed before. It will, however, not be necessary to completely revert the modification of the attribute value, because the modification was necessary for another transformation to restore consistency. Thus, the causal change for which consistency was restored would need to be reverted as well. Finally, this would result in reverting a user change, which should never happen.

We assume the consistency relations of transformations to be compatible according to Definition 5.3, which excludes contradictions that may prevent transformations from finding a consistent result for specific changes. This assumptions reduces the potential conflicts that may occur when changes of different transformations need to be synchronized.

A bidirectional transformation according to Definition 6.3 consists of two unidirectional consistency preservation rules. We have discussed in Subsection 6.1.2 that it is not possible to extend those consistency preservation rules to be synchronizing such that the execution of a single unidirectional consistency preservation rule restores consistency to all consistency relations after changes to both models. In fact, it will be necessary to execute both preservation rules at least once to restore consistency. Different options to apply the rules exist, each having individual benefits and drawbacks.

We have sketched two scenarios for executing multiple consistency preservation rules in Subsection 4.1.3, which can be transferred to the case of executing the two consistency preservation rules of a bidirectional transformation. A first option is to independently apply the consistency preservation rules and then merge the results. Imagine models m_1 and m_2 and changes δ_{M_1} and δ_{M_2} to them. Applying the two unidirectional consistency preservation rules independently yields $\delta'_{M_2} = \textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}(m_1, m_2, \delta_{M_1})$ and $\delta'_{M_1} = \textsc{Cpr}_{\overleftarrow{\mathbb{CR}}}(m_2, m_1, \delta_{M_2})$. It is, however, not guaranteed that $\langle \delta'_{M_1}(\delta_{M_1}(m_1)), \delta'_{M_2}(\delta_{M_2}(m_2)) \rangle$ is consistent to $\mathbb{CR}$. It is even not guaranteed that the changes, such as δ_{M_1} and δ'_{M_1}, can be concatenated at all, since δ'_{M_1} was generated for m_1 and not for $\delta_{M_1}(m_1)$. As an example, δ_{M_1} may remove an element from m_1, which δ'_{M_1} changes. Even if the change is still defined for that modified model, the result may not be consistent, because the necessary change produced by $\textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}$ cannot be applied anymore. Thus merging the changes of both consistency preservation rules does not necessarily yield a consistent result.

Another option is to sequence the execution. In a first step, we generate the change $\delta'_{M_2} = \textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}(m_1, m_2, \delta_{M_1})$ as before. Then, $\langle \delta_{M_1}(m_1), \delta'_{M_2}(m_2) \rangle$ is consistent due to correctness of $\textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}$. Afterwards, we apply the second consistency preservation rule to the newly generated consistent models and the original change δ_{M_2} to m_2, thus $\delta'_{M_1} = \textsc{Cpr}_{\overleftarrow{\mathbb{CR}}}(\delta'_{M_2}(m_2), \delta_{M_1}(m_1), \delta_{M_2})$.

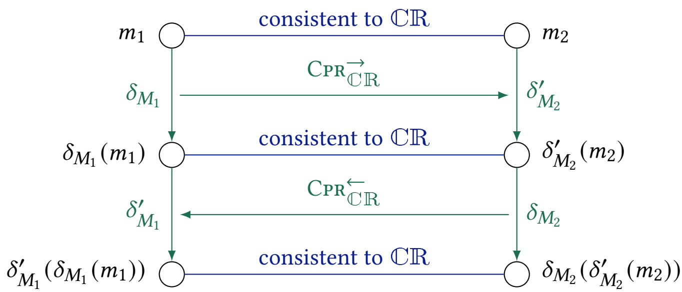


Figure 6.3.: Schema for sequencing unidirectional consistency preservation rules after concurrent changes. Circles denote model states, blue lines connect consistent models, and green lines with arrowheads denote the execution of changes or consistency preservation.

As a result, we receive $\langle \delta'_{M_1}(\delta_{M_1}(m_1)), \delta_{M_2}(\delta'_{M_2}(m_2)) \rangle$, which is consistent to $\mathbb{CR}$. This means that δ_{M_2} is not applied to m_2 anymore, in which the changes were performed originally, but needs to be applied to $\delta'_{M_2}(m_2)$. It is, again, unclear whether the change can be applied to that state, i.e., whether δ_{M_2} is defined for $\delta'_{M_2}(m_2)$. However, if the changes are applicable, all original changes are reflected in the result. In addition, the resulting models are consistent because of correctness of the consistency preservation rules.

Both discussed options have the drawback that they cannot guarantee to produce a result, as it is possible that the involved changes cannot be concatenated. In addition, the first option of independently applying the consistency preservation rules and then merging the results cannot even guarantee that the resulting models are consistent if changes can be concatenated. Thus, we only consider the second option of sequencing the execution of consistency preservation rules and further discuss it in the following.

6.2.2. Sequencing of Consistency Preservation Rules

The sequential application of original changes and execution of consistency preservation rules is depicted schematically in Figure 6.3. It has two important properties. First, it ensures that all original changes are applied to the models and, second, it guarantees that the resulting models are consistent. It is, however, only applicable in specific situations. The optimal case, in which the approach is always applicable, is if $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ produces changes for the

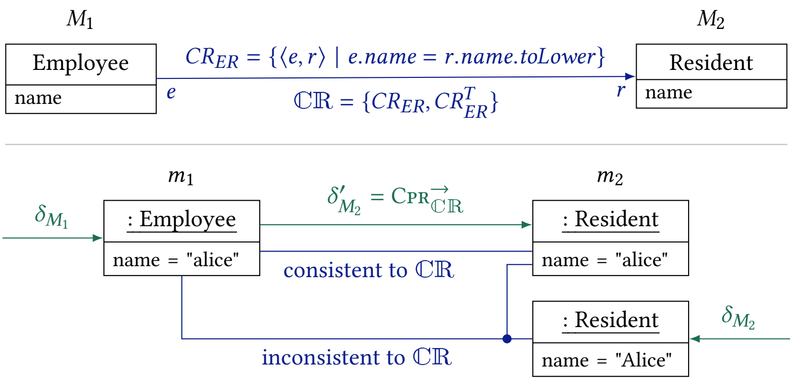


Figure 6.4.: Example for non-transformability when sequencing the application of unidirectional consistency preservation rules and concurrent changes. Blue lines without arrowheads connect elements that are (in-)consistent to $\mathbb{CR}$, and green lines with arrowheads indicate changes.

second model that affect a disjoint set of elements in $\mathbb{CR}$ compared to the original changes to the second model δ_{M_2}. If two changes affect completely disjoint sets of elements, they can obviously be consecutively applied. It would then not even make a difference in which order they are applied.

Unfortunately, the change δ'_{M_2} produced by $\textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}$ and the original one δ_{M_2} produced by other transformations do not necessarily affect disjoint sets of elements. In that case, the two following problems can occur.

Non-Applicability: The most obvious problem, which we have already discussed, is that the original change to the second model δ_{M_2} cannot be applied to the model changed by δ'_{M_2} as the result of $\textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}$. This can, for example, happen when δ'_{M_2} removes an element that is affected by δ_{M_2}. Since the element was changed in δ_{M_2}, it is part of a condition element in another transformation that was executed before. As $\textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}$ removed that element, the condition element does no longer exist anyway, thus this removal has to be propagated back by the transformation that originally introduced the change δ_{M_2}. In consequence, the modification in δ_{M_2} can simply be ignored. In the worst case, all elements affected by δ_{M_2} were removed by δ'_{M_2}. Then, δ_{M_2} can be completely ignored, because all condition elements of the involved consistency relations were removed. Thus, we can always ensure that the changes, at least those that are still relevant, can still be applied.

Non-Transformability: Even if the change δ_{M_2} can be applied to $\delta'_{M_2}(m_2)$, this does not guarantee that $\textsc{Cpr}^{\leftarrow}_{\mathbb{CR}}$ is able to process the given change. In fact, this requirement applies to all changes, even including original user changes, but there are special circumstances in this situation that make the transformation prone to not being able to transform the changes. Whenever δ'_{M_2} adds condition elements that were already added by δ_{M_2}, their concatenation can lead to a duplication of those elements. Consider the scenario depicted in Figure 6.4 with consistency relations $\mathbb{CR} = \{CR_{ER}, CR^T_{ER}\}$. An employee "alice" is added by the original change to m_1. The consistency preservation rule then generates an appropriate resident with the same name to fulfill the consistency relation. The original change to m_2 adds a resident "Alice", which was generated by another transformation, e.g., the one that created an appropriate person and changed the capitalization of the name. Applying this original change leads to two residents with different name capitalizations. Now it is impossible for $\textsc{Cpr}^{\leftarrow}_{\mathbb{CR}}$ to generate a change δ'_{M_1} for the first model to restore consistency. The employee corresponds to both residents, as both fulfill the constraint of the consistency relation. But there is no additional employee that could be added to achieve a unique mapping between corresponding elements. A synchronizing transformation would have been able to produce a consistent result by considering both original changes at once and then simply not performing any additional changes, as the originally added resident is already consistent to the originally added employee. In consequence, if the unidirectional consistency preservation rule had known that the resident was already added, it would not have performed any changes.

As remarked before, the situation that certain changes cannot be processed by the consistency preservation rules cannot be avoided. If the user had added the second resident in the previous scenario, there would have also been no possibility for the consistency preservation rule to generate changes that restore consistency. The difference is, however, that in this case it is fine that no result is found. In case of the scenario discussed above, the original changes could have been reasonably processed to a consistent result if the unidirectional consistency preservation rule would have considered that there was already a change that restored consistency.

In consequence, it is inevitable that consistency preservation rules need to be able to deal with the situation that the target model was already modified, such that the given models are not initially consistent. This is necessary

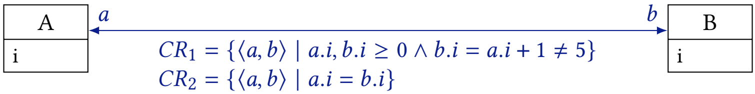


Figure 6.5.: Two consistency relations requiring multiple executions of unidirectional consistency preservation rules to find a consistent result.

to reflect the changes that have already been made and to integrate them into consistency preservation. In consequence, we finally have to relax our requirements for the input of consistency preservation rules to be able to consider the changes to both models. This means that we need to make further requirements to the preservation rules, because we do not yet assume the consistency preservation rules to produce results for inputs that are not consistent. We have already given examples for scenarios in which it is not possible to restore consistency by one unidirectional consistency preservation rule after changes in both models.

Before we define a precise notion of further requirements to consistency preservation rules that accept inconsistent inputs, we first discuss how often it may be necessary to execute both consistency preservation rules to restore consistency, as this directly affects the requirements we have to define.

6.2.3. Execution Bounds

Correctness of unidirectional consistency preservation rules ensures that after executing such a rule the resulting models are consistent. It is easy to see that this correctness notion cannot be fulfilled for certain sets of consistency relation sets. This is exemplified at the artificial scenario depicted in Figure 6.5. We consider two consistency relations CR_1 and CR_2 and their transposed relations, i.e., $\mathbb{CR} = \{CR_1, CR_1^T, CR_2, CR_2^T\}$. CR_1 requires that for each A an instance of B exists that has the same value of i incremented by 1. The only exception is that if i in A is 4 (or any other arbitrary value), then no corresponding element B is required. CR_2 requires that for each A an instance of B exists, which has the same value of i. We want to define a bidirectional transformation of two unidirectional consistency preservation rules $\textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}$ for propagating changes in models with instances of A to one with instances of B and $\textsc{Cpr}_{\overleftarrow{\mathbb{CR}}}$ to propagate changes in the opposite direction.

Consider the following scenario: If an A with $i = 0$ is added to an empty model, $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ cannot perform any changes in an (also empty) model with instances of B that restore consistency. Because of CR_1, a B with $i = 1$ has to be created, and because of CR_2, a B with $i = 0$ has to be created. While this also fulfills CR_1^T, the existence of B with $i = 1$ requires the existence of an A with $i = 1$ due to CR_2^T. Since $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ cannot modify the model with instances of A, it is impossible for $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ to restore consistency in that case.

Allowing the consistency preservation rules to react to each other multiple times can, however, lead to a consistent result. If $\textsc{Cpr}_{\mathbb{CR}}^{\leftarrow}$ adds an A with $i = 1$ in response to the previous execution of $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$, all consistency relations except CR_1 are fulfilled. $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ can then create a B with $i = 2$, which is iteratively processed by $\textsc{Cpr}_{\mathbb{CR}}^{\leftarrow}$. This process terminates as soon as $\textsc{Cpr}_{\mathbb{CR}}^{\leftarrow}$ adds an A with $i = 4$, as then CR_1 is also fulfilled, because it does not require a corresponding B for an A with $i = 4$.

We have seen that it is possible to execute unidirectional consistency preservation rules multiple times to achieve a consistent state and that it is not always possible to ensure consistency with only one execution of such a rule. In fact, the number of necessary executions of consistency preservation rules can be arbitrarily high. The value of 5 in CR_1 of the example can be exchanged by any value requiring an arbitrary high number of executions. We may only circumvent this by requiring that $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ must perform changes such that $\textsc{Cpr}_{\mathbb{CR}}^{\leftarrow}$ can then restore consistency with a single execution. In our scenario, this would mean that $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ adds all instances of B with $i \leq 4$. Anyway, such a behavior requires a relaxation of the correctness requirement for consistency preservation rules, because the execution of $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ can never result in a consistent state.

Additionally, it may be desired that elements of a consistency relation are created by a consistency preservation rule, although a condition element was only created partially so far. In that case, the partial condition element has to be completed in one model in addition to the creation of the corresponding condition element in the other model. Thus, changes in both models are required, which can only be achieved by executing both consistency preservation rules and accepting that executing the first one does not result in consistent models. An example for such a scenario could be the consistency relation between a component in the PCM and its realization as a package and a class in Java. It may be desired that a package at a specific place, e.g., within a "components" package, or with a specific name, e.g., containing

"Component", in the Java code is identified as a component. Creating such a package shall then lead to the creation of a component in the the PCM model as well as of the implementation class in Java. In that case, there is no complete condition element created in Java, because this would also require the existence of an appropriate class. If the elements shall still be created, both models have to be changed. Thus, the first consistency preservation rule introduces the PCM component, which introduces an inconsistency between the models, as the corresponding Java class is missing. This is then corrected by the consistency preservation rule in opposite direction adding the implementation class.

Finally, it is questionable whether such scenarios should be considered in the formal framework or if it should be up to a developer to implement such a scenario without having specific guarantees regarding termination of the consistency preservation rules or regarding consistency of the models after executing the rules a specific number of times. Since we need to relax the requirement of consistency preservation rules to always produce consistent results after one execution in the synchronization scenario where both models have been modified, we will allow the consistency preservation rules to be executed more than once anyway. Regarding the example in Figure 6.5, if we started with an A with $i = 6$ and let the consistency preservation rules operate as discussed above, i.e., always adding the elements with i incremented by one, this process would never terminate. We thus need to ensure that such an execution terminates. Since the consistency preservation rules depend on each other, this will, however, be a property of the bidirectional transformation rather than the individual consistency preservation rule.

6.2.4. Necessity for Synchronization Extension

In the previous subsections, we have discussed that after changes to two models, these changes and the ones produced by consistency preservation rules that restore consistency between these models cannot be sequenced in a way such that we receive consistent models in all cases the consistency preservation rules are able to handle. We especially found that it is necessary for a unidirectional consistency preservation rule to consider the changes made to the model it is supposed to modify. Thus, we need to enable consistency preservation rules to deal with the situation that the input models are inconsistent. In our current definition, no behavior of a consistency preservation rule and the encapsulating bidirectional transformation for such a situation is defined. Thus, we discuss an appropriate extension of bidirectional transformations that support this scenario of synchronization in the following section.

Additionally, we found that consistency preservation rules may need to be executed multiple times. This is obviously necessary to make bidirectional transformations synchronizing, as they need to be able to change both models after both of them may have been modified. Therefore, we consider how we can achieve execution bounds, such that the termination of multiple executions of the consistency preservation rules of a bidirectional transformation is guaranteed.

6.3. Synchronizing Bidirectional Transformations

In the following, we discuss how we can extend bidirectional transformations and, in particular, their unidirectional consistency preservation rules such that they are able to deal with the situation that both models may have been modified. To achieve this, we extend consistency preservation rules to also accept models that are not initially consistent. We can then not require them to restore consistency with a single execution anymore. Instead, we define a notion of *partial consistency*, which allows us to specify how the execution of consistency preservation rules has to improve the degree of consistency. We derive requirements for the transformations to improve partial consistency and finally show that transformations fulfilling these requirements terminate consistently.

6.3.1. Partial Consistency of Models

Given two models m_1 and m_2 and changes δ_{M_1} and δ_{M_2} to each of them, a unidirectional consistency preservation rule $\textsc{Cpr}_{\vec{\mathbb{CR}}}$ needs to accept and process the change in one model, be it δ_{M_1} without loss of generality, and receive the unchanged model m_1 as well as the changed second model $\delta_{M_2}(m_2)$. We have discussed the necessity to process the changed second model in the previous section. While m_1 and m_2 are consistent, m_1 and $\delta_{M_2}(m_2)$ may not. In consequence, $\textsc{Cpr}_{\vec{\mathbb{CR}}}$, even if correct according to Definition 6.2, does not guarantee that applying the returned change yields consistent models, as its behavior for inconsistent input models is undefined. m_1 and $\delta_{M_2}(m_2)$ will, however, usually still fulfill some kind of partial consistency notion. Depending on the complexity of δ_{M_2} large parts of the models will still be consistent. Such a notion of partial consistency may be defined in two ways. First, two models may only fulfill an extract of the consistency relations. Second, only extracts of two models may fulfill the consistency relations.

In the first option, we consider that the given models are only consistent to a subset of the given consistency relations. There may, however, be only a single element in the models that leads to the violation of all consistency relations. Thus, we would call the models completely inconsistent just because of a single element. We could circumvent that by defining a notion of partial consistency relations, such that we can consider models consistent to a part of a consistency relation. Such a notion would have to be defined at the level of consistency relation pairs and their condition elements within the consistency relations. Considering subsets of consistency relations, i.e., only a subset of their consistency relation pairs, would, however, not make sense, because when analyzing consistency of two models those consistency relation pairs are not independent. Consistency is not evaluated individually for each consistency relation pair but by the ability to find a witness structure, which is a subset of the consistency relation pairs that uniquely relates the condition elements of a consistency relation that occur within models. Thus, if consistency to a relation is given by removing only a single consistency relation pair does not mean that there is only one missing or superfluous element in the models to be consistent. Due to these interdependencies of consistency relation pairs, consistency to partial consistency relations will in general not provide insights on the reasons for models being inconsistent, which is why we do not consider this as our notion for partial consistency.

In the second option, we consider that only parts of the given models are consistent to all given consistency relations. In addition to the missing ability of the first option to give reasonable insights on inconsistencies, this, intuitively, is a more reasonable notion, because it explicitly defines that parts of the models are consistent whereas other parts are not. We thus define partial consistency as models having subsets that are actually consistent. To identify how far models are partially consistent, we also define an according metric based on finding maximal subsets of the models that are consistent.

Definition 6.5 (Partial Consistency)

Let $\mathbb{CR}$ be a set of consistency relations. Given two models $m_1 \in I_{M_1}$ and $m_2 \in I_{M_2}$, their *maximal consistent subsets* $m_1^p \subseteq m_1$ and $m_2^p \subseteq m_2$ with regards to $\mathbb{CR}$ are the subsets of m_1 and m_2 that are consistent and larger than all other consistent subsets:

$$\begin{aligned}
&\langle m_1^p, m_2^p \rangle \textit{ consistent to } \mathbb{CR} \wedge m_1^p \subseteq m_1 \wedge m_2^p \subseteq m_2 \\
&\quad \wedge \forall m_1^{p'} \in \mathcal{P}(m_1), m_2^{p'} \in \mathcal{P}(m_2) : \\
&\qquad \big(\langle m_1^{p'}, m_2^{p'} \rangle \textit{ consistent to } \mathbb{CR} \Rightarrow |m_1^{p'}| + |m_2^{p'}| \leq |m_1^p| + |m_2^p|\big)
\end{aligned}$$

Partial consistency $\textsc{cons}_{\mathbb{CR}}$ of two models regarding $\mathbb{CR}$ is the ratio between the sizes of the maximal consistent subsets and the models:

$$\textsc{cons}_{\mathbb{CR}} : (I_{M_1}, I_{M_2}) \rightarrow [0,1], \quad (m_1, m_2) \mapsto \tfrac{|m_1^p|+|m_2^p|}{|m_1|+|m_2|}$$

Such maximal consistent subsets always exist. When models are not consistent in any way, it is $m_1^p = m_2^p = \emptyset$, because empty models are consistent by definition. In that case, partial consistency of the models is 0. When models are consistent, the maximal consistent subsets are the models themselves, which is why partial consistency is 1.

A comparable notion of partial consistency has been introduced by Stevens [Ste14; Ste20b]. She introduces a *consistency indicator* replacing a consistency relation, which determines how consistent two models are. It is based on a partial order between models regarding their degree of consistency. This notion is used to define partial bidirectional transformations that ensure that their execution does not reduce consistency. Our definition of partial consistency can be seen as an implementation of such a consistency indicator. We, however, use the notion to ensure that the iterative application of consistency preservation rules of transformations results in totally consistent models after a finite number of steps.

6.3.2. Transformations for Partially Consistent Models

Before we consider the case that two models have been modified and need to be synchronized, we start with the case that of two initially consistent models one has been changed. We then extend that scenario to the case when both models have been changed. We use the notion of partial consistency to define that the given models are initially partially consistent and how this partial consistency improves by executing the bidirectional transformation. As discussed in Subsection 6.2.3, it may be necessary to execute the consistency preservation rules multiple times to achieve a consistent state, producing several intermediate changes that generate partially consistent models.

In the following, we derive the properties a bidirectional transformation has to fulfill to eventually return models that are consistent if applied repeatedly. They are based on the idea that each execution has to improve partial consistency of the given models. Since a single consistency preservation rule may not be able to improve partial consistency in every case, we always consider the combination of both preservation rules of a bidirectional transformation and require that property from them. Therefore, we define the notion of a *bidirectional transformation execution step*, which is composed of a single execution of both unidirectional consistency preservation rules.

Definition 6.6 (Bidirectional Transformation Execution Step)
Let $\mathfrak{t} = \langle \mathbb{CR}, \textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}, \textsc{Cpr}_{\overleftarrow{\mathbb{CR}}} \rangle$ be a bidirectional transformation for metamodels M_1 and M_2. An *execution step* $\mathrm{Ex}^1_{\mathfrak{t}}$ of $\mathfrak{t}$ is a function:

$$\begin{aligned} \mathrm{Ex}^1_{\mathfrak{t}} : (I_{M_1}, I_{M_2}, \Delta_{M_1}) &\rightarrow (I_{M_1}, I_{M_2}, \Delta_{M_1}) \cup \{\perp\} \\ (m_1, m_2, \delta_{M_1}) &\mapsto \begin{cases} (m_1', m_2', \delta_{M_1}') \\ \perp \end{cases} \qquad \textit{with:} \end{aligned}$$

$$\begin{aligned} \delta_{M_2}' &:= \textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}(m_1, m_2, \delta_{M_1}) \qquad & m_1' &:= \delta_{M_1}(m_1) \\ \delta_{M_1}' &:= \textsc{Cpr}_{\overleftarrow{\mathbb{CR}}}(m_2, m_1', \delta_{M_2}') \qquad & m_2' &:= \delta_{M_2}'(m_2) \end{aligned}$$

If $\textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}(m_1, m_2, \delta_{M_1}) = \perp$ or $\textsc{Cpr}_{\overleftarrow{\mathbb{CR}}}(m_2, m_1', \delta_{M_2}') = \perp$, then the execution is undefined, i.e., $\mathrm{Ex}^1_{\mathfrak{t}}(m_1, m_2, \delta_{M_1}) = \perp$.

Such execution steps can be applied repeatedly. Each execution step delivers a new change to the first model and a changed version of the second model by applying the changes delivered by the consistency preservation rules of the bidirectional transformation. The execution step can be reapplied to these resulting models and the resulting change.

Algorithm 6.1 Execution of a bidirectional transformation.

procedure $\textsc{Execute}(\mathfrak{t} = \langle \mathbb{CR}, \textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}, \textsc{Cpr}_{\mathbb{CR}}^{\leftarrow} \rangle, m_1, m_2, \delta_{M_1})$
  if $\neg(\langle m_1, m_2 \rangle$ *consistent to* $\mathbb{CR})$ **then**
    return $\perp$
  end if
  while $\neg(\langle \delta_{M_1}(m_1), m_2 \rangle$ *consistent to* $\mathbb{CR})$ **do**
    $(m_1, m_2, \delta_{M_1}) \leftarrow \mathrm{Ex}_{\mathfrak{t}}^{1}(m_1, m_2, \delta_{M_1})$
    if $(m_1, m_2, \delta_{M_1}) = \perp$ **then**
      return $\perp$
    end if
  end while
  return $\langle \delta_{M_1}(m_1), m_2 \rangle$
end procedure

The execution of a bidirectional transformation consists of the consecutive application of execution steps until the delivered models are consistent, as defined in Algorithm 6.1. Although we, theoretically, require the consistency preservation rules to handle initial models that can be arbitrarily inconsistent, it will not be possible to define such rules in practice. Therefore, we stick to the requirement that inconsistencies are introduced by changes. Then, it is up to the consistency preservation rules to process the changes in a way such that all introduced inconsistencies are resolved.

Without loss of generality, we have defined bidirectional transformation execution steps for original changes in M_1, although the consistency preservation rules of a transformation are also able to handle changes in M_2. The definitions can be applied to that case accordingly by swapping $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$ and $\textsc{Cpr}_{\mathbb{CR}}^{\leftarrow}$. Since we finally consider the case that both models have been changed, it is not relevant for us which change to consider first.

6.3.3. Transformation Execution Termination

The algorithm obviously only returns $\perp$ if an execution step of the transformation cannot be applied. Additionally, we can easily show that in all other cases in which the algorithm terminates, it returns consistent models.

Lemma 6.1 (Bidirectional Transformation Execution Consistency)

If Algorithm 6.1 terminates, it either returns $\bot$ or a consistent model pair.

Proof. Algorithm 6.1 terminates with one of its return statements. It returns $\bot$ in Line 3 or Line 8, which fulfills the lemma. It returns $\langle \delta_{M_1}(m_1), m_2 \rangle$ in Line 11. This line is reached when the loop condition was not fulfilled, i.e., when $\langle \delta_{M_1}(m_1), m_2 \rangle$ *consistent to* $\mathbb{CR}$, which fulfills the lemma. $\square$

The algorithm does, however, not ensure termination for arbitrary bidirectional transformations and input models and changes. To ensure termination, we need to assure that after a finite number of execution steps of the transformation the algorithm either delivers consistent models or cannot apply further execution steps, i.e., returns $\bot$. To achieve this, we enforce execution steps to improve partial consistency to finally reach a consistent state. We provide the following notion of *partial consistency improvement* for that.

Definition 6.7 (Partial Consistency Improvement)

Let $\mathfrak{t}$ be a bidirectional transformation for metamodels M_1 and M_2. We say that $\mathfrak{t}$ is *partial-consistency-improving* if, and only if, an execution step always improves partial consistency by reducing the size of the models or improving the size of the maximal consistent subsets.

We define $(m_1', m_2', \delta_{M_1}') := \mathrm{Ex}_{\mathfrak{t}}^1(m_1, m_2, \delta_{M_1})$ for all inputs, for which $\mathrm{Ex}_{\mathfrak{t}}^1$ does not return $\bot$. We denote $\delta_{M_1}(m_1)^p$ and m_2^p as the maximal consistent subsets of $\delta_{M_1}(m_1)$ and m_2. We denote $\delta_{M_1}'(\delta_{M_1}(m_1))^p$ and $\delta_{M_2}'(m_2)^p$ as the maximal consistent subsets of $\delta_{M_1}'(\delta_{M_1}(m_1))$ and $\delta_{M_2}'(m_2)$. We require that when $\delta_{M_1}(m_1)^p \neq \delta_{M_1}(m_1)$ and $m_2^p \neq m_2$ (i.e., when $\delta_{M_1}(m_1)$ and m_2 are not consistent):

$$
\begin{aligned}
&|\delta_{M_1}'(\delta_{M_1}(m_1))^p| + |\delta_{M_2}'(m_2)^p| - |\delta_{M_1}(m_1)^p| - |m_2^p| \\
&\quad > |\delta_{M_1}'(\delta_{M_1}(m_1))| + |\delta_{M_2}'(m_2)| - |\delta_{M_1}(m_1)| - |m_2|
\end{aligned}
$$

Although the definition may first look like a rather theoretic requirement, it obviously matches an intuitive expectation regarding consistency preservation. In each execution step of the bidirectional transformation, we expect that no existing consistency is destroyed and that further consistency is introduced. To this end, we expect either the size of the maximal consistent subsets to improve more than the size of the models or the size of the models to decrease more than the size of the maximal consistent subsets. This is reasonable, because consistency preservation should either add or modify elements such that more elements are consistent or remove elements that are inconsistent because their corresponding elements were removed.

In the first case, the size of the maximal consistent subsets is improved by adding or modifying elements such that they are consistent again. At the same time, models should not increase in size by the same value as the maximal consistent subsets do, because then elements were added which do either not improve consistency of any already existing element or otherwise violate consistency of some of the existing elements. We do, however, not want consistency preservation rules to violate consistency for any already consistent element. In the second case, the size of the models is decreased by removing elements that were not consistent because of the removal of a corresponding element. At the same time, models should not decrease in size by the same value due to the same reasons as in the first case. If elements are removed from the models, which were also present in the maximal consistent subsets, elements that were actually consistent are removed, which is undesired. For these reasons, we consider the requirement in Definition 6.7 to be appropriate for practical transformation definition. They even represent a weaker notion than what we aim to achieve in practice, because the requirement only bases on the sizes of the models and their maximal consistent subsets but not their actual contents. In practice, the consistent subsets before executing a transformation will be a subset of those after executing a transformation, although this is not formally required by the definition.

Remark. The definition for partial-consistency-improving transformations is based on a notion of partial consistency that considers the *maximal* consistent subsets. In practice, the subsets of the models that are to be considered consistent may not necessarily be the maximal ones. It is possible that there are larger subsets that could be considered consistent, but due to the history of changes, other, smaller subsets actually represent the consistent subsets. The requirement in the formalization is, however, only necessary to have a unique subset that can be calculated from each model state and to make statements about. In practice, usually trace models are used to represent which elements are corresponding and thus witness consistency. Ensuring that the requirements of partial consistency improvement apply to the consistent subsets induced by that trace model, the previous and following insights are still applicable, as it is only necessary that partial consistency improves with each transformation execution step and finally reaches 1.

The given notion of partial consistency improvement is stronger than the intuitive notion of just requiring the application of an execution step to improve partial consistency according to the metric in Definition 6.5. Although expecting such an improvement also ensures that the execution steps are strongly monotone regarding partial consistency, it does not ensure that a partial consistency of 1 is reached after a finite number of execution steps. This is due to the possibility of just having an asymptotic approximation of 1, which can, e.g., be achieved by adding consistent elements that do not affect the existing elements in each step. Then the sizes of the maximal consistent subsets and the models themselves increase by the same value, thus partial consistency improves but never reaches 1.

Lemma 6.2 (Bidirectional Transformation Execution Termination)
Let $\mathfrak{t} = \langle \mathbb{CR}, \textit{Cpr}_{\mathbb{CR}}^{\rightarrow}, \textit{Cpr}_{\mathbb{CR}}^{\leftarrow} \rangle$ *be a partial-consistency-improving bidirectional transformation. Then Algorithm 6.1 terminates for every input.*

Proof. The while loop of the algorithm consecutively applies an execution step of the bidirectional transformation $\mathfrak{t}$. The algorithm terminates when at some point a return statement is executed, thus either an execution step cannot be executed and returns $\bot$, or the loop condition is not fulfilled anymore. To quit the loop, the model pair $\langle \delta_{M_1}(m_1), m_2 \rangle$ needs to be consistent. m_1, m_2, and δ_{M_1} are the results of an execution step of $\mathfrak{t}$, to which the values m_1, m_2, and δ_{M_1} of the previous iteration were given. We know that $\langle \delta_{M_1}(m_1), m_2 \rangle$ *consistent to* $\mathbb{CR}$ if, and only if, their partial consistency is 1, i.e., $\textsc{Cons}_{\mathbb{CR}}(\delta_{M_1}(m_1), m_2) = 1$. Partial consistency is 1 if, and only if, the sizes of the maximal consistent subsets are equal to the sizes of the models themselves, i.e., when $|\delta_{M_1}(m_1)^p| + |m_2^p| = |\delta_{M_1}(m_1)| + |m_2|$. To show that partial consistency reaches 1, we consider the development of the size differences of the maximal consistent subsets and the models during the execution of the algorithm. We start with the initial size difference:

$$\textit{sizeDifference}_0 \coloneqq |\delta_{M_1}(m_1)| + |m_2| - |\delta_{M_1}(m_1)^p| - |m_2^p|$$

It is $\mathit{sizeDifference}_0 \geq 0$, because the models are always larger than their maximal consistent subsets. In the i-th iteration of the loop, we start with models m_1^{i-1}, m_2^{i-1} and change $\delta_{M_1}^{i-1}$, and the execution step returns m_1^i, m_2^i, and $\delta_{M_1}^i$. Then we have the size differences before this iteration, i.e., the difference after iteration $i-1$, and after this iteration, as:

$$\begin{aligned} \mathit{sizeDifference}_{i-1} &:= |\delta_{M_1}^{i-1}(m_1^{i-1})| + |m_2^{i-1}| - |\delta_{M_1}^{i-1}(m_1^{i-1})^p| - |m_2^{i-1,p}| \\ \mathit{sizeDifference}_{i} &:= |\delta_{M_1}^{i}(m_1^{i})| + |m_2^{i}| - |\delta_{M_1}^{i}(m_1^{i})^p| - |m_2^{i,p}| \end{aligned}$$

The reduction of the size difference in the i-th iteration is given by:

$$\begin{aligned} &\mathit{sizeDifferenceReduction}_i := \mathit{sizeDifference}_i - \mathit{sizeDifference}_{i-1} \\ &\quad = |\delta_{M_1}^{i}(m_1^{i})| + |m_2^{i}| - |\delta_{M_1}^{i}(m_1^{i})^p| - |m_2^{i,p}| \\ &\quad\quad - \left(|\delta_{M_1}^{i-1}(m_1^{i-1})| + |m_2^{i-1}| - |\delta_{M_1}^{i-1}(m_1^{i-1})^p| - |m_2^{i-1,p}|\right) \end{aligned}$$

We know that $\mathit{sizeDifferenceReduction}_i > 0$, because $\mathfrak{t}$ is partial-consistency-improving. Because of the model sizes being natural numbers, we know:

$$\mathit{sizeDifferenceReduction}_i \geq 1$$

So we can calculate the remaining size difference in the i-th iteration by applying all size difference reductions starting from $\mathit{sizeDifference}_0$:

$$\begin{aligned} \mathit{sizeDifference}_i &= \mathit{sizeDifference}_0 - \sum_{k=1}^{i} \mathit{sizeDifferenceReduction}_k \\ &\leq \mathit{sizeDifference}_0 - \sum_{k=1}^{i} 1 = \mathit{sizeDifference}_0 - i \end{aligned}$$

Thus, we have $\mathit{sizeDifference}_i \leq 0$ if $i \geq \mathit{sizeDifference}_0$. In fact, we have $\mathit{sizeDifference}_i = 0$, because $\mathit{sizeDifference}_i \geq 0$ by definition. Thus, models are consistent after at most $\mathit{sizeDifference}_0$ loop iterations. Since $0 \leq \mathit{sizeDifference}_0 < \infty$, the algorithm leaves the loop after a finite number of iterations. Note that, for reasons of simplicity, we have ignored that $\mathit{sizeDifferenceReduction}_i = 0$ if models were already consistent in iteration $i-1$ and thus models and their maximal consistent subsets have equal size in iterations $i-1$ and i. □

With Lemma 6.2, we know that we are able to execute transformations for given models that are not initially consistent such that their execution terminates in a consistent state whenever possible, as long as these transformations fulfill the property of being partial-consistency-improving. Note that this property substitutes the correctness property of consistency preservation rules. In fact, the original correctness notion is a special case of being partial-consistency-improving, because in that case one execution of a consistency preservation rules leads to a completely consistent pair of models.

We thus found a requirement for transformations that enables us to repeatedly apply their execution step to consecutively improve consistency until the models are finally consistent again. Based on this requirement, we can define a process for integrating changes to both involved models to finally yield consistent models. The requirement is, however, still only a theoretic requirement. Although it conforms to an intuitive expectation regarding transformations, it does not provide any assistance in how to be achieved in practice. We discuss this in the subsequent section.

6.3.4. Synchronizing Execution of Transformations

We have discussed how and under which conditions unidirectional consistency preservation rules can be executed iteratively to restore consistency between two models. The approach is, theoretically, able to process changes to models that are initially arbitrarily inconsistent. For practical applicability, we restricted the approach to initially consistent models and a change to one of them introducing an inconsistency. The transformation then iteratively improves partial consistency until consistent models are delivered.

Since we want to consider the case that both models instead of only one of them have been modified, we extend the approach to process changes to both models. More precisely, we introduce a modified notion of transformation execution steps that is able to process changes to both models. The operation of such an execution step is depicted in Figure 6.6. To this end, the first executed consistency preservation rule is applied to the first model and the change to it, but receives the modified state of the second model. We have motivated the necessity not to apply the first consistency preservation rule to the unmodified second model in Subsection 6.2.2. Afterwards, we apply the second consistency preservation rule to the modified first model, the original

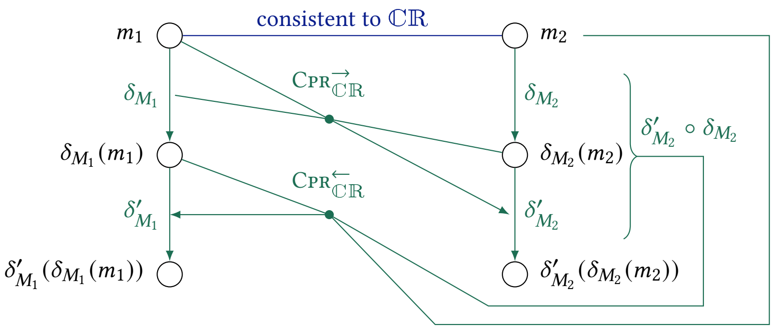


Figure 6.6.: Operation of a synchronizing bidirectional transformation execution step. The blue line without arrowheads connects elements that are consistent to $\mathbb{CR}$, and green lines with arrowheads indicate changes or consistency preservation execution.

second model, and the modifications to the second model as the concatenation of the original change and the one generated by the first consistency preservation rule. This ensures that all inconsistencies are introduced by changes processed by the consistency preservation rules, which was our requirement for practical applicability, as it requires to only react to changes instead of processing arbitrarily inconsistent models states.

Definition 6.8 (Synchronizing Bidirectional Execution Step)
Let $\mathfrak{t} = \langle \mathbb{CR}, \textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}, \textsc{Cpr}_{\overleftarrow{\mathbb{CR}}} \rangle$ be a bidirectional transformation for metamodels M_1 and M_2. A *synchronizing execution step* $\textsc{SyncEx}^1_{\mathfrak{t}}$ of $\mathfrak{t}$ is a function:

$$\textsc{SyncEx}^1_{\mathfrak{t}} : (I_{M_1}, I_{M_2}, \Delta_{M_1}, \Delta_{M_2}) \rightarrow (I_{M_1}, I_{M_2}, \Delta_{M_1}) \cup \{\bot\}$$

$$(m_1, m_2, \delta_{M_1}, \delta_{M_2}) \mapsto \begin{cases} (m'_1, m'_2, \delta'_{M_1}) \\ \bot \end{cases} \qquad \textit{with:}$$

$$\delta'_{M_2} := \textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}(m_1, \delta_{M_2}(m_2), \delta_{M_1}) \qquad m'_1 := \delta_{M_1}(m_1)$$

$$\delta'_{M_1} := \textsc{Cpr}_{\overleftarrow{\mathbb{CR}}}(m_2, m'_1, \delta'_{M_2} \circ \delta_{M_2}) \qquad m'_2 := \delta'_{M_2}(\delta_{M_2}(m_2))$$

If $\textsc{Cpr}_{\overrightarrow{\mathbb{CR}}}(m_1, \delta_{M_2}(m_2), \delta_{M_1}) = \bot$ or $\textsc{Cpr}_{\overleftarrow{\mathbb{CR}}}(m_2, m'_1, \delta'_{M_2} \circ \delta_{M_2}) = \bot$, the execution is undefined, i.e., $\textsc{SyncEx}^1_{\mathfrak{t}}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) = \bot$.

Algorithm 6.2 Synchronizing execution of a bidirectional transformation.

1: **procedure** $\textsc{ExecuteSync}(\mathfrak{t} = \langle \mathbb{CR}, \textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}, \textsc{Cpr}_{\mathbb{CR}}^{\leftarrow} \rangle, m_1, m_2, \delta_{M_1}, \delta_{M_2})$
2:     **if** $\neg(\langle m_1, m_2 \rangle$ *consistent to* $\mathbb{CR})$ **then**
3:         **return** $\perp$
4:     **end if**
5:     **if** $\neg(\langle \delta_{M_1}(m_1), \delta_{M_2}(m_2) \rangle$ *consistent to* $\mathbb{CR})$ **then**
6:         $(m_1, m_2, \delta_{M_1}) \leftarrow \textsc{SyncEx}_{\mathfrak{t}}^{1}(m_1, m_2, \delta_{M_1}, \delta_{M_2})$
7:         **if** $(m_1, m_2, \delta_{M_1}) = \perp$ **then**
8:             **return** $\perp$
9:         **end if**
10:     **end if**
11:     **while** $\neg(\langle \delta_{M_1}(m_1), m_2 \rangle$ *consistent to* $\mathbb{CR})$ **do**
12:         $(m_1, m_2, \delta_{M_1}) \leftarrow \textsc{Ex}_{\mathfrak{t}}^{1}(m_1, m_2, \delta_{M_1})$
13:         **if** $(m_1, m_2, \delta_{M_1}) = \perp$ **then**
14:             **return** $\perp$
15:         **end if**
16:     **end while**
17:     **return** $\langle \delta_{M_1}(m_1), m_2 \rangle$
18: **end procedure**

The synchronizing bidirectional execution step is necessary to first integrate the changes made in both models. It only produces a change in the first model, such that afterwards ordinary execution steps that only need to deal with a change to one model can be applied. This leads to Algorithm 6.2 for the synchronizing execution of a bidirectional transformation. It is an extension of Algorithm 6.1 for the non-synchronizing case. Thus, it has the same properties regarding termination and return values.

Theorem 6.3 (Synchronizing Transformation Termination)
Let $\mathfrak{t}$ be a partial-consistency-improving bidirectional transformation. Then Algorithm 6.2 terminates for every input and either returns $\perp$ or a consistent model pair.

Proof. The algorithm is identical to Algorithm 6.1 except for Lines 5–10, which add the initial synchronization step. These lines add a single return statement that can return $\perp$. The return statement in Line 17 not returning $\perp$ is still preceded by the while loop having the loop condition that the model pair needs to be inconsistent. Thus, the argument of the proof for Lemma 6.1 ensuring that only consistent models are returned still applies. In consequence, we know that the algorithm either returns $\perp$ or a consistent model pair.

Termination of the algorithm is guaranteed for the non-synchronizing case proven in Lemma 6.2. Although the additional execution of $\textsc{SyncEx}_{\mathfrak{t}}^{1}$ may introduce further inconsistencies, the proof already considered that the models given to the while loop may be arbitrarily inconsistent. Thus, the inductive improvement in partial consistency through the while loop is given in the same way and, thus, the models finally become consistent. □

We have proven that a bidirectional transformation that is partial-consistency-improving can be executed for two given models and changes to both of them such that consistent models are delivered, as long as the transformation can process the changes. In fact, we have already restricted the algorithm such that it does not need to deal with arbitrarily inconsistent models but with models that are initially consistent, such that only the given changes introduce inconsistencies. This is supposed to ease the definition of transformations that fulfill the property of being partial-consistency-improving in practice, as they can rely on the assumption that inconsistency is only introduced by the given changes.

With the insight that partial-consistency-improving bidirectional transformations can be used to integrate changes to both of two models and deliver consistent models based on those changes, we define *synchronizing bidirectional transformations* as bidirectional transformations with the property of being partial-consistency-improving.

> **Definition 6.9** (Synchronizing Bidirectional Transformation)
>
> Let $\mathfrak{t}$ be a partial-consistency-improving bidirectional transformation. Then we call $\mathfrak{t}$ a *synchronizing bidirectional transformation.*

As discussed in Subsection 6.3.2, we have defined bidirectional transformation execution steps starting with $\textsc{Cpr}_{\mathbb{CR}}^{\rightarrow}$, although it can also be necessary to start with $\textsc{Cpr}_{\mathbb{CR}}^{\leftarrow}$ depending on which model was changed. We have discussed that the order restriction is without loss of generality and that definitions can be transferred by swapping the rules. For the synchronization case, in which both models have been modified, the execution order does, theoretically, not even make a difference, because changes to both models are present. From a practical perspective, it can, however, make sense to define which of the consistency preservation rules to execute first. For example, it might make sense to first execute the consistency preservation rule from the more abstract to the more detailed model, if such a relation exists between the models. We leave such considerations up to the individual transformation developer or future research, as the selection of the order does not provide any conceptual benefits but, in the best case, only eases the definition of appropriate consistency preservation rules and improves usability.

6.3.5. Equivalence to Synchronizing Transformations

For our definition of transformation networks, we have used the notion of synchronizing transformations (see Definition 4.7). Its single consistency preservation rule accepts two consistent models as well as a change to each of them and returns two changes that, if applied to the models, result in consistent models again. Synchronizing bidirectional transformations, i.e., the just defined transformations composed of unidirectional consistency preservation rules, also accept two consistent models and a change to each of them and return two consistent models. We could also define those transformations to return changes rather than the consistent models by concatenating the changes calculated by the execution steps. For reasons of simplicity, we have omitted that in the formalization.

Although synchronizing transformations and synchronizing bidirectional transformations have the same requirements for their inputs and provide the same guarantees regarding consistency for their outputs, both may also return $\perp$. While a synchronizing transformation can be defined such that it never returns $\perp$ by defining a consistency preservation rule that is total, the ability of a synchronizing bidirectional transformation to never return $\perp$ depends on the interplay of the two unidirectional consistency preservation rules. Nevertheless, we can show that both have equal expressiveness, i.e., they can always return the same results for the same inputs.

Theorem 6.4 (Synchronizing Transformation Expressiveness)
Synchronizing bidirectional transformations and synchronizing transformations have equal expressiveness, i.e., each synchronizing transformation can be expressed by a synchronizing bidirectional transformation and vice versa.

Proof. Each synchronizing bidirectional transformation can be realized by a synchronizing transformation by simply defining the function of the consistency preservation rule such that it returns the result that is produced by the execution of the synchronizing bidirectional transformation. Let $\mathfrak{t}$ be a synchronizing bidirectional transformation with:

$$\textsc{ExecuteSync}(\mathfrak{t}, m_1, m_2, \delta_{M_1}, \delta_{M_2}) = (m_1', m_2')$$

We define the consistency preservation rule $\textsc{Cpr}$ of a synchronizing transformation for each possible input as:

$$\begin{aligned}&\textsc{Cpr}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) \coloneqq (m_1, m_2, \delta_{M_1}', \delta_{M_2}')\\&\quad \textit{with } \delta_{M_1}'(m_1) \coloneqq m_1' \wedge \delta_{M_2}'(m_2) \coloneqq m_2'\end{aligned}$$

Per definition, applying the resulting changes to the input models, the synchronizing transformation delivers for every possible input the same result by applying $\textsc{Cpr}$ as the synchronizing bidirectional transformation.

Realizing a synchronizing transformation by a synchronizing bidirectional transformation requires the repeated execution of the two consistency preservation rules to emulate the behavior of the single synchronizing consistency preservation rule. Let $\textsc{Cpr}$ be the consistency preservation rule of a synchronizing transformation with:

$$\textsc{Cpr}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) = (m_1, m_2, \delta_{M_1}', \delta_{M_2}')$$

Then we can define the two unidirectional consistency preservation rules of the synchronizing transformation $\mathfrak{t}$ as follows.

$$\begin{aligned}&\textsc{Cpr}^{\rightarrow}(m_1, \delta_{M_2}(m_2), \delta_{M_1}) \coloneqq \delta_{M_2}^b && \textit{with } \delta_{M_2}^b(\delta_{M_2}(m_2)) \coloneqq \delta_{M_2}'(m_2)\\&\textsc{Cpr}^{\leftarrow}(m_2, \delta_{M_1}(m_1), \delta_{M_2}^b \circ \delta_{M_2}) \coloneqq \delta_{M_1}^b && \textit{with } \delta_{M_1}^b(\delta_{M_1}(m_1)) \coloneqq \delta_{M_1}'(m_1)\end{aligned}$$

So we simply define the two consistency preservation rules in a way such that each of them delivers for the inputs in the synchronizing execution step $\textsc{SyncEx}^1_{\mathfrak{t}}$ those changes that are necessary to produce exactly the results of the consistency preservation rule $\textsc{Cpr}$ of the synchronizing transformation. Then, according to the behavior of $\textsc{SyncEx}^1_{\mathfrak{t}}$, we have:

$$
\begin{aligned}
&\textsc{SyncEx}^1_{\mathfrak{t}}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) = (m^s_1, m^s_2, \delta^s_{M_1}) \ \mathit{with}\\
&\quad \delta^s_{M_1}(m^s_1) = \delta^s_{M_1}(\delta_{M_1}(m_1)) = \textsc{Cpr}^{\leftarrow}(m_2, \delta_{M_1}(m_1), \delta^b_{M_2} \circ \delta_{M_2})(\delta_{M_1}(m_1))\\
&\qquad = \delta^b_{M_1}(\delta_{M_1}(m_1)) = \delta'_{M_1}(m_1)\\
&\quad \wedge m^s_2 = \textsc{Cpr}^{\rightarrow}(m_1, \delta_{M_2}(m_2), \delta_{M_1})(\delta_{M_2}(m_2))\\
&\qquad = \delta^b_{M_2}(\delta_{M_2}(m_2)) = \delta'_{M_2}(m_2)
\end{aligned}
$$

So $\textsc{SyncEx}^1_{\mathfrak{t}}$ produces m^s_1, m^s_2, and $\delta^s_{M_1}$, for which we know that $\delta^s_{M_1}(m^s_1)$ and m^s_2 are consistent, because their equivalents $\delta'_{M_1}(m_1)$ and $\delta'_{M_2}(m_2)$ are consistent by assumption. Thus, the execution of the synchronizing bidirectional transformation $\mathfrak{t}$ according to Algorithm 6.2 terminates after the conditional statement in Line 5 with the same consistent models that are delivered when applying the changes calculated by the consistency preservation rule $\textsc{Cpr}$ of the assumed synchronizing transformation.

With these construction approaches, we have shown that each synchronizing transformation can be expressed by a synchronizing bidirectional transformation and vice versa. $\square$

Although we have proven that each synchronizing transformation can be expressed by a synchronizing bidirectional transformations and thus the latter ones can be used to express any desired consistency preservation in a transformation network, the constructive proof does not reflect a practical construction approach for the unidirectional consistency preservation rules of a synchronizing bidirectional transformation. In practice, it will usually not be possible to define the rules in a way that they deliver consistent models after executing each of them once, as we have already discussed in Subsection 6.2.3. It shows, however, that it would possible in theory.

Based on the knowledge that we can use synchronizing bidirectional transformations in transformation networks, we discuss in the following how a transformation developer can actually achieve that the specification of a bidirectional transformation in terms of two unidirectional consistency preservation rules does actually fulfill the requirements of being partial-consistency-improving and thus represents a synchronizing bidirectional transformation that can be used in a transformation network.

6.4. Achieving Synchronization

We have introduced the notion of synchronizing bidirectional transformations, which can be used within transformation networks in place of synchronizing transformations. They are composed of two unidirectional consistency preservation rules, which fits to the way how transformations are specified in transformation languages. In contrast to only being correct, as commonly required of transformations, they need to fulfill the notion of being partial-consistency-improving to be used instead of synchronizing transformations.

The knowledge about this requirement, theoretically, gives a transformation developer the ability to define appropriate transformations to be used in transformation networks. We have discussed that the requirement for transformations to be partial-consistency-improving is reasonable, as it reflect intuitive requirements to transformations to always restore more consistency than is violated by their execution. There is, however, still no canonical way to fulfill that requirement. It may be possible to define analyses for transformations or even appropriate transformation languages that guarantee the property by construction. This could, however, even lead to severe restrictions in expressiveness if analyzability is the primary goal. In addition, research about synchronizing concurrent changes (e.g. [Her+12; OPN20; Xio+13; Xio+09] already addresses a comparable problem. Thus, we do not discuss or investigate such approaches in this thesis.

We leave it up to transformation developers to thoroughly define their transformations such that they fulfill the required property. Having precise knowledge about the property that needs to be fulfilled by the transformations already provides a benefit regarding the baseline of using ordinary transformations in a transformation network without knowing how the transformations have to be improved to work properly. In addition, we discuss a distinction of possible scenarios that can occur when changes need to be synchronized and come up with engineering considerations how to systematically deal with these scenarios. We identify one essentially problematic scenario and propose a strategy to avoid that problem by proper construction of transformations. In our evaluation in Chapter 9, we will see that it is actually the only occurring and thus most relevant problem scenario that transformation developers have to deal with when developing synchronizing bidirectional transformations.

6.4.1. Synchronization Scenarios

For the execution of synchronizing bidirectional transformations, we have assumed that inconsistencies are only introduced by changes. Thus, defining a consistency preservation rule that processes changes in one model must consider that it has to deal with the situation that the other model has been changed as well. Although this might intuitively lead to the expectation that distinguishing the different types of changes, such as element insertions and removals, helps to identify relevant scenarios, actually the modification of condition elements of the consistency relations rather than individual elements is relevant.

If we process a change δ_{M_1} to model m_1, and if m_2 was changed by δ_{M_2} as well, a consistency preservation rule $\textsc{Cpr}^{\rightarrow}$ from M_1 to M_2 of a synchronizing bidirectional transformation $\mathfrak{t}$ produces a change δ'_{M_2} in the execution of the synchronizing execution step $\textsc{SyncEx}^{1}_{\mathfrak{t}}$. If we assume that δ_{M_1} performs a change that introduces a new condition element, $\textsc{Cpr}^{\rightarrow}$ is responsible for adding a corresponding element to $\delta_{M_2}(m_2)$ such that partial consistency between the two is improved and, in the best case, already restored to 1. $\textsc{Cpr}^{\rightarrow}$ must also consider the change δ_{M_2}, which may have already added an appropriate corresponding element, such that adding a further one may reduce instead of improve partial consistency. Adding a condition element to a model can, however, not only be the result of adding an element but also of different types of changes, such as also the change of an attribute or reference value. In fact, it must only be considered that a condition element was added but not which kind of change introduced it.

We have already discussed in Subsection 6.1.2 that the addition, removal, and change of condition elements are the relevant scenarios that can lead to consistency violations. In case of adding a condition element, an appropriate corresponding element for it may be missing, such that no witness structure for consistency is given. This requires an appropriate element to be added. In case of removing a condition element, the element was corresponding to another one, which now has no corresponding element anymore. This requires the corresponding condition element to be removed. Changing a condition element can be seen as a modification of model elements such that they represent another condition element of the same condition, thus still belonging to the same consistency relation. The consistency preservation rule must then update the corresponding condition element appropriately.

This behavior is what consistency preservation rules are actually supposed to implement. A bidirectional transformation with such preservation rules is inherently supposed to fulfill the property of being partial-consistency-improving, because the elements that have no corresponding elements due to a change are not part of the maximal consistent subsets before executing the consistency preservation rule. After executing it, either the corresponding element is removed and thus the model size decreases, or a corresponding element is added and the size of the maximal consistent subsets improves.

In addition to the above considerations, a transformation may be prevented from being partial-consistency-improving, because the addition or removal of a condition element to improve consistency affects further condition elements. This can occur because these condition elements overlap, i.e., some model elements may be part of several condition elements. Then, if all elements of a condition element are removed, the other condition element is not present anymore as well. A consistency preservation rule must thus be carefully defined such that removing one condition element does not lead to the removal of another one, which was actually part of the maximal consistent subset. Otherwise, the consistency preservation rule introduces a new violation of consistency. The same applies to the scenario of adding condition elements. If the addition leads to the introduction of an additional condition element, because some objects in the added condition element together with other existing objects form a condition element of another consistency relation, this introduces an inconsistency if no corresponding element exists yet, thus reducing partial consistency. If the previously existing elements within the induced condition element were part of the maximal consistent subset, the consistency preservation rule is actually not correct. If the models were consistent before and only the change to one model is performed, correctness of the consistency preservation rule requires the result to be consistent. However, it introduces a further condition element that has no corresponding element, thus the result is not consistent. If, on the other hand, the previously existing elements within the induced condition element were not part of the maximal consistent subset, it is fine that these elements are still inconsistent, as the consistency preservation rules still need to process them anyway. These problems are comparable to those of fine-grained transformation rules, as discussed in Subsection 4.4.1, which need to be defined such that one rule does not lead to the violation of the consistency relation of another.

The previous considerations reflected the case that only one model was changed. If the other model was changed as well, the combinations of changes can lead to specific situations that have to be handled differently. We therefore distinguish the addition, removal, and change of a condition element to be processed by the consistency preservation rule and discuss what conflicts may occur by changes performed in the other model. Changes of condition elements are, in practice, traced by the usage of trace models that store trace links between corresponding elements. It can be seen as a representation of the witness structure we have defined for identifying consistency. If elements become changed, the trace links still exist and indicate which corresponding elements need to be adapted. According to the defined notion of consistency, these potential conflicts are just based on the question whether appropriate condition elements exist or not.

Addition: Whenever a condition element is added to one model, it must be ensured that a corresponding condition element in the other model exists. In the case that both models were consistent before, the corresponding element cannot already be present in the other model and thus has to be added. If the other model has been changed, an appropriate corresponding element may have already been added. That scenario has to be explicitly considered to avoid a duplicate creation of the condition element, which then may lead to a violation of consistency that cannot be resolved by adding further elements anymore.

Removal: Whenever a condition element is removed from one model, the corresponding condition element must be removed from the other model, as otherwise its corresponding element is missing, which would violate consistency. If the models were consistent before, it is guaranteed that the corresponding element exists and can thus be removed. If the corresponding condition element is not present, because it was already removed from the other model, the element cannot but also does not need to be removed anymore. It must only be considered that the existence of the corresponding element cannot be assumed.

Change: When model elements are changed such that they represent a different condition element of the same condition as before, they usually also require the corresponding element to be updated to represent the condition element of an applicable consistency relation pair. If the corresponding element was removed, the consistency preservation rule in the opposite direction will remove the changed condition element anyway to restore consistency. Thus, the consistency preservation rule must only consider that the corresponding element may have been removed but does not need to perform changes. If the corresponding element was changed, which is identified by the trace model still containing a link to a changed element, it must be adapted such that both elements form a consistency relation pair again. The modification to the corresponding element will then be propagated back by the opposite consistency preservation rule.

In summary, we have to deal with two specific situations that can occur when the target model of a consistency preservation rule may have been changed. First, when adding condition elements, their corresponding elements may already exist in the other model. Second, when removing condition elements, their corresponding elements may have already been removed from the other model. While the second scenario is easy to handle by doing nothing whenever the corresponding elements of removed elements are not present anymore, the first scenario requires an approach to identify whether corresponding elements already exist. While existing corresponding elements can be retrieved from a trace model, no trace links exist for these newly created elements. In the following, we thus discuss an approach to find corresponding elements.

6.4.2. Identification of Existing Corresponding Elements

Whenever a condition element is added, which requires a corresponding element to exist in the other model, the consistency preservation rule will usually create appropriate elements in the other model. This is due to the reason that in the case when the target model has not been modified as well, these elements cannot already exist. In the synchronization case, however, the change to the target model may have already introduced those elements, thus it is necessary to find them to avoid their duplicate creation.

In previous work [Kla+19b], we have proposed a strategy to identify such corresponding elements. Transformation languages usually use trace models to store the information about which elements are corresponding to each other. Thus, whenever the consistency preservation rule in the opposite direction added the element whose addition is currently processed, a trace link already exists. When the corresponding elements were created by different transformations, however, there will not be a trace link between them.

An intuitive attempt would be to use the trace links of the other transformations across which they were created. For example, if for a PCM component a UML class is created, and for this UML class a Java class is created, then there are trace links between the PCM component and the UML class, as well as between the UML class and the Java class. Synchronizing the addition of the PCM component and the Java class should not result in a redundant addition of a further Java class. Resolving the existing trace links transitively is, however, not a solution. In this case, a unique one-to-one mapping exists that actually traces the PCM component to the corresponding Java class. It would, however, also be possible that a PCM component has trace links to several elements in the Java model across the UML. If those elements are even multiple classes, such as one public and one internal utility class, but the consistency relation between PCM and Java only requires one Java class for a PCM component, it would be unclear which to select.

Transformation languages usually tag trace links with additional information, for example, containing the transformation rule that created them, to distinguish links to instances of the same class. Since these tags are created by other transformations, considering them would violate our assumption of independent development and modular reuse of transformations. Even worse, it could also be the case that another third class is required by the consistency relation between PCM and Java. Finally, it is up to the actual consistency relation to define when elements are to be considered corresponding, because there may be more semantics beyond the types of the elements related by a trace link that determines how they belong together.

Thus, whether corresponding elements already exist cannot be identified by transitively resolving trace links of other transformations but only by considering the two involved models. The information to identify whether elements can be considered corresponding is precisely given in the consistency relation. For example, if the relation specifies that, in a very simplified notion, a PCM component is consistent to all Java classes that have the same name, no matter what implementation the class contains, then if any class with the name of the PCM component is found in the Java code, it can be considered corresponding.

We come up with the following three levels of identifying corresponding elements.

Explicit Unique: The information that elements correspond is unique and represented explicitly, e.g., within a trace model.

Implicit Unique: The information that elements correspond is unique but represented implicitly, e.g., in terms of key information within the models, such as element names.

Non-Unique: Without unique information, heuristics based on ambiguous information or transitive resolution of indirect trace links must be used.

In the best case, a trace link already exists between the corresponding elements. This can be because a consistency preservation rule in one direction created the corresponding element and added the trace link. Then the consistency preservation rule in the other direction processes the change that introduced the corresponding element but now can already retrieve the trace link. This is what we call *explicit unique* information, because the information is represented explicitly and unambiguously in the trace model.

If no trace link exists, like in the synchronization scenario, the information specified in consistency relations to identify corresponding elements needs to be used. This can be considered key information, because the information is used as the key to identify corresponding elements. To this end, the model has to be queried for elements with the given information. The transformation language QVT-R already provides a language construct to specify such key information within transformation rules [QVT, Sec. 7.10.2.]. We call this information *implicit unique*, because elements can be unambiguously identified by implicit information within the models. Note that in case that multiple corresponding elements match the key information, any of them can be selected. It is up to the consistency preservation rule for the other direction to add further elements such that corresponding elements for all of them are added, such that a valid witness structure is induced.

In the worst case, no unique information is given. Precisely following our formalism, this scenario can never occur, because each consistency relation defines the necessary key information. Thus, this scenario can only occur

Algorithm 6.3 Retrieval of corresponding elements.

procedure FindCorresponding($CR, \mathfrak{c}_l, m_2, m_{traces}$)
    $tracedElements \leftarrow \{\mathfrak{c}_r \mid \langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in m_{traces}\}$
    for $\mathfrak{c}_r \in tracedElements$ **do**
        if $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR$ **then**
            return $\mathfrak{c}_r$
        end if
    end for
    for $\mathfrak{c}_r \in \mathcal{P}(m_2)$ **do**
        if $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR$ **then**
            $m_{traces} \leftarrow m_{traces} \cup \{\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle\}$
            return $\mathfrak{c}_r$
        end if
    end for
    return $\perp$
end procedure

in practice with a relaxed notion of consistency. This can be the case when for an element a corresponding one is always created, containing some related information, but no unique information to identify that the two are corresponding is given. In that case, only trace links identify that the elements are corresponding. Thus, if other transformations created the elements and thus no direct trace link exists, it is impossible to identify that these elements shall be corresponding. Since no information to identify that the elements should be corresponding is present anyway and since this requires a relaxed consistency notion, we assume this scenario unlikely to occur at all and did not face it in our evaluation at any time. If, nevertheless, this scenario occurs, only heuristics can be used to identify corresponding elements without any guarantee of success. It would also be possible to involve the developer and let him decide whether an element should be considered corresponding.

In summary, it is necessary that transformation developers use key information for identifying corresponding elements based on *implicit unique* information in addition to the usage of *explicit unique* information in terms of trace links. In case that corresponding elements are found based on implicit unique information, they need to establish a trace link for the elements. We define this behavior in Algorithm 6.3, which is an extended version of an algorithm [Sağ20, Alg. 1] defined in the Master's thesis of Sağlam, which was supervised by the author of this thesis, and adapted to our formalism.

Algorithm 6.3 receives the consistency relation for which corresponding elements shall be found, the condition element $\mathfrak{c}_l$ of the condition $\mathbb{C}_{l,CR}$ that was added to model m_1 for which corresponding elements shall be found or created, the model m_2 in which the corresponding elements shall be searched, and the trace model $m_{traces} \subseteq \mathcal{P}(m_1) \times \mathcal{P}(m_2)$ containing pairs of elements in m_1 and m_2, which represents a combination of witness structures for consistency relations between metamodels M_1 and M_2. The algorithm first retrieves all corresponding elements for the condition element from the trace model and then, in Line 3, checks whether any of the corresponding elements according to the trace model is a corresponding element in CR. If this is the case, a corresponding element $\mathfrak{c}_r$ is found and the procedure returns $\mathfrak{c}_r$. Otherwise, model m_2 is browsed for the existence of a corresponding element in the loop starting in Line 8. It considers all subset of m_2, i.e., the potency set $\mathcal{P}(m_2)$, of which each could be such a corresponding element. If one of them is corresponding according to CR, then the pair $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle$ is added to the trace model m_{traces} as an appropriate trace link and the procedure returns the found element $\mathfrak{c}_r$. If no such element is found, the procedure returns $\perp$ to indicate that no corresponding element is found and thus has to be created by the consistency preservation rule.

The loop in Line 8 is defined in a rather inefficient way but describes its purpose in the most general way. In a practical implementation, it may not consider every subset of the model m_2 but instead retrieve all candidate elements, for example, by filtering model elements by their class. Depending on the modeling framework, different possibilities to efficiently find specific elements can be used. The implementation of the EMF, for example, provides functions that yield all instances of a specific class.

The transformation developer has to apply this algorithm every time he or she specifies the creation of corresponding elements due to a change adding a condition element. This ensures that applying the bidirectional transformation to the synchronization case properly handles the situation that a change has already created the corresponding elements to ensure that the resulting transformation is partial-consistency-improving.

In contrast to the insights of the previous sections, the engineering considerations we have made in this section are not completely formally founded and proven. We have not proven that if a transformation developer follows the discussed rules for the construction of consistency preservation rules and applies the FINDCORRESPONDING function whenever condition elements are created leads to a synchronizing bidirectional transformation, i.e., a bidirectional transformation that fulfills the requirement of being partial-consistency-improving. We derived the insights from thorough argumentation but further validate them in the evaluation in Chapter 9.

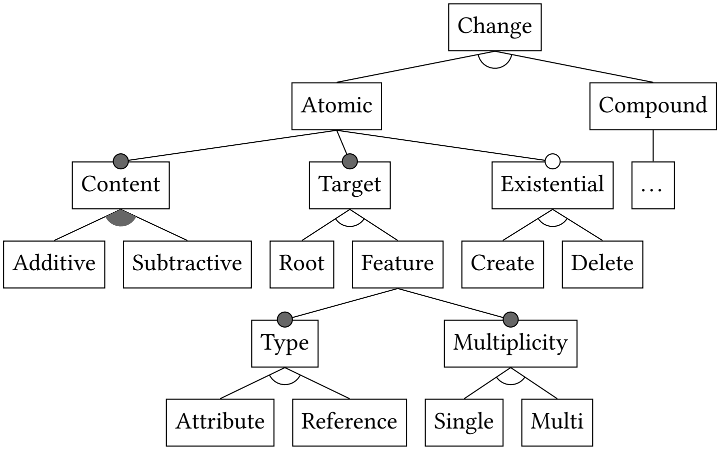
6.4.3. Model Changes To Condition Element Changes

The previous discussions distinguished different change scenarios for condition elements, as those are relevant when considering the synchronization case of bidirectional transformations. A transformation does, however, not receive changes of condition elements but changes of actual model elements. These then eventually lead to the addition, removal, or change of a condition element. Thus, a transformation developer needs to decide which model changes introduce which modifications of condition elements to determine appropriate behavior of the consistency preservation rules.

The possible types of model changes are induced by the used modeling formalism, as the meta-metamodel defines which types of changes can be performed in models. Our modeling formalism introduced in Section 3.3 is conforming to the EMOF, which is why we consider changes in EMOF- and Ecore-based models. Kramer proposes feature models that express all kinds of possible changes in EMOF-based models [Kra17, Fig. 5.2] and Ecore-based models [Kra17, Fig. 5.3]. Since both are rather similar (see Subsection 2.2.2), we focus on Ecore as the practically realized modeling formalism. We depict a modified version of the feature model for changes in Ecore-based models in Figure 6.7. In comparison to the original model [Kra17, Fig. 5.3], we have made the following changes.

No Compound Changes: We do not consider compound changes, because they are simply compositions of atomic changes and thus do not need to be considered explicitly.

No Permutation: We removed the *Permutation* feature, because it can be considered as a compound change of a subtractive and additive multi-valued feature change. Whether or not permutation rather than the removal and addition is relevant is up to the interpretation of the change



Constraints:

1. Single $\Rightarrow$ (Additive $\wedge$ Subtractive)
2. Multi $\Rightarrow$ (Additive $\oplus$ Subtractive)
3. Root $\Rightarrow$ (Additive $\oplus$ Subtractive)
4. Existential $\Rightarrow$ (Root $\oplus$ Reference)
5. Create $\Rightarrow$ Additive
6. Delete $\Rightarrow$ Subtractive

Figure 6.7.: Feature model for changes in Ecore-based models. Adapted from [Kra17, Fig. 5.3].

sequence and is comparable to moving an element from one reference to another, which is also modeled as a compound change.

Mandatory Content: We made the *Content* feature mandatory, because every change is either additive or subtractive due to the removal of the permutation.

Constraints Reduction: We reduced the constraints to those that are still relevant after performing the previously discussed changes.

Error Correction: We fixed an error in the constraints of the original model. They required a *Create* change of a root element to be subtractive, which does not make sense. We corrected that error by simplification.

The set of all possible change types in Ecore-based models is given by enumerating all valid configurations of the feature model. We discuss for each of the resulting changes the types of condition element changes it may induce.

Additive Root Change (Possibly Create): Adding a root element can lead to the addition of a condition element, which consists only of this root element. It will not induce a change or removal of a condition element.

Subtractive Root Change (Possibly Delete): The removal of a root element can lead to the removal of a condition element, which involves the root element. Since it removes an element, it can neither lead to a change nor to an addition of a condition element.

Single-Valued Attribute Change: Changing an attribute can lead to either an addition, removal, or change of a condition element. The change may lead to an element that now, potentially together with other elements, forms a condition element. It may also lead to a different condition element of the same condition, e.g., by renaming an element. Finally, it can also lead to an element that is not present in a condition anymore. This applies no matter whether the attribute change is only additive, only subtractive, or both, thus whether it adds, removes, or replaces the attribute value.

Additive Multi-Valued Attribute Change: Adding a value to a multi-valued attribute can lead to either an addition, removal, or change of a condition element. The change can lead to the situation that the element is now part of a condition element, is not part of a condition element anymore, or that it represents a different condition element of the same condition and is thus comparable to the change of a single-valued attribute.

Subtractive Multi-Valued Attribute Change: Removing a value from a multi-valued attribute can lead to either an addition, removal, or change of a condition element, due to the same reasons as the additive multi-valued attribute change.

Single-Valued Reference Change (Possibly Create/Delete): The change of a reference can lead to either an addition, removal, or change of a condition element, due to the same reasons as for single-valued attribute changes. This is even independent from whether the added or removed element is created or deleted, respectively.

Additive Multi-Valued Reference Change (Possibly Create): The addition of a value to a multi-valued reference can lead to either an addition, removal, or change of a condition element, due to the same reasons as for adding an attribute to a multi-valued attribute. Like for single-valued reference changes, this is even independent from whether the element was created or already existed before.

Subtractive Multi-Valued Reference Change (Possibly Delete): If a value is removed from a multi-valued reference, this can lead to either an addition, removal, or change of a condition element, due to the same reasons as for removing an attribute from a multi-valued attribute. Like for single-valued reference changes and additive multi-valued reference changes, this is even independent from whether the element was created or already existed before.

It is easy to see that except for root changes each type of model change can lead to any kind of condition element change, because almost every type of change can lead to the situation that model elements form a condition element or do not form a condition element anymore. There may be a missing reference or attribute value, or even a superfluous reference or attribute value, after whose change the model elements form a condition element. This conforms to the notion of creating a corresponding element whenever all conditions for some model elements are fulfilled in the QVT-R-like *Mappings Language* [Kra17, p. 283]. Since all types of changes can lead to the fulfillment of conditions, the addition of a condition element is not tied to a specific type of change.

Depending on the specific consistency relation, there may, however, be some change types that are not relevant in that case. For example, if a consistency relation puts two model elements having only reference values into relation, then no attribute change will lead to the addition, removal, or change of a condition element of that consistency relation.

The specific case of identifying corresponding elements during synchronization discussed in the previous subsection needs to be considered whenever a condition element was added. Since this can occur because of any type of change except for removals of root elements, we cannot make any general restrictions on the types of model changes that need to be explicitly considered for the synchronization case. The transformation developer must decide after which changes a condition element may be created, independent from whether corresponding elements may already exist or not. Thus, he or she makes this decision anyway and must only extend the existing logic for finding corresponding elements according to the given algorithm.

6.5. Summary

In this chapter, we have discussed how synchronizing transformations, as required in transformation networks, can be defined with existing transformation languages. To this end, we have defined synchronizing bidirectional transformations as an extension of bidirectional transformations specified in transformation languages. We have formally proven that these transformations always terminate consistently and have equal expressiveness than synchronizing transformations. Finally, we have identified properties and proposed an algorithm to be implemented by a transformation specified in a transformation language to be synchronizing. We close this chapter with the following central insight.

Insight II.3 (Synchronization)

Synchronizing transformations, as required in transformation networks, process pairs of models that both may have been and need to be modified. In contrast, ordinary bidirectional transformations consist of two unidirectional consistency preservation rules, each of them accepting changes in one model and updating the other. We have shown that if changes have been performed to both models, the consistency preservation rules cannot be sequenced such that they produce consistent results. By requiring that a bidirectional transformation fulfills a notion of being *partial-consistency-improving*, we were able to define an execution algorithm for it that delivers consistent models after a finite number of execution steps. In return, we were able to formally prove that such transformations have equal expressiveness than synchronizing transformations as required for transformation networks. Finally, we found that a transformation developer needs to consider only few situations explicitly to make a bidirectional transformation partial-consistency-improving. The most important situation is that a transformation creates elements that already exist, because another transformation already created them, for which we provide an algorithm to avoid issues due to duplicate element creation already by construction. In consequence, synchronizing transformations can be constructed with existing transformation languages by fulfilling an additional property for which we provide a constructive strategy and without knowing about other transformations to combine them with.

7. Orchestrating Transformation Networks

A transformation network is composed of transformations and an application function, which executes the transformations in an order determined by an orchestration function. In the previous chapters, we have discussed how the individual transformations can be defined and which properties they have to fulfill to be properly usable in a transformation network. In this chapter, we discuss how the combination of transformations, as the second essential part of a transformation network, can be realized by an application function.

Although the behavior of an application function has already been defined in Definition 4.13, we have shortly discussed that we cannot require correctness for such a function in the sense that it always yields consistent models for every given models and changes to them. We will prove that statement and show that this can either be because there is no execution order of the given transformations that yields consistent models for given models and changes to them or, even if it exists, it may not be possible to find it.

In this chapter, we thus discuss under which conditions we can require an application function to return consistent models. We derive an algorithm that realizes an application function and prove that it is not possible to ensure its termination without further restrictions to the transformations or the cases in which the algorithm is expected to return consistent models. The discussion of different restriction options gives us the insight that none of them is practically applicable, because they restrict expressiveness of transformations and transformation networks too much. Thus, we finally propose an algorithm that operates conservatively. This means, if it returns models, they are consistent, but it may not always return consistent models although an execution order of transformations that yields them exists. This algorithm is supposed to improve the ability of a transformation developer to identify why no execution order of transformations could be found although it existed. We have envisioned this as the *comprehensibility* property in Subsection 1.1.3.

This chapter thus constitutes our contribution **C 1.4**, which consists of four subordinate contributions: a discussion of the design of an application function with possible bounds for the number of executions and a notion of optimality leading to the definition of the *orchestration problem*; the derivation of an algorithm for an application function, for which we discuss termination, prove undecidability of the orchestration problem, and discuss different strategies to restrict transformations such that the orchestration problem becomes decidable; a gradual definition of optimality of an application function and a discussion of its systematic improvement; and finally the proposal of an algorithm that operates conservatively based on well-defined properties that ensure its termination and help to find the reasons whenever no execution order of transformations that yields consistent models is found. It answers the following research question:

RQ 1.4: How can transformations in a network be orchestrated and which properties can such an orchestration strategy fulfill?

While existing approaches to orchestrate transformations are restricted to specific network topologies, our approach is supposed to not restrict the supported topology in any way. Existing work proposes, for example, to define an execution order explicitly [Pil+08; Van+07] or to derive a topological order [Ste20b], which restricts the supported topologies to those in which a transformation needs to be executed only once. We prove that it is not possible to orchestrate arbitrary transformations such that they always yield consistent models whenever that is possible, i.e., when an according execution order of the transformations exists. We do, however, provide an algorithm that is able to process transformation networks of arbitrary topology, which follows a specific orchestration strategy: it does not necessarily find an execution order that yields consistent models whenever it exists, but it is defined in way that it supports the transformation developer or user in finding the reason for the inability to find such an order. On the one hand, this gives transformation developers systematic knowledge about limitations regarding the possibility to orchestrate transformations and, on the other hand, gives them an algorithm for the orchestration to be readily applied.

Selected insights presented in this chapter have been developed in a scientific internship together with Joshua Gleitze, which was supervised by the author of this thesis, and have also been published [GKB21].

7.1. Orchestration Goals and Problem Statement

To recapitulate, an application function $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ for transformation networks, as defined in Definition 4.13, accepts models and changes to them and yields either a tuple of models or $\perp$. Whenever it returns a tuple of models, they must be the result of applying the transformations in $\mathbb{t}$ of the network in an order determined by the orchestration function $\textsc{Orc}_{\mathbb{t}}$. We then say that this execution order is an *orchestration* of the transformations and that the execution of transformations in that order *yields* those models. The notion of correctness for the application function given in Definition 4.14 additionally requires the returned models to be consistent. We did, however, not yet define when we expect the function to return consistent models and when we allow it to return $\perp$, as this requires a further discussion of the alternatives, which we provide in the following.

The application function highly depends on the results of the orchestration function. If that function does not deliver an orchestration that yields consistent models, a correct application function may only return $\perp$. Thus, we are particularly concerned with ensuring that the orchestration function finds an orchestration that yields consistent models as often as possible. We call an orchestration that yields consistent models a *consistent orchestration*. Precisely, we define an orchestration and a consistent orchestration as follows.

Definition 7.1 (Orchestration)

Let $\mathbb{t}$ be a transformation set. We call a sequence $[\mathfrak{t}_1, \mathfrak{t}_2, \dots] \in \mathbb{t}^{<\mathbb{N}} := \bigcup_{i=0}^{\infty} \mathbb{t}^i$ of these transformations an *orchestration* of them.

For models $\mathfrak{m} \in \mathfrak{M}$ and changes $\delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}}$, we say that an orchestration $[\mathfrak{t}_1, \dots, \mathfrak{t}_n]$ is *consistent* if, and only if, the subsequent application of the transformations to $\mathfrak{m}$ and $\delta_{\mathfrak{M}}$ is consistent, i.e.:

$$\exists\, \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \big(\textsc{Gen}_{\mathfrak{M},\mathfrak{t}_n} \circ \dots \circ \textsc{Gen}_{\mathfrak{M},\mathfrak{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}}) \\ \wedge\ \delta'_{\mathfrak{M}}(\mathfrak{m})\ \textit{consistent to}\ \mathbb{t}\big)$$

The definition of an orchestration function allows it to determine an arbitrarily long sequence of transformations, also including each transformation multiple times. We have introduced this general notion to avoid unnecessary restrictions. In the following, we show the necessity of having this unrestricted notion rather than allowing each transformation to be executed only once, as proposed in existing work [Ste20b]. From the insight that we need to allow transformations to be executed multiple times, we derive and discuss when we expect the application function to return consistent models to finally come up with a notion of *optimality* for the orchestration function determining the execution order. This leads to the definition of the central *orchestration problem* that we want a transformation network to solve.

7.1.1. Single Transformation Execution

The possible numbers of executions for transformations of a network range from a selected execution of a subset, e.g., in terms of an induced spanning tree, over the execution of each transformation for one or a fixed number of times, to an arbitrary number of executions per transformation. In the following, we demonstrate why a single execution of each transformation is not sufficient in practice and prove that it is not sufficient in general.

The even stronger restriction to spanning trees is obviously insufficient. Consider the following consistency relations. For simplicity reasons, we use model-level relations (Definition 4.1) instead of fine-grained ones:

$$
\begin{aligned}
CR_{12} &= \{\langle m_1, m_2\rangle, \langle m_1, m_2'\rangle, \langle m_1', m_2'\rangle, \langle m_1', m_2''\rangle\}\\
CR_{13} &= \{\langle m_1, m_3\rangle, \langle m_1, m_3''\rangle, \langle m_1', m_3\rangle, \langle m_1', m_3'\rangle\}\\
CR_{23} &= \{\langle m_2, m_3\rangle, \langle m_2', m_3'\rangle, \langle m_2', m_3''\rangle, \langle m_2'', m_3\rangle\}
\end{aligned}
$$

This set of relations $\{CR_{12}, CR_{13}, CR_{23}\}$ is compatible according to Definition 5.3, because for each model there is a containing tuple of models that is consistent. For the initial tuple of models $\langle m_1, m_2, m_3\rangle$, we consider a change that changes m_1 to m_1'. Then we can distinguish three possible spanning trees, each of two transformations that try to restore consistency. We denote the transformations as $\mathfrak{t}_{12}$, $\mathfrak{t}_{13}$, and $\mathfrak{t}_{23}$ for the according consistency relations. Each tree consists of two transformations:

$\mathfrak{t}_{12}, \mathfrak{t}_{13}$: $\mathfrak{t}_{12}$ may change m_2 to m_2'. $\mathfrak{t}_{13}$ does nothing, because m_1' and m_3 are already consistent to CR_{13}, but m_2' and m_3 are not consistent to CR_{23}.

$\mathsf{t}_{12}, \mathsf{t}_{23}$: Like before, t_{12} may change m_2 to m_2'. t_{23} may then change m_3 to m_3''. m_1' and m_3'' are, however, not consistent to CR_{13}.

$\mathsf{t}_{13}, \mathsf{t}_{23}$: t_{13} may do nothing, because m_1' and m_3 are already consistent to CR_{13}. t_{23} does also nothing, because m_2 and m_3 are still consistent to CR_{23}. m_1' and m_2 are, however, not consistent to CR_{12}.

Thus, we need to execute each transformation at least once, because each transformation is only responsible for restoring consistency to its consistency relations. We cannot expect the resulting models to be consistent if some transformations were not executed, although the involved models were changed by other transformations. However, restricting the execution to each transformation once is not appropriate either. To show that, we consider examples that we derived from those we have already presented in previous work [GKB21], which use a different scenario context.

Consider the example in Figure 7.1, which depicts the introductory one of Figure 1.4 more precisely. In the example, interfaces in the UML and Java are related to architectural interfaces in a PCM model. PCM components are realized by equally named classes in the UML and Java. Additionally, when a PCM component requires an interface, this is realized by a field of the interface type and an appropriate constructor parameter in the component realization class in the UML and Java. Consistency is defined by transformations between PCM and UML, as well as between UML and Java.

In the scenario in Figure 7.1, we begin with a consistent state of one interface and component, each realized by an interface and class, respectively, in both the UML and Java. A user then introduces a change of the Java code, in which he or she adds a field of the interface type to the component realization class. The transformation between UML and Java propagates this change to the UML model, such that both models are consistent again. The transformation between PCM and UML then detects that the added field is of the type of an architectural interface, thus representing a requires relation between the corresponding component and the architectural interface. It adds the appropriate requires relation to the PCM model but also adds an appropriate parameter to the constructor of the component realization class in the UML. This introduces a further inconsistency between the UML and the Java model, which requires the transformation between UML and Java to be executed again to also add that constructor parameter in the Java code.

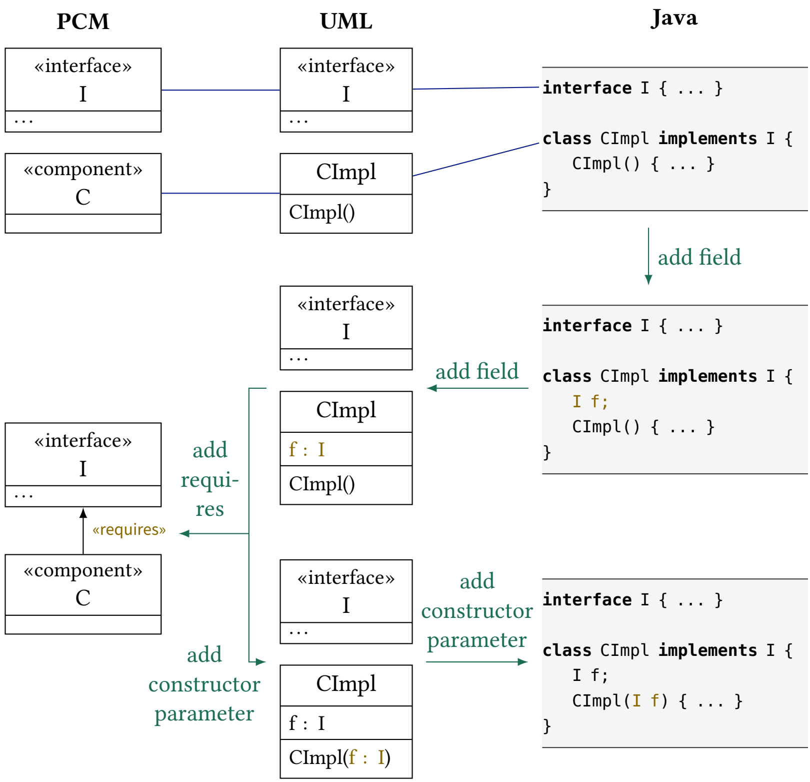


Figure 7.1.: Necessity of executing a transformation multiple times. For initially consistent models, the Java code is changed, requiring the UML and PCM models to be updated accordingly. Blue lines without arrowheads connect initially corresponding elements, and green lines with arrowheads indicate changes performed by a user or consistency preservation.

We have simplified the example to the necessary core, although in practice a further transformation between PCM and Java may be required, e.g., to ensure that the field is set within the constructor. One might argue that having such a cycle in the graph induced by the transformations between PCM, UML, and Java resolves the problem, as the second execution of the transformation between UML and Java is not necessary if the information is propagated from the PCM to Java. This is, however, only true if exactly this execution order is chosen and if the transformation between PCM and

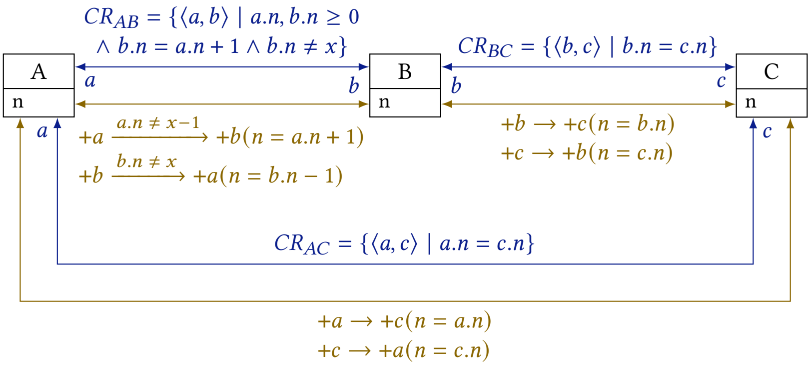


Figure 7.2.: Example of consistency relations and associated transformations with an arbitrary bound of necessary transformation executions depending on the value of x.

Java does not add further information to the Java model that must then be propagated to the UML.

In general, it is always possible that transformations need to react to the changes performed by other transformations if they are not in some way aligned with each other. This is because a synchronizing transformation may change both models. Thus, if one transformation restores consistency between two models and another transformation reacts to this by restoring consistency between one of these models and another one, then both these models become changed, which requires the first transformation to process the newly created changes again.

We can generalize the previous example to the one of Figure 7.2. It is an extension of the example given in Figure 6.5 for the necessity to execute the consistency preservation rules of a bidirectional transformation multiple times. This also applies to the case in which multiple synchronizing transformations are combined. The depicted relations and the sketched consistency preservation rules require that elements A, B, and C with the same value of n exist, and that for each A with value n, a B and C with n incremented by 1 exist except for the case that $n = x - 1$. Thus, for an A with $n = i$, all A, B, and C with $i \leq n < x$ must exist. This, obviously, requires the transformations to be executed $x - 1 - i$ times.

We prove the informally given statement with the following precise definition of the transformations for a fixed but arbitrary value of x. Let A, B, and C be the classes depicted in Figure 7.1.

$$I_{M_1} := \mathcal{P}(I_A),\; I_{M_2} := \mathcal{P}(I_B),\; I_{M_3} := \mathcal{P}(I_C)$$

$$\begin{aligned}
&CR_{12} := \{\langle a, b\rangle \in I_A \times I_B \mid a.n, b.n \geq 0 \wedge b.n = a.n + 1 \neq x\}\\
&\mathbb{CR}_{12} := \{CR_{12}, CR_{12}^T\}\\
&\textsc{Cpr}^{\rightarrow}_{\mathbb{CR}_{12}}(m_1, m_2, \delta_{M_1}) := \delta_{M_2}\\
&\quad \textit{with } \delta_{M_2}(m_2) := \{b \in I_B \mid \exists\, a \in \delta_{M_1}(m_1) : b.n = a.n + 1 \neq x\}\\
&\textsc{Cpr}^{\leftarrow}_{\mathbb{CR}_{12}}(m_2, m_1, \delta_{M_2}) := \delta_{M_1}\\
&\quad \textit{with } \delta_{M_1}(m_1) := \{a \in I_A \mid \exists\, b \in \delta_{M_2}(m_2) : b.n = a.n + 1 \neq x \wedge a \geq 0\}\\
&\mathfrak{t}_{12} := \langle \mathbb{CR}_{12}, \textsc{Cpr}^{\rightarrow}_{\mathbb{CR}_{12}}, \textsc{Cpr}^{\leftarrow}_{\mathbb{CR}_{12}}\rangle
\end{aligned}$$

$$\begin{aligned}
&CR_{13} := \{\langle a, c\rangle \in I_A \times I_C \mid a.n = c.n\},\; \mathbb{CR}_{13} := \{CR_{13}, CR_{13}^T\}\\
&\textsc{Cpr}^{\rightarrow}_{\mathbb{CR}_{13}}(m_1, m_3, \delta_{M_1}) := \delta_{M_3}\\
&\quad \textit{with } \delta_{M_3}(m_3) := \{c \in I_C \mid \exists\, a \in \delta_{M_1}(m_1) : a.n = c.n\}\\
&\textsc{Cpr}^{\leftarrow}_{\mathbb{CR}_{13}}(m_3, m_1, \delta_{M_3}) := \delta_{M_1}\\
&\quad \textit{with } \delta_{M_1}(m_1) := \{a \in I_A \mid \exists\, c \in \delta_{M_3}(m_3) : a.n = c.n\}\\
&\mathfrak{t}_{13} := \langle \mathbb{CR}_{13}, \textsc{Cpr}^{\rightarrow}_{\mathbb{CR}_{13}}, \textsc{Cpr}^{\leftarrow}_{\mathbb{CR}_{13}}\rangle
\end{aligned}$$

$$\begin{aligned}
&CR_{23} := \{\langle b, c\rangle \in I_B \times I_C \mid b.n = c.n\},\; \mathbb{CR}_{23} := \{CR_{23}, CR_{23}^T\}\\
&\textsc{Cpr}^{\rightarrow}_{\mathbb{CR}_{23}}, \textsc{Cpr}^{\leftarrow}_{\mathbb{CR}_{23}},\; \textit{and } \mathfrak{t}_{23} \textit{ accordingly}
\end{aligned}$$

$$\begin{aligned}
&\mathbb{CR} := \mathbb{CR}_{12} \cup \mathbb{CR}_{13} \cup \mathbb{CR}_{23}\\
&\mathbb{t}_{inc} := \{\mathfrak{t}_{12}, \mathfrak{t}_{13}, \mathfrak{t}_{23}\}
\end{aligned}$$

For these transformations, we can show that the transformation $\mathfrak{t}_{12}$ needs to be executed a minimal number of times depending on x for a specific input. Thus, it is not sufficient to execute each transformation only once in this network, and, even worse, we can enforce the necessity for an arbitrary high number of executions by proper selection of x.

Lemma 7.1 (Minimal Number of Transformation Executions)
Let $\mathbb{t}_{inc}$ be the previously defined transformation set, let $m_1 = m_2 = m_3 = \emptyset$ be empty models, and let $\delta_{M_1} \in \Delta_{M_1}$ be a change with $\delta_{M_1}(m_1) = \{a \in I_A \mid a.n = 0\}$. Then every orchestration function $\textsc{Orc}_{\mathbb{t}_{inc}}$ with $\textsc{App}_{\textsc{Orc}_{\mathbb{t}_{inc}}}(\langle m_1, m_2, m_3\rangle, \langle \delta_{M_1}, \delta_{id}, \delta_{id}\rangle)$ consistent to $\mathbb{CR}$ yields an orchestration that contains $\mathfrak{t}_{12}$ at least $x-1$ times.

Proof. $\textsc{App}_{\textsc{Orc}_{\mathbb{t}_{inc}}}$ can only return consistent models when it applies the transformations in the order delivered by $\textsc{Orc}_{\mathbb{t}_{inc}}$ by Definition 4.13. We thus consider every orchestration, as delivered by any orchestration function, to show that it contains $\mathfrak{t}_{12}$ at least $x-1$ times to deliver consistent models.

Let $max_n(m_1, m_2, m_3) \coloneqq max\{e.n \mid e \in m_1 \cup m_2 \cup m_3\}$ be the maximal value of n among all instances of A, B, and C in the given models m_1, m_2, and m_3. In the following, we shortly note max_n whenever the actual models are not relevant. We show three statements that together prove the lemma.

Executing $\mathfrak{t}_{13}$ and $\mathfrak{t}_{23}$ does not increase max_n: The transformations only ensure that for given models the returned models contain all elements with the same values of n and do not introduce new elements with values of n larger than the existing ones.

One execution of $\mathfrak{t}_{12}$ increases max_n by at most 1: There is no A or B with $n > max_n$. For every A with $n < max_n$, $\mathfrak{t}_{12}$ creates, if necessary, a B with value $n+1 \leq max_n$, thus not increasing max_n. For every B with $n \leq max_n$, it creates, if necessary, an A with value $n-1 < max_n$. For every A with $n = max_n$, a B with value $n+1 = max_n + 1$ is created, as long as $n \neq x-1$. For the newly created B, no further elements need to be created to fulfill the relations. Thus, max_n is, at most, increased by 1.

$max_n(m_1, m_2, m_3) < x-1 \Rightarrow \langle m_1, m_2, m_3\rangle$ *inconsistent to* $\mathbb{CR}$: There is at least one element with $n = max_n$ within the models. If the element with $n = max_n$ is an A, there must be a B with value $n+1$ due to $\mathbb{CR}_{12}$ and $n < x-1$. But due to $n = max_n$ such a B cannot exist, because otherwise $max_n = n+1$, so this is a contradiction. If the element with $n = max_n$ is a C, $\mathbb{CR}_{13}$ requires an A with the same value of n to exist and the same argument as before leads to a contradiction. Finally, if the element with $n = max_n$ is a B, then because of $\mathbb{CR}_{23}$, a C with the same value must exist and the same argument as before leads to a contradiction.

In summary, we have shown that models m_1, m_2, and m_3 are only consistent to $\mathbb{CR}$ when $max_n(m_1, m_2, m_3) \geq x-1$. Additionally, only $\mathfrak{t}_{12}$ increases max_n and with each execution it only increases it by at most 1. In consequence, starting with $max_n = 0$, we need at least $x-1$ executions of $\mathfrak{t}_{12}$ in an arbitrary sequence of the transformations in $\mathbb{t}_{inc}$ to achieve consistent models. $\square$

We have proven that transformation networks can require an arbitrary high number of executions of each transformation. By selecting an appropriate x in the example network, we can force the network to perform at least $x - 1$ executions of one transformation to yield a consistent tuple of models. With this insight, it directly follows that we cannot find an approach to define orchestration functions that deliver sequences containing each transformation only once if we want to ensure that the approach delivers a consistent orchestration of transformations if it exists.

Theorem 7.2 (Orchestration with Single Execution)
For a set of transformations $\mathbb{t}$, there can be models $\mathfrak{m}$ and changes δ to them for which each possible orchestration function $\textsc{Orc}_{\mathbb{t}}$ with whom $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta)$ is consistent delivers a sequence as $\textsc{Orc}_{\mathbb{t}}(\mathfrak{m}, \delta)$ that contains at least one transformation twice.

Proof. According to Lemma 7.1, $\mathbb{t}_{inc}$ requires at least two executions of $\mathfrak{t}_{12}$ for the inputs in Lemma 7.1 and $x \geq 3$. This proves the theorem by example. $\square$

Of course, for a specific set of transformations it may be possible that there is an orchestration for all possible models and changes to them leading to a consistent state and only requiring each transformation to be executed once. Theorem 7.2 shows, however, that this cannot be assumed in general. If we execute each transformation only once, we may exclude cases for which multiple executions of transformations would have led to a consistent tuple of models. The example we have given in Figure 7.1 is a simplification of a realistic transformation scenario, which we generalized to the previous network with transformations $\mathbb{t}_{inc}$. For that reason, the insight is likely to be relevant in realistic scenarios. We should not restrict orchestration to execute each transformation only once, as there can be realistic scenarios that require multiple executions to find consistent models. In the following, we thus allow an arbitrary number of executions of each transformation.

In addition, the examples, both the concrete one and the generalized abstract one, demonstrate that it can be necessary to modify the model that was originally changed by the user again. This contradicts the notion of *authoritative* models as, for example, introduced by Stevens [Ste20b]. With that notion, specific models are defined authoritative and cannot be changed, for example, because they are immutable or because they were changed by the user, and reverting those changes shall be avoided. While that behavior may be a desired, forbidding the modification of a whole model is not a proper solution as shown in the examples, which is why we do not consider a notion of authoritative models.

7.1.2. Orchestration Function Behavior

An application function is defined to return models only when they can be derived by applying transformations in an order delivered by the orchestration function and otherwise to return $\perp$. In addition, we expect a *correct* application function only to deliver consistent models. We have, however, not yet defined under which conditions we expect the function not to return $\perp$, because there are different reasons why the function may not be able to deliver consistent models, although we could expect it to do so. In fact, with the current definition, the function is even considered correct if it always returns $\perp$, which is obviously not practical. Thus, we need to define when exactly we expect the function to return $\perp$.

It might be intuitive to expect an application function to always return consistent models when the input models are consistent and when there is an execution order of the transformations, i.e., an orchestration, that delivers consistent models. This, in consequence, would lead to the requirement that the orchestration function delivers a sequence of transformations whose application delivers consistent models whenever such a sequence exists for the given models and changes to them. There can be the following reasons why the orchestration function may not deliver such a sequence.

Incompatible Relations: If the consistency relations are incompatible, a user change may introduce an element for which no consistent models exist. In consequence, the transformations cannot be executed in an order returning models that are consistent and still reflect the user change.

No Consistent Orchestration Exists: Even if relations are compatible, transformations may be defined in a way that they make contradictory decisions for locally consistent solutions. Thus, for a given change the consistency relations provide different ways of restoring consistency, of which each transformation selects one that is not consistent to one of the other relations. Then, no order of the transformations can restore consistency, although consistent models exist for the given change.

No Consistent Orchestration Found: Even if an order of transformations for given changes that delivers consistent models exists, the orchestration function may not deliver it.

These reasons can be considered to reside at different levels, because each of them induces the next. This means, if there is no orchestration, it cannot be found, and having contradictory relations, there exists no orchestration for some of the changes. In the end, all of them lead to the situation that no orchestration can be found and, thus, the orchestration function is not able to deliver it.

The intuitive requirement that the orchestration function delivers a consistent orchestration whenever it exists would ensure the third level and needs to assume fulfillment of the first two levels to avoid situations in which no consistent orchestration is found. While we can assume compatibility of the relations, for which we proposed an analysis in Chapter 5, we cannot assume that an orchestration does always exists, as we see in the following.

Although compatibility reduces the chance that an orchestration function does not deliver a consistent orchestration, as we have motivated with the scenario depicted in Figure 5.6, it does not ensure that there is always such a sequence of transformations that the orchestration function can find. In general, this is always the case when consistency relations define different options for consistency, i.e., if they allow the existence of different corresponding elements to consider the models consistent. Compatibility ensures that there is an overlap of these corresponding elements, such that for every element, for which consistency is restricted, consistent models can be found. If, however, the transformations always restore consistency by introducing corresponding elements that are not in this overlap, each transformation will restore consistency locally to its consistency relation, but they can, together, never restore consistency to all consistency relations.

Consider the situation that we have three metamodels A, B, and C with instances a_i, b_k, and c_l. Let us assume that these models are uniquely indexed by i, k, and l, and that we defined the following model-level consistency relations:

$$CR_{AB} = \{\langle a_i, b_k \rangle \mid k = i\}$$
$$CR_{AC} = \{\langle a_i, c_l \rangle \mid l = i \vee l = i + 1\}$$
$$CR_{BC} = \{\langle b_k, c_l \rangle \mid l = k + 1 \vee l = k + 2\}$$

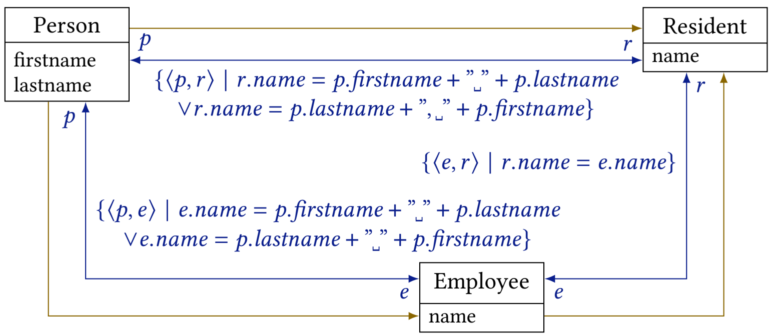
This induces the model tuples $\{\langle a_i, b_k, c_l \rangle \mid i = k = l - 1\}$, which are consistent to all three consistency relations. Thus, for any given model we are able to find instances of the other metamodels that are consistent to all consistency relations. If we define consistency preservation rules for these consistency relations, the ones for CR_{AC} and CR_{BC} may decide between two models to restore consistency, because their conditions define two options for consistent models. The set of consistent models, however, contains only those models fulfilling the first of these two conditions. If each consistency preservation rule selects the models that fulfill the second condition, the resulting models are locally consistent to its consistency relation, but they will never become globally consistent to all three relations. More precisely, if the consistency preservation rules for CR_{AC} select c_i for a_i and vice versa, and if the rules for CR_{BC} select c_{i+2} for a_i and vice versa, no orchestration of the transformations will yield consistent models, because they never select those models that are in the overlap of the consistency relations.

Figure 7.3 demonstrates this situation at a derivation of the running example. The consistency relation between employees and residents ensures that for each resident and employee there is a corresponding other element with the same *name*. The consistency relations between employees and persons, as well as between residents and persons ensure that for each person there is a corresponding employee and resident, respectively, but they allow different relations of their names. While both consider elements corresponding if the *name* of an employee and resident, respectively, are the concatenation of the *firstname* and *lastname* of a person, an employee is also allowed to have the inverse concatenation of *lastname* and *firstname*, whereas a resident is also allowed to have this inverse concatenation but with an additional separation of the *lastname* and *firstname* with a comma. These options for the consistency relations provide further degrees of freedom for each transformation on its own, as they allow, for example, employee names to

$$+p \longrightarrow r(name = p.lastname + \text{", "} + p.firstname)$$

$$\text{Alternative 1:} \quad +r \longrightarrow p(firstname = r.name.substringAfter(\text{" "}) \quad lastname = r.name.substringBeforeFirst(\text{","}, \text{" "}))$$

$$\text{Alternative 2:} \quad +r \longrightarrow \mathit{if}\ (r.name.contains(\text{","}))\ \mathit{then\ Alternative\ 1,\ else}\ p(firstname = r.name.substringBefore(\text{" "}) \quad lastname = r.name.substringAfter(\text{" "}))$$



$$+p \longrightarrow +e(name = p.lastname + \text{" "} + p.firstname)$$

$$+e \longrightarrow +p(firstname = e.name.substringAfter(\text{" "}) \quad lastname = e.name.substringBefore(\text{" "}))$$

$$+e \longrightarrow +r(name = e.name)$$

$$+r \longrightarrow +e(name = r.name)$$

---

with:

$$x.substringBefore(separator) \coloneqq x.substring(0, x.indexOf(separator))$$

$$x.substringBeforeFirst(separator1, separator2) \coloneqq x.substring(min(x.indexOf(separator1), x.indexOf(separator2)))$$

$$x.substringAfter(separator) \coloneqq x.substring(x.indexOf(separator) + 1)$$

Figure 7.3.: Consistency relations with options for corresponding elements leading to consistency preservation rules for which no consistent orchestration exists.

be encoded differently. This can be reasonable if the order of *firstname* and *lastname* is not relevant in a model managing employees. In combination with the other consistency relations, however, the only employees, residents, and persons that are considered consistent to all of the consistency relations are those having the same names with the concatenation of *firstname* and *lastname*. Nevertheless, these consistency relations are compatible, because for each possible condition element, i.e., for every possible employee, person, and resident, consistent models exist that contain them.

Consistency preservation rules for these consistency relations need to choose one of the given options for the names of corresponding employees, residents, and persons. Figure 7.3 sketches consistency preservation rules that make such a selection. The rules with alternative 1 ensure that for each employee, resident, and person corresponding elements exist, which fulfill those relations of the names that are conflicting. This means, the employee's *name* is the concatenation of the *lastname* and *firstname* of a person, whereas the resident's *name* contains an additional comma in that concatenation. In the other direction, the names of employees and residents are split at the appropriate indices given by the whitespace and comma, respectively, to calculate the required *firstname* and *lastname* of a person. In consequence, there is no execution sequence of the transformations that results in consistent models, because the execution of the transformation between employees and persons always leads to a violation of the consistency relation between residents and persons and vice versa. This is because the transformation between persons and residents always introduces a comma in the resident's *name*, which is then appended to the *lastname* by the transformation between employees and persons. A repeated execution of the transformation repeatedly appends that comma. On the other hand, the execution of any of the transformations does never lead to the introduction of a person that fulfills the non-conflicting conditions of both consistency relations by simply containing a *firstname* and *lastname* that is represented as a concatenation of *firstname* and *lastname* in both an employee and a resident. This is a concrete example for the abstract situation that of different options in consistency relations always the non-overlapping ones are chosen by the consistency preservation rules.

If we consider alternative 2 for the consistency preservation rule between persons and residents, we can always find a consistent orchestration. The alternative rule decides how consistency is ensured based on the existence of a comma within the resident's *name*. If a comma is present, the *name* relation containing a comma is used, and otherwise the simple concatenation of *firstname* and *lastname* is assumed. After adding an employee, first executing the transformation from employees to residents and afterwards the one from residents to persons ensures that all consistency relations are fulfilled, because the one between residents and persons sets the *firstname* and *lastname* of a person according to the relation that is also fulfilled between the person and the employee, because the *name* does not contain a comma. After adding a person, first executing the transformation from persons to employees and then the one from employees to residents results in an employee and a resident with inverse *firstname* and *lastname*. Since this resident is not consistent to the person, the transformation from residents to persons adds another person, which then also contains the swapped *firstname* and *lastname*. Executing the same process again results in two persons, residents, and employees with both assignments of *firstname* and *lastname*, which may not be intended but actually represents a consistent result. Finally, after adding a resident we can, for example, first apply the transformation between residents and employees and then the one between residents and persons, resulting in consistent models due to the same reasons as above.

Although consistent orchestrations of the transformations with the consistency preservation rule defined as alternative 2 exist, not every execution order leads to consistent models. In the scenarios discussed above, we have ensured that the transformation between residents and persons is executed after the addition of a resident. If this transformation is executed after the addition of a person, a comma is added, which leads to the subsequent application of the same consistency preservation rules as with alternative 1 and implies that no further orchestration yields consistent models.

No matter whether exactly these consistency relations and preservation rules for them may occur in an actual transformation network, they exemplify the general situation of having consistency preservation rules that select one of different options provided by the consistency relations to introduce corresponding elements to restore consistency. The example shows that whether or not a consistent orchestration of transformations exists in such a situation depends on whether at least one transformation selects an option that is consistent to other consistency relations as well. It also shows that even if a consistent orchestration exists, not all orchestrations yield consistent models. Thus, we need to be able to find one that does.

In accordance with existing work [Ste20b], we call a given tuple of models and changes *resolvable* by a transformation network if a consistent orchestration exists. We have to accept that transformation networks may be unresolvable, i.e., that there is no consistent orchestration of the transformations. Ensuring that a network is resolvable for every change would lead to restrictions for the individual transformations that would especially require different transformations to be aligned with each other. Since that conflicts our assumption of independent development and modular reuse, we accept unresolvability and instead focus on how we can find an orchestration if it exists.

In conclusion, we expect the application function to deliver consistent models whenever a consistent orchestration, i.e., an execution order that yields consistent models, exists. Thus, we want to ensure that the orchestration function is able to always find such an orchestration if it exists. We define this as an *optimality* property in the following.

7.1.3. Optimal Orchestration

To ensure that an application function delivers consistent models whenever a consistent orchestration exists, we need to find an orchestration function that fulfills this property. We denote this as an *optimal* orchestration function. Recall that $\textsc{Gen}_{\mathfrak{M},\mathfrak{t}}$ is the generalization function that applies a transformation to a model tuple that instantiates all metamodels in a tuple $\mathfrak{M}$.

Definition 7.2 (Optimal Orchestration Function)

Let $\mathbb{t}$ be a set of transformations for consistency relations $\mathbb{CR}$ and metamodels $\mathfrak{M}$. We say that an orchestration function $\textsc{Orc}_{\mathbb{t}}$ for these transformations is *optimal* if, and only if, it returns a consistent orchestration whenever it exists:

$$\begin{aligned}
&\forall\, \mathfrak{m} \in I_{\mathfrak{M}} \mid \mathfrak{m} \textit{ consistent to } \mathbb{CR} : \forall\, \delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \\
&\quad \Big[\exists\, \mathfrak{t}_1, \ldots, \mathfrak{t}_i \in \mathbb{t} : \exists\, \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \big(\delta'_{\mathfrak{M}}(\mathfrak{m}) \textit{ consistent to } \mathbb{CR} \\
&\qquad \wedge \textsc{Gen}_{\mathfrak{M},\mathfrak{t}_i} \circ \ldots \circ \textsc{Gen}_{\mathfrak{M},\mathfrak{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}})\big) \\
&\quad \Rightarrow \exists\, \mathfrak{t}'_1, \ldots, \mathfrak{t}'_k \in \mathbb{t} : \exists\, \delta''_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \big(\delta''_{\mathfrak{M}}(\mathfrak{m}) \textit{ consistent to } \mathbb{CR} \\
&\qquad \wedge \textsc{Gen}_{\mathfrak{M},\mathfrak{t}'_k} \circ \ldots \circ \textsc{Gen}_{\mathfrak{M},\mathfrak{t}'_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta''_{\mathfrak{M}}) \\
&\qquad \wedge \textsc{Orc}_{\mathbb{t}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = [\mathfrak{t}'_1, \ldots, \mathfrak{t}'_k]\big)\Big]
\end{aligned}$$

Note that we allow an optimal orchestration function to return a sequence even when there is no consistent orchestration. This is reasonable, because an application function may also support finding the reasons when no consistent orchestration is found by delivering a sequence of transformations that leads to a failure, as we discuss in Section 7.4.

Finally, the result of the application function is what is relevant in the process of consistency preservation. Thus, we apply the notion of *optimality* to that function accordingly by requiring it to deliver consistent models whenever a consistent orchestration exists.

Definition 7.3 (Optimal Application Function)

Let $\mathbb{t}$ be a set of transformations for consistency relations $\mathbb{CR}$ and metamodels $\mathfrak{M}$. We say that an application function $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ for these transformations is *optimal* if, and only if, it returns models that are consistent whenever there is a consistent orchestration of the transformations:

$$\begin{aligned}&\forall\, \mathfrak{m} \in I_{\mathfrak{M}} \mid \mathfrak{m} \textit{ consistent to } \mathbb{CR} : \forall\, \delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} :\\ &\quad \big[\, \exists\, \mathsf{t}_1, \dots, \mathsf{t}_i \in \mathbb{t} : \exists\, \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \big(\delta'_{\mathfrak{M}}(\mathfrak{m}) \textit{ consistent to } \mathbb{CR}\\ &\qquad \wedge \textsc{Gen}_{\mathfrak{M},\mathsf{t}_i} \circ \dots \circ \textsc{Gen}_{\mathfrak{M},\mathsf{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}})\big)\\ &\quad \Rightarrow \textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \textit{ consistent to } \mathbb{CR}\big]\end{aligned}$$

According to the defined behavior of an application function, an optimal application function requires an optimal orchestration function.

Lemma 7.3 (Application / Orchestration Function Optimality)

An application function $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ can only be optimal if $\textsc{Orc}_{\mathbb{t}}$ is optimal.

Proof. Let us assume that the condition in Definition 7.3 is fulfilled, i.e., that the input models are consistent and that a consistent orchestration of the transformations exists for them. Then, to be optimal, the application function needs to return models that are consistent. According to the definition of an application function (see Definition 4.13), the sequence of transformations delivered by $\textsc{Orc}_{\mathbb{t}}$ for that input must yield the same model tuple as $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$. Thus, the orchestration function must deliver a sequence for such inputs that yields consistent models, which is equivalent to $\textsc{Orc}_{\mathbb{t}}$ being optimal. □

7.1.4. The Orchestration Problem

The problem to find a consistent orchestration whenever it is exists, i.e., to find an optimal orchestration function, is the central subject of the following sections. This is what we denote as the *orchestration problem.* We prove that the problem is undecidable, discuss how we can make it decidable, and propose strategies to deal with its undecidability. Finally, we come up with a discussion of conservatively approximating a solution to the problem. We define the problem as follows.

Definition 7.4 (Orchestration Problem)
The problem to find a consistent orchestration of transformations for given inputs (models and changes to them) if it exists is called the *orchestration problem.*

Often, the more general problem of deciding whether a consistent orchestration exists is sufficient.

Definition 7.5 (Orchestration Existence Problem)
The question whether a consistent orchestration of transformations for given inputs (models and changes to them) exists is called the *orchestration existence problem.*

In fact, both these problems are equivalent in the sense that having a solution for one of them also delivers a solution for the other.

Theorem 7.4 (Orchestration / Existence Problem Equivalence)
The orchestration problem can be solved if, and only if, the orchestration existence problem can be solved.

Proof. If a solution for the orchestration problem exists, it directly induces a solution for the orchestration existence problem, because if we find a consistent orchestration whenever it exists, we also know whether it exists. If a solution for the orchestration existence problem exists and we know that a consistent orchestration exists, we can find it by systematically testing all orchestrations of growing size until a consistent orchestration is found, since models are of finite size. Since we know that such an orchestration exists, this test must terminate, even though it may take an impractically long time. □

Since the orchestration problem is derived from the goal of finding an optimal application function, it is obviously equivalent to find an optimal application function or to solve the orchestration (existence) problem.

> **Theorem 7.5** (Optimality / Orchestration Problem Equivalence)
> *An optimal application function* $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ *can be defined if, and only if, a solution for the orchestration (existence) problem exists.*

Proof. We give the proof for the orchestration existence problem, which is, according to Theorem 7.4, equivalent to the orchestration problem. An optimal $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ returns consistent models whenever there is a consistent orchestration. With such a function, we are able to decide whether such an orchestration exists or not.

$$\textsc{ExistsOrc}(\mathbb{t}, \mathfrak{m}, \delta_{\mathfrak{M}}) := \begin{cases} \textsc{true}, & \textit{if } \textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \textit{ consistent to } \mathbb{t} \\ \textsc{false}, & \textit{otherwise} \end{cases}$$

$\textsc{ExistsOrc}$ returns $\textsc{true}$ if, and only if, a consistent orchestration exists. Since $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ is optimal, it returns consistent models in exactly those cases in which a consistent orchestration that yields them exists.

If a solution for the orchestration existence problem exists, we know whether a consistent orchestration exists for an input. In that case, we can define $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ to apply an according orchestration, which can be found by exhaustively testing different orchestration as discussed in the proof for Theorem 7.4, and otherwise to return $\perp$. □

7.2. Limitations of Orchestration Decidability

We have introduced the orchestration problem as the problem to find a consistent orchestration if it exists. This is equivalent to the existence of an optimal orchestration function. We can distinguish two approaches to ensure that the orchestration function is optimal. Let P be the problem space, i.e., all possible orchestrations of given transformations, and let S_i be the solution space with those orders that yield consistent models for a specific input i of models and changes to them.

Strategy Definition: Define a strategy that explores the problem space P to find one of the sequences in the solution space S_i if $S_i \neq \emptyset$.

Transformation Restriction: Define a *well-behavedness* property for transformations that ensures that executing the transformations in any order often enough they yield consistent models if $S_i \neq \emptyset$. This means, for any given input i there is an $n \in \mathbb{N}$ such that $\forall\, s \in P : \big(|s| > n \Rightarrow s \in S_i\big)$.

In the latter case, the orchestration function only needs to return orders that are longer than a specific length to be optimal. This means, performing an iterative execution of the transformations leads to a consistent result. Since optimality is a property of an orchestration function with respect to a set of transformations, defining a *well-behavedness* property for transformations to ease finding an optimal orchestration function will potentially not concern a single transformation but the set of them. This can easily contradict our assumption of independent development and reuse, or lead to restrictions of transformations that are not practical anymore.

In the following, we first investigate the possibility to find an optimal orchestration function without restricting the transformations. We define a general algorithm that realizes an application function, as in practice the function will be realized in terms of an algorithm that dynamically selects the next transformation to execute. We then discuss its correctness and termination and relate it to the orchestration problem. After proving undecidability of the orchestration problem, we discuss the possibilities to restrict transformations such that the problem becomes decidable. Finally, we shortly discuss confluence as a considerable property of transformation networks.

7.2.1. An Algorithm for Application Functions

We have so far discussed the orchestration and application functions as mathematical functions. In practice, they will be implemented as algorithms. In Algorithm 7.1, we propose an algorithm that realizes an application function. It also encodes the orchestration function, because an algorithm for the orchestration function will not determine a complete sequence of transformations for given models and changes but dynamically select the transformation to be executed next. As soon as the orchestration function determines no further transformation for execution, the algorithm returns the resulting models if they are consistent and $\perp$ otherwise.

Algorithm 7.1 Application function implementation.

procedure Apply($\mathbb{t}, \mathfrak{m}, \delta_{\mathfrak{M}}$)
  if $\neg$CheckConsistency($\mathbb{t}, \mathfrak{m}$) **then**
    return $\perp$
  end if
  $\mathfrak{t}_{executed}[\,] \leftarrow [\,]$
  $\delta_{\mathfrak{M},generated}[\,] \leftarrow [\,]$
  $\mathfrak{t}_{next} \leftarrow \text{Orchestrate}_{\mathbb{t}}(\mathfrak{m}, \delta_{\mathfrak{M}}, \mathfrak{t}_{executed}[\,], \delta_{\mathfrak{M},generated}[\,])$
  while $\mathfrak{t}_{next} \neq \perp$ **do**
    $(\mathfrak{m}, \delta_{\mathfrak{M}}) \leftarrow \text{Gen}_{\mathfrak{M},\mathfrak{t}_{next}}(\mathfrak{m}, \delta_{\mathfrak{M}})$
    $\mathfrak{t}_{executed}[\,] \leftarrow \mathfrak{t}_{executed}[\,] + \mathfrak{t}_{next}$
    $\delta_{\mathfrak{M},generated}[\,] \leftarrow \delta_{\mathfrak{M},generated}[\,] + \delta_{\mathfrak{M}}$
    $\mathfrak{t}_{next} \leftarrow \text{Orchestrate}_{\mathbb{t}}(\mathfrak{m}, \delta_{\mathfrak{M}}, \mathfrak{t}_{executed}[\,], \delta_{\mathfrak{M},generated}[\,])$
  end while
  $\mathfrak{m}_{res} \leftarrow \delta_{\mathfrak{M}}(\mathfrak{m})$
  if $\neg$CheckConsistency($\mathbb{t}, \mathfrak{m}_{res}$) **then**
    return $\perp$
  end if
  return $\mathfrak{m}_{res}$
end procedure

An application function according to Definition 4.13 is parametrized by an orchestration function, which, in turn, is parametrized by the set of transformations $\mathbb{t}$ that it is supposed to be executed on. A transformation network according to Definition 4.15 is defined to consist of a set of transformations and an application function, which may suggest that both the application as well as the orchestration function can be defined specific for one network. Algorithm 7.1 reflects this by assuming an Orchestrate function that is specific for a set of transformations. It may, however, be implemented by a generic function that works independent from the actual transformations and, instead, accepts them as a parameter. We do, however, focus on a general algorithm and an Orchestrate function that can be applied to any set of transformations. In that case, the algorithm does not realize a single application function but actually a family of application functions for all possible transformation sets $\mathbb{t}$.

The dynamic selection of transformations is realized by an ORCHESTRATE function and stops as soon as no further transformations to apply are delivered. The latter may be the case because the models are already consistent or because no further transformations can be applied. It is essential that ORCHESTRATE only returns a transformation that can be applied to the models and current changes, because otherwise its application by the GEN function in Line 9 would fail. The complete logic of the orchestration function is combined with the application of the delivered sequence in Lines 7–13. Since, in practice, the selection of transformations has to be performed dynamically anyway, an implementation of the orchestration function always needs to apply the transformations. Thus, a separation of the orchestration function into a separate algorithm, which performs the same steps as in Lines 7–13, leads to a redundancy by applying the transformations both in the separate orchestration algorithm as well as in the application algorithm.

The ORCHESTRATE function receives the history of executed transformations and generated changes, because if the complete orchestration function was implemented in a separate method, it would also be able to use that information to determine a proper orchestration. Otherwise, its expressiveness would be restricted with respect to the definition of an orchestration function, because that function makes a global decision for all transformations to execute based on the original input, which is not available to the ORCHESTRATE function after its first execution anymore. In a practical implementation of that function, the history may not be considered or truncated, depending on the information necessary for the implemented orchestration strategy.

The ORCHESTRATE function may implement different strategies for selecting the next transformation, which we later discuss in more detail. One simple strategy would execute the same order of transformations iteratively, thus always executing the transformation that was not executed for the longest time. Another reasonable strategy would be to manage a queue of transformations. After executing one transformation, all transformations that are adjacent to the metamodels of the two models that were modified by the transformation are enqueued if they were not enqueued yet. This ensures that those transformations that can process changes that have just been produced by another transformation are executed next. Both these strategies are independent from the actual transformations and could thus be implemented in a function that can be used for any set of transformations $\mathbb{t}$. In Section 7.4, we discuss a specific orchestration strategy. Until then, the actual strategy is not important and any of the exemplified ones can be imagined.

Next to Orchestrate, the algorithm uses the external functions Gen and CheckConsistency. The Gen function is the generalization function, which simply applies the given transformation to the appropriate models of the given tuple, as defined in Definition 4.12. The CheckConsistency function checks whether the given models are consistent to the set of transformations, according to Definition 4.10. This function can be implemented in two ways. First, it may be implemented as an explicit check regarding the consistency relations of the transformations. If the transformations are defined by their consistency relations, from which a transformations language derives the consistency preservation rules, such as QVT-R, the models can be checked regarding the given relations. In case of QVT-R, the transformations can be executed in *checkonly mode* [QVT, Sec. 7.9]. Second, it may be implemented by (virtually) executing the consistency preservation rules and checking whether their execution performs changes. If the transformations are hippocratic according to Definition 4.9, i.e., if they do not perform changes when the models are already consistent, consistency can be checked this way. This is always necessary when the consistency relations are not explicitly given but are implicitly defined as the fixed points of the consistency preservation rules, such as for transformations defined in QVT-O. Due to their simplicity, we do not provide an explicit implementation of these two functions.

7.2.2. Correctness and Termination of the Algorithm

Algorithm 7.1 is constructed to implement an application function according to Definition 4.13. It is designed to be correct, i.e., to return models only when they are consistent. We show that the algorithm fulfills these properties in the following theorem.

Theorem 7.6 (Apply Algorithm Correctness)

The Apply function in Algorithm 7.1 fulfills the functional behavior of an application function as defined in Definition 4.13 and is correct according to Definition 4.14.

Proof. The Apply function fulfills the input and output requirements of an application function according to Definition 4.13. It returns a model tuple only in Line 18, which is achieved by applying the changes that the sequence of transformations as delivered by the orchestration function yields, which is realized as a repeated call of the Orchestrate function in Lines 7–13. Thus, Apply fulfills the definition of an application function.

Correctness of an application function according to Definition 4.14 requires the output models, if not returning $\perp$, to be consistent to the consistency relations of all transformations, as long as the input models were consistent. The algorithm returns models only in Line 18 and otherwise returns $\perp$ before. The returned models are always consistent to the consistency relations of all transformations, because Lines 15–17 ensure this. $\square$

In addition to being correct, the algorithm needs to terminate for every input. The only source of non-termination is the loop for orchestrating transformations, as there are no recursions and further loops. According to the definition, an orchestration function is defined to return a finite sequence of transformations, which would also result in a finite number of executions of the loop for orchestrating transformations. The implementation by a dynamic selection of the next transformation to execute can, however, lead to an infinite sequence of transformations. The Orchestrate function receives the list of previously executed transformations, as otherwise it would never be able to identify that, for example, always the same transformation sequence is executed and leads to the same changes, which means that the algorithm performs an infinite alternation. We do, however, need to ensure that the Orchestrate function returns $\perp$ after a finite number of calls.

If we assumed that we can achieve optimality for the orchestration function, we would have the guarantee that if a consistent orchestration exists, the function will find it. There is, however, no restriction to what the orchestration function may return when there is no consistent orchestration. Thus, we have the following two options to ensure termination.

1. We enable the orchestration function to identify whether a consistent orchestration exists.
2. We find an upper bound for the number of necessary transformation executions. Then, if a higher number of transformations was executed, we cannot expect the algorithm to find consistent models anymore and thus abort it.

As the simplest solution, an upper bound would restrict the number of necessary transformation executions. We do, however, prove in the following that there is no such upper bound. Afterwards, we show that identifying whether a consistent orchestration exists is not possible either. This leads to the insight that we cannot guarantee termination of the algorithm with an optimal orchestration function.

With the example in Figure 7.1, in which values are incremented by one during each execution of one specific transformation until a fixed but arbitrary value x is reached, we were able to show Lemma 7.1. It states that there can be transformation networks in which a transformation needs to be executed at least $x - 1$ times for a fixed but arbitrary x until consistent models are found. Thus, any consistent orchestration contains that transformation at least $x - 1$ times. While we have used that insight in Theorem 7.2 to show that executing each transformation only once is, in general, insufficient, we can also use it to show the more general statement that there is no maximal length for the orchestration of transformation networks, independent from the network size.

Theorem 7.7 (Shortest Consistent Orchestration Upper Bound)
For every $t_{size} \geq 3$ and every $n \geq 0$, there is a set of transformations $\mathbb{t}$ with $|\mathbb{t}| > t_{size}$ such that there are models $\mathfrak{m}$ and changes $\delta_{\mathfrak{M}}$ to them for which each possible orchestration function $\textsc{Orc}_{\mathbb{t}}$ with whom $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}})$ is consistent delivers a transformation sequence with $|\textsc{Orc}_{\mathbb{t}}(\mathfrak{m}, \delta_{\mathfrak{M}})| > n$.

Proof. We know from Lemma 7.1 that $\mathbb{t}_{inc}$ requires at least $x - 1$ executions of $\mathfrak{t}_{12}$ for the inputs defined in Lemma 7.1 and the fixed but arbitrary value x. Thus, with $x \geq n + 2$, we know that at least $x - 1 \geq n + 1$ executions of $\mathfrak{t}_{12}$ are necessary. Let $\mathfrak{m}$ and $\delta_{\mathfrak{M}}$ be the inputs defined in Lemma 7.1. Then, for every orchestration function $\textsc{Orc}_{\mathbb{t}_{inc}}$ that delivers a consistent orchestration for $\mathfrak{m}$ and $\delta_{\mathfrak{M}}$, we know that $|\textsc{Orc}_{\mathbb{t}_{inc}}(\mathfrak{m}, \delta_{\mathfrak{M}})| \geq x - 1 \geq n + 1 > n$. Since adding arbitrary transformations whose consistency preservation rules implement the identity function to a set of transformations does not alter the results of the network, we can construct a network of arbitrary size ≥ 3 with the same behavior out of $\mathbb{t}_{inc}$ by adding such transformations. This proves the theorem by example. □

In consequence, there is no fixed value and no value depending on the transformation network size that defines an upper bound for the necessary number of transformation executions to yield consistent models, i.e., there is

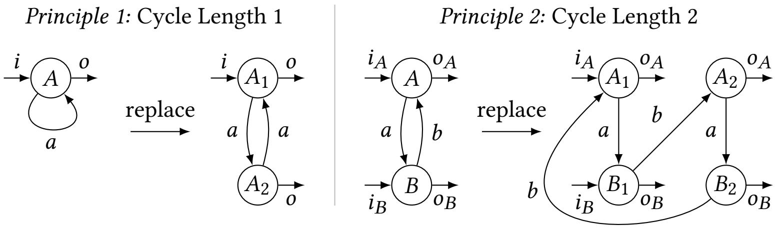


Figure 7.4.: Principles to eliminate cycles of length ≤ 2 in the transition function of a Turing machine. i and o are placeholders for all incoming and outgoing transitions of a state.

no upper bound for the shortest consistent orchestration. Thus, we cannot abort the execution of the Apply function after a fixed number of loop iterations without the possibility that consistent models would have been found if the execution had proceeded and thus not ensuring optimality.

7.2.3. Undecidability of the Orchestration Problem

To ensure termination of the Apply algorithm with an optimal orchestration function, we need to identify the case that no consistent orchestration exists, because that is the only situation in which otherwise an infinite number of transformation executions is possible. Unfortunately, we can show that this orchestration existence problem is undecidable. We reduce the halting problem for Turing machines to the orchestration problem. Thus, solving the orchestration problem would solve the halting problem. We have published a simplified version of this proof, based on a more concise formalism, in previous work [GKB21].

Given a Turing machine TM over some alphabet Σ, we construct metamodels $\mathfrak{M}_{\text{TM}}$ and a transformation network with a set of transformations $\mathbb{t}_{\text{TM}}$, as well as initial models $\mathfrak{m}_{\text{TM},x} \in I_{\mathfrak{M}_{\text{TM}}}$ and changes $\delta_{\mathfrak{M},\text{TM},x}$, for them for which a consistent orchestration exists if, and only if, TM halts on input $x \in \Sigma^*$. Without loss of generality, we assume that the graph of the transition function of TM contains no cycles of length ≤ 2. This means that it contains no self-loops, i.e., that the transition function always changes the state, and that there is no cycle between two states. This is without loss of generality, because cycles of these two lengths can be eliminated by replicating states. A self-loop can be eliminated by duplicating the state with a cycle of length 2 between the duplicated states, replicating all outgoing transitions for both states, and letting all incoming transitions go to one of these two states. Likewise, eliminating cycles of length 2 can be achieved by duplicating both involved states and replacing the cycle of length 2 by one of length 4, replicating all outgoing transitions for all states, and letting all incoming transitions go to one of the two states of each replicated one. Inductively applying these duplication principles can eliminate all cycles of length ≤ 2. The two principles are depicted in Figure 7.4.

We construct models that consist of a timestamp, the tape content, and the tape position. We encode this into a metamodel M_{TM}, whose instances represent exactly these contents. In a simplified notation, which considers a model as a tuple of these three elements rather than a set of elements, the instances of such a metamodel are given by $I_{M_{\text{TM}}} := \mathbb{N}_0 \times \Sigma^* \times \mathbb{N}_0$. A model $m := \langle \mathit{time}, \mathit{cont}, \mathit{pos} \rangle \in I_{M_{\text{TM}}}$ then represents a tuple of timestamp, tape content, and tape position. To represent the states of the Turing machine, we consider one such metamodel for each state of the Turing machine, although they are all equal. Thus, $\mathfrak{M}_{\text{TM}} := \langle M_{1,\text{TM}}, \ldots, M_{n,\text{TM}} \rangle$ with $n = |Q_{\text{TM}}|$ if we assume $Q_{\text{TM}} = \{q_1, \ldots, q_n\}$ to be the set of states of TM. We define the following function that returns the state of the Turing machine represented by a metamodel:

$$\mathrm{Q} : M_{i,\text{TM}} \mapsto q_i$$

We consider a transformation between each pair of metamodels whose represented states in the Turing machine have a transition between them. Finally, we consider models instantiating each of the metamodels to be kept consistent by an appropriate definition of these transformations representing the transitions of the Turing machine.

The transformations increment the timestamp, change the tape content, and update the tape position according to the transitions of TM if, and only if, the timestamp of one model is higher than the one of the other. More formally, let $\mathrm{Tr}(q_1, q_2) \subseteq \Sigma \times \{-1, 0, 1\} \times \Sigma$ be the transitions defined between the states $q_1 \in Q_{\text{TM}}$ and $q_2 \in Q_{\text{TM}}$, with -1, 0, and 1 indicating the head movements “left”, “stay”, and “right”, respectively. We define a consistency preservation rule for the transformation between the metamodels $M_{i,\text{TM}}$ and $M_{k,\text{TM}}$, which realizes the transition between the represented states of the Turing machine, as follows.

$$\begin{aligned}
&\textsc{Cpr}_{i,k}(m_i, m_k, \delta_{M_{i,\textsc{tm}}}, \delta_{M_{k,\textsc{tm}}}) := (\delta'_{M_{i,\textsc{tm}}}, \delta'_{M_{k,\textsc{tm}}}) \quad \textit{with:}\\
&m'_i := \langle time_{m'_i}, cont_{m'_i}[], pos_{m'_i}\rangle := \delta_{M_{i,\textsc{tm}}}(m_i)\\
&m'_k := \langle time_{m'_k}, cont_{m'_k}[], pos_{m'_k}\rangle := \delta_{M_{k,\textsc{tm}}}(m_k)\\
&\delta'_{M_{i,\textsc{tm}}}(m_i) := \begin{cases} \langle time_{m'_k}+1, cont_{m'_k}|_{pos_{m'_k}\leftarrow repl}, pos_{m'_k}+dir\rangle,\\ \quad \textit{if } time_{m'_k} > time_{m'_i} \wedge\\ \quad \exists\langle cont_{m'_k}[pos_{m'_k}], dir, repl\rangle \in \mathrm{Tr}(Q(M_{k,\textsc{tm}}), Q(M_{i,\textsc{tm}}))\\ m'_i, \quad \textit{otherwise}\end{cases}\\
&\delta'_{M_{k,\textsc{tm}}}(m_k) := \begin{cases} \langle time_{m'_i}+1, cont_{m'_i}|_{pos_{m'_i}\leftarrow repl}, pos_{m'_i}+dir\rangle,\\ \quad \textit{if } time_{m'_i} > time_{m'_k} \wedge\\ \quad \exists\langle cont_{m'_i}[pos_{m'_i}], dir, repl\rangle \in \mathrm{Tr}(Q(M_{i,\textsc{tm}}), Q(M_{k,\textsc{tm}}))\\ m'_k, \quad \textit{otherwise}\end{cases}
\end{aligned}$$

where $cont|_{pos\leftarrow repl} := cont[0 \ldots pos-1] \cdot repl \cdot cont[pos+1 \ldots |cont|-1]$.

The model-level consistency relations are implicitly given by the fixed points of the consistency preservation rules. For a consistency preservation rule $\textsc{Cpr}_{i,k}$, we define:

$$\begin{aligned}
CR_{i,k} := \{\langle m_i, m_k\rangle \in I_{M_{i,\textsc{tm}}} \times I_{M_{k,\textsc{tm}}} \mid \exists\, m'_i, m'_k, \delta_{M_{i,\textsc{tm}}}, \delta_{M_{k,\textsc{tm}}} :\\
\textsc{Cpr}_{i,k}(m'_i, m'_k, \delta_{M_{i,\textsc{tm}}}, \delta_{M_{k,\textsc{tm}}})(m'_i, m'_k) = \langle m_i, m_k\rangle\}
\end{aligned}$$

With this definition, each consistency preservation rule is correct, i.e., one application of it yields models that are consistent to its defined consistency relation. This is because due to the assumption that the graph induced by the transition function of TM does not contain cycles of length ≤ 2, there may be no cyclic transitions between the states which are represented by the models kept consistent by a single transformation.

We denote the set of all transformations realizing the transitions of TM as $\mathbb{t}_{\textsc{tm}}$, containing transformations $\mathfrak{t}_{i,k} = \langle CR_{i,k}, \textsc{Cpr}_{i,k}\rangle$ for all metamodel pairs $\langle M_{i,\textsc{tm}}, M_{k,\textsc{tm}}\rangle$ for which a transition between the represented states in $Q_{\textsc{tm}}$ exists, i.e., $\mathrm{Tr}(Q(M_{i,\textsc{tm}}), Q(M_{k,\textsc{tm}})) \neq \emptyset$.

Let $s \in Q_{\text{TM}}$ be the initial state of TM. We set

$$\begin{aligned}
&\mathfrak{m}_{\text{TM},x} := \langle m_{1,\text{TM},x}, \dots, m_{n,\text{TM},x} \rangle \\
&\quad \textit{with } m_{i,\text{TM},x} := \langle 0, \varepsilon, 0 \rangle \\
&\delta_{\mathfrak{M},\text{TM},x} := \langle \delta_{M_1,\text{TM},x}, \dots, \delta_{M_n,\text{TM},x} \rangle \\
&\quad \textit{with } \delta_{M_i,\text{TM},x}(m_i) := \begin{cases} \langle 1, x, 0 \rangle, & \textit{if } Q(M_i) = s \\ m_i, & \textit{otherwise} \end{cases}
\end{aligned}$$

We can show that for every Turing machine, this construction of a transformation network out of it solves the halting problem if we are able to solve the orchestration problem. First, we show an auxiliary lemma that proves that executing the transformations until all models are consistent terminates if, and only if, the according Turing machine halts.

Lemma 7.8 (Halting to Orchestration Problem Reduction)
Executing the transformations of $\mathbb{t}_{TM}$ for the models $\mathfrak{m}_{TM,x}$ and changes $\delta_{\mathfrak{M},TM,x}$ until all models are consistent terminates if, and only if, TM halts on input x. If executing the transformations terminates with the final changes $\delta_{\mathfrak{M},f}$, then the model in $\mathfrak{m}_f := \delta_{\mathfrak{M},f}(\mathfrak{m}_{TM,x})$ with the highest timestamp contains TM(x) as tape content.

Proof. Let $\delta_s, s \in \mathbb{N}_0$ be the tuple of changes created after executing s transformations and let $\mathfrak{m}_s = \langle m_{1,s}, \dots, m_{n,s} \rangle := \delta_s(\mathfrak{m}_{\text{TM},x})$ be the state of the models after applying that change. Then we can see the following per induction over the model states $\mathfrak{m}_s$:

1. There is at most one transformation $\mathfrak{t}_{i,k} \in \mathbb{t}_{\text{TM}}$ such that $\langle m_{i,s}, m_{k,s} \rangle$ is not consistent to $\mathfrak{t}_{i,k}$, i.e., $\langle m_{i,s}, m_{k,s} \rangle \notin CR_{i,k}$. This follows from the definition of TM and the last executed transformation. Let us, in contrary, assume that there was a second transformation that could be executed, because the models are inconsistent. We can distinguish whether the transformation involves any of $m_{i,s}, m_{k,s}$ or not. If that transformation involves any of these two models, then TM would have been non-deterministic, because each transformation realizes a transition between the associated states of TM. If that transformation involves none of these models, then one them must have been

changed before, because otherwise they are consistent by construction of $\mathfrak{m}_{\text{TM},x}$. Let that changed model be m'. The transformation to which m' and another model are inconsistent cannot be the one that was executed after m' was changed, because its correctness ensures that the two are consistent afterwards. Again, due to TM being deterministic, there cannot be another transformation that needed to be executed after m' was changed. Thus, another model must have been changed later, which led to the inconsistency. Then, however, the transformation would have needed to be applied, because the other model was changed. Since another transformation was executed and, again, because of TM being deterministic, that inconsistency cannot occur, thus being a contradiction to the assumption.

2. There is exactly one model $(\mathit{time}_{h,s}, \mathit{cont}_{h,s}, \mathit{pos}_{h,s}) \coloneqq m_{h,s} \in \mathfrak{m}_s$ that has the highest timestamp $\mathit{time}_{h,s}$ of all models in $\mathfrak{m}_s$. This follows from the previous insight that there is always at most one transformation to which the models are not consistent and which can thus perform changes, and that this transformation involves the just changed model, which, per induction, has the highest timestamp of all models. Thus, this model must be $m_{i,s}$ or $m_{k,s}$. We assume without loss of generality $m_{h,s} = m_{i,s}$.

3. If a $\mathfrak{t}_{i,k}$ exists to which $\langle m_{i,s}, m_{k,s}\rangle$ is not consistent, then $m_{k,s+1}$ contains the same tape content and the same tape position as would result if TM was executed one step from the state encoded in $m_{i,s}$ with tape content $\mathit{cont}_{i,s}$ and tape position $\mathit{pos}_{i,s}$. Additionally, $m_{k,s+1}$ is the model with the highest timestamp of all models in $\mathfrak{m}_{s+1}$.

4. $\mathfrak{m}_s$ is consistent to $\mathbb{t}_{\text{TM}}$ and thus no further transformation can produce changes if, and only if, TM would halt in state $m_{i,s}$ with tape content $\mathit{cont}_{i,s}$ and tape position $\mathit{pos}_{i,s}$. This is given by construction of the transformations, because a transformation can be executed if, and only if, the timestamp of the model is lower than the timestamp of a model to which a transformation is defined and if there is an according transition in Tr of TM. Since the timestamp of $m_{i,s}$ is higher than the timestamp of all other models, a transformation can be executed if, and only if, there is an according transition of TM, thus the execution of transformations terminates exactly when TM halts. □

With this lemma, it is easy to see that we could decide the halting problem if we can decide whether a consistent orchestration for the transformation network constructed from a Turing machine exists. In consequence, the orchestration problem is undecidable.

Theorem 7.9 (Orchestration Problem Undecidability)
The orchestration (existence) problem is undecidable.

Proof. We have given the constructive proof for Lemma 7.8 that any Turing machine can be simulated by a transformation network such that a repeated execution of transformations finds consistent models of which one contains the resulting tape content of the Turing machine if, and only if, the Turing machine halts. Thus, if we could decide the orchestration problem, we could decide whether a consistent orchestration exists. The consistent orchestration for the given transformations is unique, as in each step there is always only one transformation that can be executed. In consequence, knowing that a consistent orchestration exists means, according to Lemma 7.8, that we can decide whether TM halts, i.e., we could decide the halting problem. Due to equivalence of the orchestration problem and the orchestration existence problem, according to Theorem 7.4, this also applies to the orchestration existence problem. □

According to Theorem 7.5, we can only find an optimal application function if the orchestration problem is decidable. Thus, we know that we cannot find such a function.

Corollary 7.10 (Application Function Non-Optimality)
Let $\mathrm{App}_{\mathrm{Orc}_{\mathfrak{t}}}$ be an application function. Then $\mathrm{App}_{\mathrm{Orc}_{\mathfrak{t}}}$ cannot be optimal.

Proof. According to Theorem 7.5, an optimal application function can only be defined if a solution for the orchestration problem exists. Due to Theorem 7.9, we know that the problem is undecidable and thus an optimal application function cannot be defined. □

From this corollary, it also follows that we cannot implement the Apply function of the proposed algorithm in a way that it realizes an optimal application function and terminates for every possible input.

Corollary 7.11 (Apply Algorithm Non-Optimality)
Apply according to Algorithm 7.1 cannot terminate and return consistent models whenever an orchestration exists that yields them exists for every possible input.

Proof. If Apply always terminated and returned consistent models whenever there is an orchestration that yields them, it would implement an optimal application function. According to Corollary 7.10 an application function cannot be optimal. □

In consequence, we only have the two options to either restrict the expressiveness of the transformations such that they cannot be used to simulate a Turing machine anymore or to accept the situation that Apply may either not terminate in some cases or return $\perp$ although a consistent orchestration exists. We call this behavior *conservative*, because the algorithm never returns consistent models when there is no orchestration that yields them, but it may also not return consistent models in some cases in which actually an orchestration that yields them exists.

Finally, undecidability of the orchestration problem does not mean that this must be an essential problem for executing practical transformation networks. Most programming languages are Turing-complete and thus termination of programs written in them is generally undecidable due to the halting problem, but still they are used to develop functional and usable software. Thus, it is important to know that, in general, the expressiveness of transformation networks makes the orchestration problem undecidable, but this does not have to mean that we cannot practically apply these networks, as we will also see in the evaluation. We thus especially focus on how to deal with undecidability and approximate the problem conservatively.

In the following, we discuss options to restrict transformations to make the orchestration problem solvable and finally conclude that this is not an option for solving the discussed problem. Afterwards, we discuss how we can realize Apply in a way that it always terminates and produces reasonable outputs.

7.2.4. Restriction of Transformation Networks

We have discussed the necessity to restrict transformations as an input of the application function to avoid undecidability of whether a consistent orchestration exists. The following two kinds of restrictions can be distinguished.

Transformation: Restrictions only concern the single transformations. Thus, if each transformation fulfills a specific property, the application function is able to decide whether a consistent orchestration exists.

Network: Restrictions concern the complete network. Only the combination of transformations can fulfill an appropriate property that enables the application function to decide the orchestration problem but not each transformation on its own.

Since we assume transformations to be developed and reused independently, restrictions to single transformations are of special interest. It is, however, easy to see that it will unlikely be possible to define practical restrictions to single transformations that make the orchestration problem decidable. We show that even impractical restrictions do not make the problem decidable.

We have seen in the examples and the discussion in Subsection 7.1.2 that an essential reason for the non-existence of a consistent orchestration is the existence of different options within consistency relations. This means that a condition element is allowed to correspond to different condition elements to be considered consistent, like we have seen for the mapping of names in Figure 7.3. Different transformations can define different such options for specific elements, such that some of these options can never exist in globally consistent models, but only the ones that overlap between the consistency relations of all transformations can occur there. Compatibility of the consistency relations ensures that there is at least one such element in the overlap of the consistency relations, because if there was no consistent tuple of models containing the condition element, the relations would be considered incompatible. Unfortunately, each transformation can only select one of these options to restore consistency when a condition element is added, and if all transformations choose an element that is not in the overlap of the consistency relations, they will never find a consistent tuple of models.

In consequence, an obvious option to reduce expressiveness of transformations in order to make the orchestration problem decidable by ensuring that a consistent orchestration always exists would be an according restriction of consistency relations. Such a restriction would require that each condition element is only allowed to occur in a single consistency relation pair of a consistency relation. Thus, each condition element has a unique corresponding element to which it is considered consistent. Then, the consistency preservation rules cannot select between different options to restore consistency, and if the consistency relations are compatible, all of them relate elements in an equal way. Thus, the transformations find exactly those elements.

Although that approach will at least reduce the number of cases in which no consistent orchestration is found by our algorithm, there are still inputs for which no consistent orchestration exists. Since we do not restrict what transformations are allowed to do, they can perform arbitrary changes to restore consistency. This especially includes that they may always return changes that yield the same two models being consistent to that transformation but not to any models that can be delivered by the other transformations.

Let A, B, and C be three classes, each with an integer attribute n. We define three metamodels, each consisting of one of these classes, and consistency relations that require for each element in one model a corresponding one in another with the same value n. Additionally, we define consistency preservation rules, which deliver changes that yield the same models independent from the input. The resulting models are chosen to be consistent to the according consistency relation but not to any of the others.

$$I_{M_1} \coloneqq \mathcal{P}(I_A),\ I_{M_2} \coloneqq \mathcal{P}(I_B),\ I_{M_3} \coloneqq \mathcal{P}(I_C)$$

$$\begin{aligned}
&CR_{12} \coloneqq \{\langle a, b\rangle \in I_A \times I_B \mid a.n = b.n\},\ \mathbb{CR}_{12} \coloneqq \{CR_{12}, CR_{12}^T\}\\
&\textsc{Cpr}_{\mathbb{CR}_{12}}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) \coloneqq (\delta'_{M_1}, \delta'_{M_2})\\
&\quad \textit{with } \delta'_{M_1}(m_1) \coloneqq \{a \in I_A \mid a.n = 1\} \wedge \delta'_{M_2}(m_2) \coloneqq \{b \in I_B \mid b.n = 1\}
\end{aligned}$$

$$\begin{aligned}
&CR_{13} \coloneqq \{\langle a, c\rangle \in I_A \times I_C \mid a.n = c.n\},\ \mathbb{CR}_{13} \coloneqq \{CR_{13}, CR_{13}^T\}\\
&\textsc{Cpr}_{\mathbb{CR}_{13}}(m_1, m_3, \delta_{M_1}, \delta_{M_3}) \coloneqq (\delta'_{M_1}, \delta'_{M_3})\\
&\quad \textit{with } \delta'_{M_1}(m_1) \coloneqq \{a \in I_A \mid a.n = 2\} \wedge \delta'_{M_3}(m_3) \coloneqq \{c \in I_C \mid c.n = 2\}
\end{aligned}$$

$$\begin{aligned}
&CR_{23} \coloneqq \{\langle b, c\rangle \in I_B \times I_C \mid b.n = c.n\},\ \mathbb{CR}_{23} \coloneqq \{CR_{23}, CR_{23}^T\}\\
&\textsc{Cpr}_{\mathbb{CR}_{23}}(m_2, m_3, \delta_{M_2}, \delta_{M_3}) \coloneqq (\delta'_{M_2}, \delta'_{M_3})\\
&\quad \textit{with } \delta'_{M_2}(m_2) \coloneqq \{b \in I_B \mid b.n = 3\} \wedge \delta'_{M_3}(m_3) \coloneqq \{c \in I_C \mid c.n = 3\}
\end{aligned}$$

The example is a further simplification of our running example. Its consistency relations are compatible, as for each condition element, i.e., each instance of one of the classes, a consistent model tuple is given by that object together with instances of the other two classes having the same value of n. The consistency preservation rules are correct, as their result is consistent to the relation. Still, there is no consistent orchestration for any input that is not already consistent, because the consistency preservation rules always produce models that are inconsistent to the other consistency relations.

One might argue that the defined consistency preservation rules are highly unreasonable and will not occur in that way in practice. We may assume consistency preservation rules to preserve the input models and changes in some way instead of returning models that are completely unrelated to the input. We have, however, not defined an appropriate notion for that, because it is prone to be impractically restrictive. Some work on transformations [Che+17; MC16] proposes a notion of *least change* to ensure that transformations do not perform arbitrary unrelated changes, which could exclude those situations.

Although the given example is rather artificial and although there might be the additional property of least change that could further reduce the cases in which no consistent orchestration exists, the essential drawback is that these restrictions are not reasonable. Allowing a condition element to occur in multiple consistency relation pairs is essential, because options for corresponding elements are necessary, especially if there is a gap in the abstraction of two related metamodels. For example, a UML class needs to be able to correspond to all Java classes that provide different implementations of that class. Requiring exactly one Java class that is considered consistent to a UML class is obviously not applicable in practice. Thus, the restriction would make the consistency notion useless.

If we, instead, only require some notion of least change, like that only elements are changed which are involved in a violated consistency relation, this does also not solve the problem. In the example in Figure 7.3, relating the names of employees, residents, and persons, we have defined consistency preservation rules that only require changes to elements that actually violate consistency. Nevertheless, we have shown that for these consistency preservation rules only specific orchestrations are consistent and that with some modifications even no consistent orchestration exists.

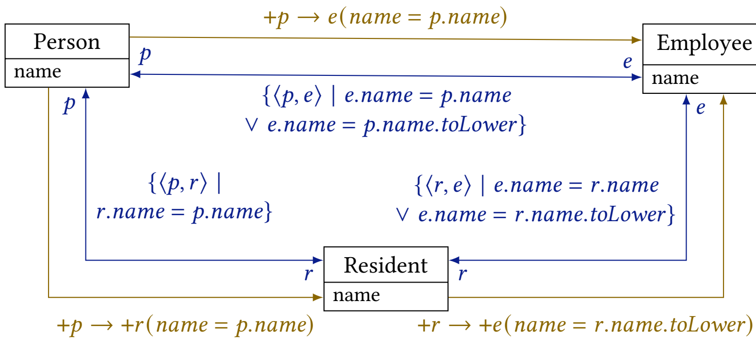


Figure 7.5.: Consistency relations and transformations representing a counterexample for the practicality of confluence in transformation networks.

In consequence, we found that even a well-defined restriction that is too strong to be applied in practice still cannot ensure that a consistent orchestration exists for every input, even though the examples at which we have shown that are rather artificial. Although this does not prove the impossibility to find a suitable restriction that solves the orchestration problem, which is even impossible because there is no unique notion of what an acceptable restriction would be, the investigated case shows that it is unlikely to find practical restrictions that solve the problem, because even impractical restrictions do not solve it.

7.2.5. Confluence in Transformation Networks

Confluence is an even stronger requirement than the existence of an optimal orchestration. In literature [Ste20b], confluence in a transformation network is described as the property that for given models and changes a consistent orchestration exists, and that two consistent orchestrations for the same input always yield the same models. Thus, executing transformations in any order such that the result is consistent will deliver the same result. It is, however, easy to see that this is an impractical requirement.

In the example depicted in Figure 7.5, derived from the running example, three consistency relations expect for each person, employee, and resident the two corresponding others to exist. They need to have the same *name* or, in case of the relations between persons and employees as well as between residents and employees, the employee may have the same *name* in lowercase. The consistency preservation rule between persons and employees ensures that an employee with the same *name* exists, whereas the one between residents and employees ensures that an employee with the *name* in lowercase exists. Whenever a person is added, two consistent orchestrations can be distinguished. First, the transformation between persons and employees can be executed, either followed or preceded by the one between persons and residents. Then, all elements have the same *name*. The models are also consistent to the relation between residents and employees, because the relation allows the names to be equal. Alternatively, the transformation between persons and residents can be executed, followed by the one between residents and employees. Then the employee has the *name* in lowercase, but still this is consistent to the relation between persons and employees.

Apart from that artificial example, such a situation can always occur if transformations have different options for elements to be consistent. If the overlap of consistent elements between all transformations does not contain a single element, the result may be any of the elements in the overlap. And the result may depend on which transformation made the first selection that fell into the overlap. This behavior is actually desired, thus preventing it by requiring confluence is not practical. Finally, Stevens [Ste20b, p. 14] also states that a network will only be confluent under very specific circumstances.

7.3. Conservatively Approaching the Orchestration Problem

In the preceding section, we have proven undecidability of the orchestration problem, and we have discussed that it is unlikely to find a practical restriction of the problem such that it becomes decidable. In consequence, we cannot achieve optimality of an orchestration and application function, which results in an algorithm that does not return optimal results and, depending on its implementation, may even not terminate. Since the algorithm cannot return optimal results anyway, termination can at least be achieved by introducing an artificial upper bound for the number of executed transformation. This potentially prevents the algorithm from finding consistent orchestrations in even more cases.

Based on those insights, we assume in this section that the orchestration problem cannot be restricted such that it becomes decidable. We accept that any application function and any algorithm that realizes it will only realize a conservative approximation of the orchestration problem. This means that it may only return consistent models delivered by a consistent orchestration, but it may not find a consistent orchestration although it exists. Considering consistent orchestrations as *positives*, we say that the function or algorithm, respectively, may deliver *false negatives* but no *false positives* and call it *conservative*. We investigate how we can define optimality of the application function in a gradual rather than a binary way, which is supposed to indicate how likely it is that it finds a consistent orchestration. We then follow the goal of finding means to systematically improve optimality. Since there are always cases in which the algorithm does not find a consistent orchestration, we propose an algorithm that is supposed to help identifying the reasons for failing in such cases in the subsequent section.

7.3.1. Systematic Improvement of Optimality

Although no optimal application function can be achieved, we can at least define a gradual notion of optimality. It indicates for how many input models and changes the application function returns consistent models compared to the number of cases in which a consistent orchestration exists at all. This can be seen as a fitness function for optimality $\textsc{Opt}$ of an application function:

$$\begin{aligned}&\textsc{Opt}(\textsc{App}_{\textsc{Orc}_{\mathbb{t}}})\\&\quad := \frac{|\{\langle \mathfrak{m}, \delta_{\mathfrak{M}}\rangle \mid \textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \textit{ is consistent to } \mathbb{t}\}|}{|\{\langle \mathfrak{m}, \delta_{\mathfrak{M}}\rangle \mid \textit{consistent orchestration of } \mathbb{t} \textit{ exists for } \langle \mathfrak{m}, \delta_{\mathfrak{M}}\rangle\}|}\end{aligned}$$

In fact, the numerator and denominator will usually both have infinite values, as there is an infinite number of possible models and changes to them. It does, however, not matter for us what the actual optimality value of an application function is. The purpose of the formula is only to explicitly state the influencing factors of optimality to discuss its systematic improvement.

Obviously, we may only improve the numerator to improve optimality, because the denominator, i.e., the number of cases in which consistent orchestrations exist, depends only on the transformations and not the application function. How to improve the numerator highly depends on the actually implemented application and orchestration functions. For the most general case, let us assume that we have an application function $\textsc{App}_{\textsc{Orc}_{\mathbb{t}}}$ whose orchestration function randomly determines any orchestration, i.e., it selects one of all possible orchestrations according to an equal distribution. So we consider the following event $E_{\mathfrak{m},\delta_{\mathfrak{M}}}$:

$$E_{\mathfrak{m},\delta_{\mathfrak{M}}} : \textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \textit{ is consistent to } \mathbb{t}$$

The probability that this event occurs is given by the ratio between the number of consistent orchestrations for that input and the number of all orchestrations:

$$P(E_{\mathfrak{m},\delta_{\mathfrak{M}}}) = \frac{|\{\mathfrak{t}[] \in \mathbb{t}^{<\mathbb{N}} \mid \mathfrak{t}[] \textit{ is consistent orchestration for } \langle \mathfrak{m}, \delta_{\mathfrak{M}} \rangle\}|}{|\mathbb{t}^{<\mathbb{N}}|}$$

Here, the denominator is the size of what we introduced as the problem space $P = \mathbb{t}^{<\mathbb{N}}$ containing all possible orchestrations, and the numerator is the size of what we introduced as the solution space $S_{\mathfrak{m},\delta_{\mathfrak{M}}} = \{\mathfrak{t}[] \in \mathbb{t}^{<\mathbb{N}} \mid \mathfrak{t}[] \textit{ is consistent orchestration for } \langle \mathfrak{m}, \delta_{\mathfrak{M}} \rangle\}$, which contains all consistent orchestrations for an input of models and changes.

We can introduce a stochastic variable $\mathit{AppCons}_{\mathfrak{m},\delta_{\mathfrak{M}}}$, which assigns the values 0 and 1 to the events $E_{\mathfrak{m},\delta_{\mathfrak{M}}}$ and its complementary:

$$\mathit{AppCons}_{\mathfrak{m},\delta_{\mathfrak{M}}}(\omega) := \begin{cases} 0, & \textit{if } \omega = \textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \textit{ is not consistent to } \mathbb{t} \\ 1, & \textit{if } \omega = \textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \textit{ is consistent to } \mathbb{t} \end{cases}$$

Its expected value is equal to the probability of event $E_{\mathfrak{m},\delta_{\mathfrak{M}}}$ to occur:

$$\mu(\mathit{AppCons}_{\mathfrak{m},\delta_{\mathfrak{M}}}) = P(\mathit{AppCons}_{\mathfrak{m},\delta_{\mathfrak{M}}} = 1) = P(E_{\mathfrak{m},\delta_{\mathfrak{M}}})$$

For an application function that chooses a random orchestration, we can thus express the numerator of $\textsc{Opt}(\textsc{App}_{\textsc{Orc}_{\mathbb{t}}})$ as the sum of expected values of the stochastic variables for all possible inputs.

$$|\{\langle \mathfrak{m}, \delta_{\mathfrak{M}} \rangle \mid \textsc{App}_{\textsc{Orc}_{\mathbb{t}}}(\mathfrak{m}, \delta_{\mathfrak{M}}) \textit{ is consistent to } \mathbb{t}\}| = \sum_{\mathfrak{m},\delta_{\mathfrak{M}}} \mu(\mathit{AppCons}_{\mathfrak{m},\delta_{\mathfrak{M}}})$$

$$= \frac{\sum_{\mathfrak{m},\delta_{\mathfrak{M}}} |\{\mathfrak{t}[] \in \mathbb{t}^{<\mathbb{N}} \mid \mathfrak{t}[] \textit{ is consistent orchestration for } \langle \mathfrak{m}, \delta_{\mathfrak{M}} \rangle\}|}{|\mathbb{t}^{<\mathbb{N}}|}$$

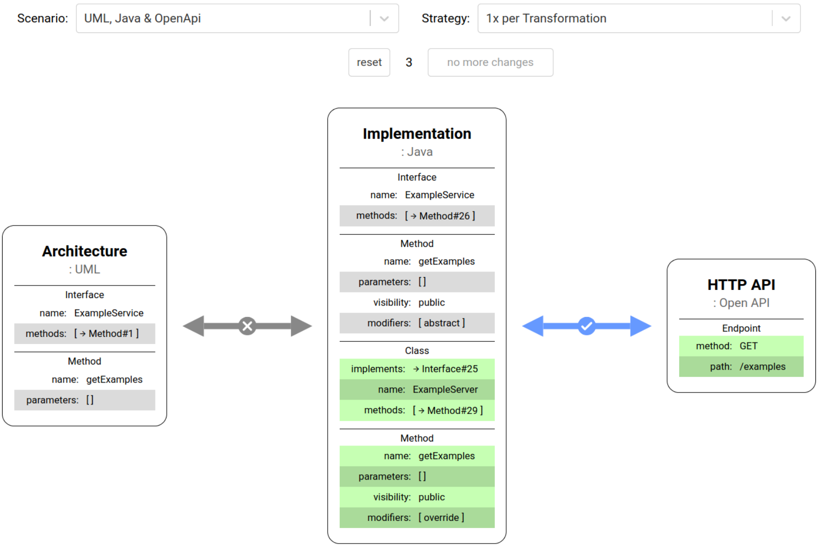


Figure 7.6.: Screenshot of a network for an architecture specification, its implementation, and an API specification in OpenAPI [OAPI], which requires multiple execution of the same transformation, developed in the transformation network simulator [GitSim].

Thus, if we can increase $P(E_{\mathfrak{m},\delta_{\mathfrak{M}}})$, we also improve optimality, even if orchestrations are chosen randomly. We can increase this probability by either improving the number of consistent orchestrations or by reducing the number of possibly considered orchestrations. The number of consistent orchestrations can only be influenced by requirements to the transformations. For example, the requirement of consistency relations to be compatible improves these values, as we have shown by example in Chapter 5. In the following, we discuss how we can reduce the number of possibly considered orchestrations while not reducing the number of consistent orchestrations, thus improving the probability of the application function to find a consistent orchestration and thus improving optimality.

The application function can, of course, contain more intelligent logic to determine an orchestration beyond random selection to improve the number of cases in which it finds a consistent orchestration. Implementing further mechanisms to make a reasonable selection may further improve the possibility to find a consistent orchestration. We investigated different orchestration strategies, such as the depth-first or breadth-first selection of transformations in the induced graph, and analyzed them with a simulator, which we developed for that purpose and which is available at GitHub [GitSim]. An example of the simulator for a scenario showing the necessity to execute a transformation more than once is depicted in Figure 7.6. For each strategy, however, we found categories of transformation networks for which it performed worse than some other strategy.

Another strategy could be to try different orchestrations as soon as it turns out that one orchestration cannot yield consistent models. This can, for example, be achieved by performing backtracking. Algorithm 7.1 dynamically selects transformations to execute. Thus, as soon as the algorithm detects that no further transformation executions can lead to a consistent orchestration, it can revert the last transformation execution and proceed with another transformation. This means that it resets the state of generated changes and executed transformations to the one before the current execution of the orchestration loop and proceeds again with another transformation. If all transformations as continuations of one sequence of executed transformations have been tried out, the algorithm recursively steps back the iterations of the loop. While this approach, in theory, allows us to explore the complete problem space $P = \mathbb{t}^{<\mathbb{N}}$, it is impractical, because the problem space is infinitely large. It may, however, be used to try different options in a subset of the problem space, such as those with a limited length.

Since we did not find a strategy that is, in general, superior to other investigated strategies, we did not proceed in that direction. This does not imply that such a strategy cannot be found, but we instead focused on finding orchestrations that should be generally avoided. To this end, we consider alternation as a possibility to reduce the number of cases in which non-termination can occur. Thus, it can improve optimality by both its dynamic detection and its avoidance.

7.3.2. Dynamic Detection of Alternation

The proposed Algorithm 7.1, like any algorithm, is supposed to *terminate* in a specific *state* to be considered correct. In our case, such a correct state, as required by an application function it implements, is the return of consistent models or $\perp$, which the algorithm fulfills by construction. In particular, the algorithm does never return models that are inconsistent, neither because it does not detect that they are inconsistent nor because it detects that they are inconsistent but still returns them. From our previous findings regarding decidability, we know that we cannot expect the algorithm to realize an optimal application function. Thus, we either need to implement ORCHESTRATE such that it always returns $\perp$ after a finite number of executions to ensure termination, which results in returning $\perp$ although an order of transformations that yields consistent models exists, or we allow an arbitrary number of executions to improve the ability to find consistent results but accept that the algorithm may not terminate.

We have discussed that non-termination of the algorithm can occur because no consistent orchestration exists at all or because the algorithm is not able to find it. A special case of non-termination is *alternation*, which means that the same states are passed repeatedly. In case of transformation networks, alternation means that from some point in time the subsequent executions of the transformations in Line 9 of Algorithm 7.1 repeatedly produce the same sequence of results, i.e., of changes. In contrast to non-termination in general, the scenario of alternation can at least be avoided by construction.

Definition 7.6 (Alternation of Apply Algorithm)

During an execution of Algorithm 7.1, let there be a number n of executions of the transformation execution loop in Lines 7–13 of Algorithm 7.1, such that for all numbers of loop executions $> n$ there is a sequence of executed transformations and generated changes that occur repeatedly at the end of the current states of $\mathfrak{t}_{executed}[]$ and $\delta_{\mathfrak{M},generated}[]$ at least two times. Then we call the execution of the algorithm *alternating*. If the execution of the algorithm does not terminate and is not alternating, we call it *diverging*.

The ORCHESTRATE function receives the history of transformations and already generated changes and is thus able to identify the situation that the same sequence of transformations was already executed and produced equal changes with each application. This allows to implement the function in a way that it does not return the same sequence of transformations when it was already passed and produced the same changes, e.g., by performing backtracking if such a situation is detected. If a concrete realization of the ORCHESTRATE function is not implemented in a way that it can react to the detection of alternation and produce a different sequence of transformations, it can at least return $\perp$ to ensure termination of Apply, because repeated execution of the same transformations will still return the same changes if Orchestrate behaves deterministically.

Alternation produces orchestrations that never yield consistent models. In consequence, they are, of course, contained in the problem space $P = \mathbb{t}^{<\mathbb{N}}$ consisting of all possible orchestrations, but they are never contained in the solution space $S_{\mathfrak{m},\delta_{\mathfrak{M}}} = |\{\mathfrak{t}[] \in \mathbb{t}^{<\mathbb{N}} \mid \mathfrak{t}[] \textit{ is consistent orchestration for } \langle \mathfrak{m}, \delta_{\mathfrak{M}} \rangle\}|$ containing the consistent orchestrations, independent from the actual models and changes. Avoiding alternation thus reduces the size of the problem space without the possibility of affecting the solution space as well and thus improves the possibility to find a consistent orchestration, as shown in the previous subsection.

7.3.3. Monotony for Avoiding Alternation

We have discussed that alternation of Algorithm 7.1, as a specific kind of non-termination scenario, can be avoided by construction of the Orchestrate function or can at least be detected by the Apply function. Instead of detecting alternation during orchestration and thus execution of transformations, we may also restrict the transformation network such that no alternation can occur by construction. We can achieve this by defining a notion of monotony for the transformations.

For the construction of synchronizing bidirectional transformations by unidirectional consistency preservation rules in Subsection 6.3.2, we have defined the property of *partial consistency improvement*, which is a monotony notion for the two unidirectional consistency preservation rules of a synchronizing bidirectional transformation, as each execution of them improves that property. We can, however, not define monotony in a similar way for the whole transformation network for two reasons. First, the notion of partial consistency is not applicable to transformation networks, because each transformation needs to completely restore consistency between two models. Second, since each transformation is developed independently from all others, we cannot apply the notion of partial consistent improvement to the other models by restricting how far a transformation may violate consistency to the other transformations. We thus define the following, different notion of monotony for transformations.

Definition 7.7 (Monotone Synchronizing Transformation)

Let $\mathfrak{M}$ be metamodels, and let $\mathfrak{t}$ be a synchronizing transformation. We call $\mathfrak{t}$ monotone if, and only if, it does not change elements that were already changed:

$$\forall\, \mathfrak{m} = \langle m_1, \ldots, m_n \rangle \in \mathfrak{M}, \delta_{\mathfrak{M}} = \langle \delta_{M_1}, \ldots, \delta_{M_n} \rangle \in \Delta_{\mathfrak{M}} :$$
$$\forall\, \delta'_{\mathfrak{M}} = \langle \delta'_{M_1}, \ldots, \delta'_{M_n} \rangle \in \Delta_{\mathfrak{M}} : \big[\textsc{Gen}_{\mathfrak{M},\mathfrak{t}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}})$$
$$\Rightarrow \forall\, i \in \{1, \ldots, n\} : \big((\delta_{M_i}(m_i) \setminus m_i) \subseteq \delta'_{M_i}(m_i)$$
$$\wedge\ (m_i \setminus \delta_{M_i}(m_i)) \cap \delta'_{M_i}(m_i) = \emptyset\big)\big]$$

The definition is based on the idea that transformations are only supposed to append changes but not to revert previous changes. This means that elements that were introduced by previous changes still need to be present after applying the transformation. Additionally, elements that were removed are not allowed to be added by the transformation again. Thus, all elements of the originally changed models were either contained in the original models or are contained in the models yielded by the transformation execution.

Having only monotone transformations ensures that the application of each orchestration that does not apply a transformation to already consistent models yields a sequence of pairwise different model states.

Lemma 7.12 (Monotone Transformation Orchestration Prefixes)

Let $\mathbb{t}$ be a set of correct, monotone synchronizing transformations for metamodels $\mathfrak{M}$. Then for all models and changes as well as any orchestration $[\mathfrak{t}_1, \ldots, \mathfrak{t}_m] \in \mathbb{t}^{<\mathbb{N}}$ that does not contain a transformation to be executed when its models are already consistent while other models are not, the prefixes of that orchestration yield the same models only if these prefixes are consistent orchestrations:

$$\forall\, \mathfrak{m} \in I_{\mathfrak{M}}, \delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \forall\, i, k \in \{1, \ldots, m\}, i \neq k :$$
$$\big[\textit{Gen}_{\mathfrak{M},\mathfrak{t}_i} \circ \ldots \circ \textit{Gen}_{\mathfrak{M},\mathfrak{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = \textit{Gen}_{\mathfrak{M},\mathfrak{t}_k} \circ \ldots \circ \textit{Gen}_{\mathfrak{M},\mathfrak{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}})$$
$$\Rightarrow \exists\, \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \big(\textit{Gen}_{\mathfrak{M},\mathfrak{t}_k} \circ \ldots \circ \textit{Gen}_{\mathfrak{M},\mathfrak{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}})$$
$$\wedge\ \delta'_{\mathfrak{M}}(\mathfrak{m})\ \textit{consistent to}\ \mathbb{t}\big)\big]$$

Proof. Assume that there are two prefixes $[\mathsf{t}_1, \ldots, \mathsf{t}_i]$ and $[\mathsf{t}_1, \ldots, \mathsf{t}_k]$ of an orchestration, $i < k$ without loss of generality, such that they yield the same inconsistent models, i.e., $\textsc{Gen}_{\mathfrak{M},\mathsf{t}_i} \circ \ldots \circ \textsc{Gen}_{\mathfrak{M},\mathsf{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = \textsc{Gen}_{\mathfrak{M},\mathsf{t}_k} \circ \ldots \circ \textsc{Gen}_{\mathfrak{M},\mathsf{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}})$ although $\textsc{Gen}_{\mathfrak{M},\mathsf{t}_k} \circ \ldots \circ \textsc{Gen}_{\mathfrak{M},\mathsf{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}})$ is not consistent to $\mathbb{t}$. We denote the change tuple delivered by any prefixes of length h as $\delta_{\mathfrak{M},h} = \langle \delta_{M_1,h}, \ldots, \delta_{M_n,h} \rangle$ with $(\mathfrak{m}, \delta_{\mathfrak{M},h}) = \textsc{Gen}_{\mathfrak{M},\mathsf{t}_h} \circ \ldots \circ \textsc{Gen}_{\mathfrak{M},\mathsf{t}_1}(\mathfrak{m}, \delta_{\mathfrak{M}})$. We know that the sequence of changes between the two prefixes does not perform any changes, i.e., $\delta_{\mathfrak{M},i}(\mathfrak{m}) = \delta_{\mathfrak{M},k}(\mathfrak{m})$. We also know that all the transformations between the prefixes, i.e., all transformations t_h with $i < h \leq k$, perform changes, i.e., $\textsc{Gen}_{\mathfrak{M},\mathsf{t}_h}(\mathfrak{m}, \delta_{\mathfrak{M},h-1}) \neq (\mathfrak{m}, \delta_{\mathfrak{M},h-1})$. Otherwise, the models affected by the transformation would either have been consistent before, which conflicts with the assumption that the orchestration does not contain a transformation when its models are already consistent while other models are not, or they would not be consistent afterwards, which conflicts with the assumed correctness of the transformations.

Thus, each transformation t_h $(i < h \leq k)$ performs modifications to the change tuple, i.e., adds or removes further elements. This especially applies to t_{i+1}. Let us assume that t_{i+1} adds an element. Then there is a model that contains the element after applying the change generated by the transformation, i.e., $\exists\, s \in \{1, \ldots, n\} : \exists\, e : e \in \delta_{M_s,i+1}(m_s) \setminus \delta_{M_s,i}(m_s)$. Due to the transformations being monotone, we know that this element was not contained before, especially not in m_s, as otherwise $e \in m_s \setminus \delta_{M_s,i}(m_s)$ and thus $(m_s \setminus \delta_{M_s,i}(m_s)) \cap \delta_{M_s,i+1}(m_s) \neq \emptyset$, which conflicts the definition of monotone transformations for t_{i+1}. Since $\delta_{M_s,k}(m_s) = \delta_{M_s,i}(m_s)$, we know that $e \notin \delta_{M_s,k}(m_s)$. Thus, there must be a transformation t_h with $i+1 < h \leq k$ which, in turn, removes this element, i.e., $e \in \delta_{M_s,h-1}(m_s) \setminus \delta_{M_s,h}(m_s)$. Then $e \in \delta_{M_s,h-1}(m_s) \setminus m_s$ and thus $\delta_{M_s,h-1}(m_s) \setminus m_s \not\subseteq \delta_{M_s,h}(m_s)$, which conflicts the definition of monotone transformations for t_h. The analogous argumentation applies for an element removal followed by its re-addition.

In consequence, each transformation t_h $(i < h \leq k)$ could neither add nor remove an element, which conflicts with the definition of monotone transformations. Thus, our assumption that there are two prefixes that yield the same inconsistent models does not hold, which proves the lemma. □

With that insight, it is easy to see that given only monotone transformations, no alternation can occur in Algorithm 7.1.

Theorem 7.13 (Monotone Transformations Alternation Prevention)
Let $\mathfrak{t}$ be a set of correct, monotone synchronizing transformations. Then the execution of Algorithm 7.1 cannot be alternating according to Definition 7.6, as long as ORCHESTRATE *does not return a transformation whose models are already consistent.*

Proof. According to Lemma 7.12, monotone transformations ensure that in an orchestration that does not contain transformations that need to be applied to already consistent models, the application of two prefixes never yields the same changes. In consequence, the sequence $\delta_{\mathfrak{M},generated}[]$ in the transformation execution loop (Lines 7–13) of Algorithm 7.1 can never contain the same two changes, which conflicts Definition 7.6 for alternation. □

In fact, the guarantee of not producing the same state twice is even stronger than non-alternation, because alternation allows to pass the same state multiple times, as long as the same sequence of states is not passed repeatedly and infinitely. It does, however, only make sense to pass the same state twice if the orchestration algorithm, which selects the next transformation to execute, is able to process that situation by trying different execution orders if an alternation occurs. Thus, the less strict requirement for alternation is suited to make statements about the orchestration strategy but not about the individual transformations, as it is unlikely to find a property for a single transformation that gives a guarantee that depends on the execution order of transformations, like alternation does.

Monotone transformations guarantee non-alternation, but monotony according to Definition 7.7 is not a property that we can assume to be fulfilled by all transformations. Although it seems intuitive that a transformation should not remove elements that were added before and vice versa, this does also mean that, for example, an attribute value may only be changed once by the transformations. This would, however, require the transformations to always make a choice for attributes that fits for all other transformations as well. We have seen in different examples, such as the one depicted in Figure 7.2 and Figure 7.3, that it may be necessary to change elements multiple times, because the transformations select values with which the models only fulfill their own consistency relation but not those of the other transformations. It may take several executions to find a value selection with which the models are consistent to all transformations. We might say that the transformations need to *negotiate* a consistent solution.

Still, the given examples were rather artificial and are not an indicator for monotony to be practically unachievable. It may, at least in some cases, be possible to specify monotone transformations. Even if only some of the transformations or only specific rules of them are monotone, it improves the chance that an orchestration strategy finds a consistent orchestration. Having the knowledge about the benefits of monotony gives a transformation developer the ability to implement it as often as possible.

Finally, the possibility to avoid alternation by construction can be combined with the ability of an orchestration strategy to react to alternation. We have discussed in Subsection 7.3.2 that an orchestration strategy can detect alternation and adapt its strategy of selecting the next transformation in that case. In addition, if monotony is given at least for some transformations, the orchestration strategy needs to try less execution orders and thus improves the chance of finding a consistent orchestration.

7.4. A Conservative Application Algorithm

We have argued why it is inevitable that any algorithm realizing an application function cannot be optimal and thus will not be able to find a consistent orchestration although it exists and, in that case, either return $\perp$ or not even terminate at all. Apart from minor improvements, such as the avoidance or detection of alternations, to improve the probability to find a consistent orchestration, or general strategies like backtracking for trying different orchestrations, we did not find systematic ways to improve optimality of the application function. Nevertheless, we want to find an algorithm that is at least correct and does always terminate, even if it does not implement a systematic way to improve optimality. Thus, it operates conservatively.

It is possible that Algorithm 7.1 does not terminate, because it generates an infinitely long orchestration, thus never leaving the loop in Lines 7–13. To ensure termination, we need to introduce an upper bound for the number of executed transformations. We have shown in Theorem 7.7 that no natural upper bound exists, thus even the shortest consistent orchestration for specific inputs can be arbitrarily long. Any arbitrary bound can prevent the algorithm from finding consistent orchestrations.

From an engineer's perspective, we may, however, consider the behavior that an arbitrary high number of transformation executions is required to yield consistent models as unwanted. Although the examples we have given are valid, they are rather artificial. We claim that a transformation network that requires a rather high number of executions compared to the number of contained transformations to find consistent models does not operate as expected. In particular, if such a high number of executions is required to find a consistent orchestration, it will be difficult to identify the reason for not finding a consistent execution in case the algorithm returns $\perp$. Thus, we introduce an artificial upper bound for the number of transformation executions. This bound will be well-defined, such that we can reasonably assume that no more executions are practically necessary.

In the following, we propose design goals for a conservative application algorithm. We derive the so called *provenance algorithm* as a practical realization and finally prove its correctness and termination properties. The algorithm was developed together with Joshua Gleitze in a scientific internship and also published in an article [GKB21].

7.4.1. Design Goals

An adapted version of Algorithm 7.1 that always terminates has two degrees of freedom. First, the execution order of transformations needs to be determined by defining the function Orchestrate. Second, an upper bound for the number of executions of transformations, thus the number of loop executions in Lines 7–13, needs to be defined.

We have discussed that improving optimality is not an achievable goal when determining the transformation execution order by the Orchestrate function. Since we know that the algorithm will always produce false negatives, i.e., it will not find a consistent orchestration although it exists, it is important for a transformation developer or user to be able to identify the reasons in case of such a failure. The algorithm can support them in this regard by delivering the final state of the models when the orchestration aborted. The execution order that was chosen until that state was reached is of central importance for identifying the reasons for failing. Consider that transformations are executed in an arbitrary order and then only some of the models of the final state are actually consistent. Apart from investigating the complete sequence of executed transformations, there is no clue for the user to find the reasons for the algorithm to fail, thus about *provenance* of the error. We have introduced this goal as the *comprehensibility* property in Subsection 1.1.3.

To improve identifying the reason whenever the algorithm fails, we propose the following principle for determining an orchestration:

> "Ensure consistency among the transformations that have already been executed before executing a transformation that has not been executed yet." [GKB21]

The principle requires that consistency is ensured incrementally for subsets of the transformations and thus the models. As long as the models are not consistent to all already executed transformations, only these transformations instead of other ones may be executed until the models are consistent to all of them. This ensures that consistency is preserved after each change in an incremental way, iteratively improving the number of models and transformations for which consistency is restored.

This approach helps to identify provenance of a failure of the algorithm, because it restricts the potentially causal transformations to consider. If the algorithm fails after executing a subset of the transformations $\mathbb{t}_{exec} \subseteq \mathbb{t}$, then there is some transformation $\mathtt{t} \in \mathbb{t}_{exec}$ that is the last of those transformation that was executed for its first time. Thus, the algorithm found an orchestration of $\mathbb{t}_{exec} \setminus \{\mathtt{t}\}$ such that the models were consistent to $\mathbb{t}_{exec}$, but it was not able to execute $\mathtt{t}$ and the transformations in $\mathbb{t}_{exec}$ afterwards such that the models become consistent to $\mathbb{t}_{exec} \cup \{\mathtt{t}\}$. This helps the transformation developer or user to understand and find the reason for failing in different ways. First, he or she can ignore any transformation in $\mathbb{t} \setminus \mathbb{t}_{exec}$, as the algorithm already failed to preserve consistency according to the other transformations, which can significantly reduce the number of transformations to consider. Second, the realization of $\mathtt{t}$ is somehow conflicting with the other transformations in $\mathbb{t}_{exec}$. This does not necessarily mean that there is something wrong with $\mathtt{t}$ but that also considering this transformation either induces the situation that no consistent orchestration exists anymore or that it cannot be found. Third, having a state of the models that is consistent to $\mathbb{t}_{exec} \setminus \mathtt{t}$ can be used as a starting point to either identify the reasons for failing or to manually restore consistency of the models.

If the algorithm operates according to the introduced principle and is not able to preserve consistency after it considers an additional transformation $\mathfrak{t}$ anymore, the selected execution order provides the discussed benefits for identifying the reasons for failing. There may, however, be another orchestration that is able to ensure consistency to $\mathbb{t}_{exec}$. Executing $\mathfrak{t}$ earlier or integrating further transformations in $\mathbb{t}$ before ensuring consistency to all transformations in $\mathbb{t}_{exec}$ can, of course, result in the algorithm finding a consistent orchestration. This can reduce optimality of the realized orchestration function, but we claim the discussed benefits to outweigh that.

We have shown that there is no inherent upper bound for the necessary number of transformation executions. Rather than specifying a concrete number, be it fixed or depending on the network size, we derive a reasonable artificial bound for the number of executions from a property that we assume reasonable for possible orchestrations of a transformation set. The idea of that property is that each transformation should be allowed to react to the execution of each possible sequence of all other transformations. If a transformation reacted to all these execution sequences of other transformations and if then other transformations are executed again, it should not be necessary that the transformation must be executed again to restore consistent. Thus, if a transformation was executed after applying the other transformations in any possible order, we expect the models to be consistent to that transformation. We define this in the following property.

Definition 7.8 (Reactive Converging Transformations)
A set of synchronizing transformations $\mathbb{t}$ is *reactive converging* with respect to models $\mathfrak{m}$ and changes $\delta_{\mathfrak{M}}$ if every orchestration of every subset $\mathbb{t}_p \subseteq \mathbb{t}$ in which a transformation $\mathfrak{t} \in \mathbb{t}_P$ has been executed after a sequence of transformations in $\mathbb{t}_p$ that contains each permutation of those transformations as a (not necessarily continuous) subsequence yields models that are consistent to $\mathfrak{t}$.

The property does not require that the other transformations were executed in each order consecutively, but only that the orchestration contains each permutation of those transformations, but potentially with other transformations in between. As an example, assume a set of transformations $\{\mathfrak{t}_1, \mathfrak{t}_2, \mathfrak{t}_3\}$, which is reactive converging for some input of models and changes. After executing them for these models and changes in the order $[\mathfrak{t}_1, \mathfrak{t}_2, \mathfrak{t}_3, \mathfrak{t}_1, \mathfrak{t}_2, \mathfrak{t}_3]$, the models yielded by that orchestration may still be inconsistent to t_1, because it was not executed after the order of the transformations $[t_3, t_2]$. After executing t_1 once more, the orchestration must yield consistent models, because t_1 was executed after the two orders of the other transformations $[t_2, t_3]$ and $[t_3, t_2]$. Likewise, t_2 was executed after $[t_1, t_3]$ and $[t_3, t_1]$, and t_3 was executed after $[t_1, t_2]$ and $[t_2, t_1]$.

7.4.2. The Provenance Algorithm

We propose an algorithm that realizes the discussed design goal with the function ProvenanceApply in Algorithm 7.2. The algorithm is a derivation of the general algorithm implementing an application function depicted in Algorithm 7.1. It first checks for consistency of the given models as a prerequisite for executing the transformations. Then the algorithm calls the recursive function Propagate, which implements the orchestration of transformations and returns a change tuple that is yielded by the determined orchestration, which delivers consistent models if applied to the input models. While this behavior is equal to the one in Algorithm 7.1, the orchestration itself is implemented differently in a recursive rather than an iterative manner, which implicitly ensures termination.

The function Propagate implementing the orchestration in a recursive manner acts as follows. It selects one of the transformations as a candidate to execute next. This selection ensures that a transformation is selected whose models are affected by any already performed change, such that the transformation may need to perform changes. Models are affected by a change if any of the two changes in $\delta_{\mathfrak{M}}$ for either of the models that are kept consistent by the selected transformation is not the identity function δ_{id}. It then applies the transformation using the generalization function Gen. If the selected transformation is not defined for the given models and changes, the function may return $\perp$, so that the complete algorithm terminates with $\perp$. Afterwards, it recursively executes the function Propagate with the subnetwork given by the transformations that have already been executed and are stored in $\mathfrak{t}_{executed}$. After that recursive execution, the selected transformation is executed again, and it is checked whether the models yielded by the resulting changes are still consistent to the executed transformations. If this consistency check fails, the transformations do not fulfill the definition of being reactive converging according to Definition 7.8, as we prove later. If the

Algorithm 7.2 The provenance algorithm. Adapted from [GKB21, Alg. 1].

procedure $\textsc{ProvenanceApply}(\mathbb{t}, \mathfrak{m}, \delta_{\mathfrak{M}})$
  if $\neg\textsc{CheckConsistency}(\mathbb{t}, \mathfrak{m})$ **then**
    return $\perp$
  end if
  $\delta_{\mathfrak{M},res} \leftarrow \textsc{Propagate}(\mathbb{t}, \mathfrak{m}, \delta_{\mathfrak{M}})$
  if $\delta_{\mathfrak{M},res} = \perp$ **then**
    return $\perp$
  end if
  return $\delta_{\mathfrak{M},res}(\mathfrak{m})$
end procedure
procedure $\textsc{Propagate}(\mathbb{t}, \mathfrak{m}, \delta_{\mathfrak{M}})$
  $\mathbb{t}_{executed} \leftarrow \emptyset$
  for $\mathfrak{t}_{candidate} \in \mathbb{t} \setminus \mathbb{t}_{executed} \mid \delta_{\mathfrak{M}}.\mathit{affects}(\mathfrak{t}_{candidate})$ **do**
    $(\mathfrak{m}, \delta_{\mathfrak{M},candidate}) \leftarrow \textsc{Gen}_{\mathfrak{M},\mathfrak{t}_{candidate}}(\mathfrak{m}, \delta_{\mathfrak{M}})$
    if $(\mathfrak{m}, \delta_{\mathfrak{M},candidate}) = \perp$ **then**
      return $\perp$
    end if
    $\delta_{\mathfrak{M},propagation} \leftarrow \textsc{Propagate}(\mathbb{t}_{executed}, \mathfrak{m}, \delta_{\mathfrak{M},candidate})$
    if $\delta_{\mathfrak{M},propagation} = \perp$ **then**
      return $\perp$
    end if
    $(\mathfrak{m}, \delta_{\mathfrak{M},candidate}) \leftarrow \textsc{Gen}_{\mathfrak{M},\mathfrak{t}_{candidate}}(\mathfrak{m}, \delta_{\mathfrak{M},propagation})$
    if $(\mathfrak{m}, \delta_{\mathfrak{M},candidate}) = \perp$ **then**
      return $\perp$
    end if
    if $\neg\textsc{CheckConsistency}(\mathbb{t}_{executed}, \delta_{\mathfrak{M},candidate}(\mathfrak{m}))$ **then**
      return $\perp$
    end if
    $\delta_{\mathfrak{M}} \leftarrow \delta_{\mathfrak{M},candidate}$
    $\mathbb{t}_{executed} \leftarrow \mathbb{t}_{executed} \cup \{\mathfrak{t}_{candidate}\}$
  end for
  return $\delta_{\mathfrak{M}}$
end procedure

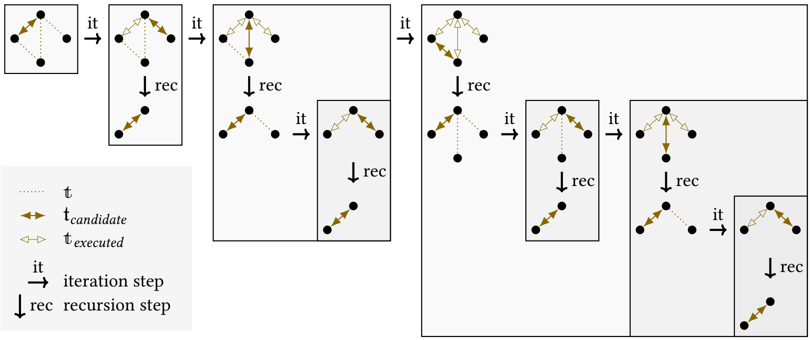


Figure 7.7.: Exemplary execution of the provenance algorithm for a change in the topmost model. The transformations present to the current execution of Propagate, as well as the executed and candidate transformations $\mathfrak{t}_{executed}$ and $\mathfrak{t}_{candidate}$ are depicted for each iteration (horizontal) and recursion step (vertical). Adapted from [GKB21, Fig. 4].

models are consistent to the transformation, the next candidate is picked. In effect, the strategy realizes the defined principle in a recursive manner, because after executing a new transformation, the recursive execution ensures consistency to all already executed transformations by applying all already executed transformations again.

Figure 7.7 depicts an exemplary execution of the ProvenanceApply algorithm for a set of four transformations between four metamodels. We assume that the algorithm receives four initially consistent models and a change to the topmost one. The example shows that in each recursion step only the subnetwork of the already executed transformations in $\mathfrak{t}_{executed}$ is considered. Thus, the set of transformations becomes smaller in each recursive call of ProvenanceApply.

7.4.3. Correctness, Termination and Goal Fulfillment

The provenance algorithm is intended to implement a correct application function and to always terminate. Additionally, it is supposed to deliver consistent models whenever the given transformations fulfill Definition 7.8 for being reactive converging. In the following, we prove that the algorithm actually fulfills these properties.

First, it is easy to see that the algorithm always terminates and always either returns consistent models yielded by an orchestration of the given transformations or $\perp$, which realizes a correct application function according to Definition 4.13 and Definition 4.14.

Theorem 7.14 (Provenance Algorithm Termination)
Algorithm 7.2 terminates for every possible input.

Proof. The algorithm terminates if CheckConsistency, Gen and Propagate terminate. We assume termination for the external function CheckConsistency, because it only validates consistency of the given models. Propagate contains a loop with a recursive call and the external calls of CheckConsistency as well as Gen. Since CheckConsistency and Gen terminate, it may only be non-terminating because of the loop in Line 13 and the recursive call in Line 18. The number of loop executions is limited by the number of given transformations, i.e., $|\mathbb{t}|$, as each iteration selects another transformation and adds it to $\mathbb{t}_{executed}$. Thus, after selecting each transformation once, all transformations are in $\mathbb{t}_{executed}$ and the loop condition is not fulfilled. The recursive call receives a set of transformations that is at least one element smaller than the set of transformations given to the calling method, because if $\mathbb{t}_{executed} = \mathbb{t}$ the loop condition is not fulfilled. If the given set of transformations is empty, the loop is not entered and thus no recursive call is performed. Thus, the recursion depth never exceeds $|\mathbb{t}|$. □

Theorem 7.15 (Provenance Algorithm Correctness)
Algorithm 7.2 realizes a correct application function.

Proof. The algorithm receives models and changes to them and it returns models being instances of the same metamodels, thus it fulfills the signature of an application function. Additionally, if it returns models, they are the result of a consecutive application of transformations in $\mathbb{t}$, as Propagate calculates the changes that are applied to the input models to calculate the result by a repeated application of the generalization function Gen to transformations in $\mathbb{t}$. Thus, Propagate implicitly implements an orchestration function according to Definition 4.11 and applies the transformations in the determined order to calculate the result delivered by ProvenanceApply. Thus, ProvenanceApply fulfills Definition 4.13 for an application function.

Let us assume that Algorithm 7.2 does not realize a correct application function. ProvenanceApply may return $\perp$ in Line 3 or Line 7, or it may return models in Line 9. Correctness requires the function to either return $\perp$ or consistent models, which may only be violated by ProvenanceApply returning inconsistent models. This means that for some input models and changes, ProvenanceApply returns models $\mathfrak{m}_{res} := \delta_{\mathfrak{M},res}(\mathfrak{m})$, such that there is a transformation $\mathfrak{t} \in \mathbb{t}$ to which $\mathfrak{m}_{res}$, or more specifically two contained models m_i and m_k, whose metamodels are related by $\mathfrak{t}$, are not consistent. We distinguish the following three cases.

1. $\mathfrak{t}$ was never executed by Propagate. This means that the changes δ_{M_i} and δ_{M_k} in $\delta_{\mathfrak{M}}$ of the two models that are kept consistent by $\mathfrak{t}$ were always empty, i.e., δ_{id}, because otherwise $\mathfrak{t}$ would have been selected in the loop header. Since the initial models m_i and m_k were consistent to $\mathfrak{t}$, the returned models are still consistent, because only the identity function is applied to them.

2. $\mathfrak{t}$ was executed producing changes δ_{M_i} and δ_{M_k}, and no other transformation that affects m_i or m_k was executed afterwards. Then the returned models, i.e., $\delta_{M_i}(m_i)$ and $\delta_{M_k}(m_k)$ are consistent by definition of correctness for $\mathfrak{t}$.

3. $\mathfrak{t}$ was executed, and another transformation $\mathfrak{t}' \in \mathbb{t}$ that involves m_i or m_k was executed afterwards. Since $\mathfrak{t}'$ was executed after $\mathfrak{t}$, $\mathfrak{t}$ was in $\mathbb{t}_{executed}$ when $\mathfrak{t}'$ was the candidate $\mathfrak{t}_{candidate}$. After executing the transformations in $\mathbb{t}_{executed}$, the candidate $\mathfrak{t}'$ is applied again in Line 22. Additionally, consistency to all transformations in $\mathbb{t}_{executed}$ is ensured by the check in Line 26 after returning from the recursion in which $\mathfrak{t}$ was executed. Thus, the returned models are consistent to $\mathfrak{t}$ and $\mathfrak{t}'$.

The third case can be applied inductively if a transformation is followed by multiple transformations that involve the same models. Thus, all cases lead to a contradiction. □

In addition to these essential properties, we can also derive the upper bound for the number of transformation executions by the algorithm.

Theorem 7.16 (Provenance Algorithm Complexity)
Algorithm 7.2 executes transformations at most $\mathcal{O}(2^{|\mathbb{t}|})$ *times.*

Proof. Let $T(m)$ denote the number of transformation executions the algorithm invokes for a set of transformations $\mathbb{t}$ with $m = |\mathbb{t}|$. The set $\mathbb{t}_{executed}$ is initialized to be empty (Line 12) and grows by one transformation every iteration of the loop (Line 30). It follows that the recursive call in Line 18 receives a set of transformations that contains one more transformation in each iteration. Thus, given m transformations, Propagate executes each of them in the loop and then makes recursive calls for 0 to $m-1$ transformations:

$$T(m) = 2m + \sum_{i=0}^{m-1} T(i) = 2 + 2\,T(m-1) = 2(2^m - 1) \in \mathcal{O}(2^m)$$
$$T(0) = 0$$
□

Finally, the algorithm shall implement the principle to ensure consistency among the transformations that have already been executed before executing a transformation that has not been executed yet, defined in Subsection 7.4.1.

Theorem 7.17 (Provenance Algorithm Design Principle)
Algorithm 7.2 ensures consistency among the transformations that have already been executed before executing a transformation that has not been executed yet.

Proof. After the recursive call in Line 18, the model tuple yielded by applying the current changes $\delta_{\mathfrak{M},candidate}$ to the initial model tuple $\mathfrak{m}$ is consistent to all executed transformations in $\mathbb{t}_{executed}$ according to the proof given for Theorem 7.15. □

We have given Definition 7.8 for the property of a transformation set to be reactive converging. This property defines that we do not want transformations to be required to react to changes they performed themselves after all other transformations have been executed in all possible permutations, as we assume this to be a reasonable property that induces an upper bound for the number of transformation executions. We have used this property as a design goal for the proposed algorithm and can now show that the algorithm always returns consistent models if the transformations fulfill that property, which means that the algorithm implements an optimal application function for these kinds of transformations.

> **Theorem 7.18** (Provenance Algorithm Optimality)
> *If the transformation set $\mathbb{t}$ passed to Algorithm 7.2 is reactive converging according to Definition 7.8 and if the consistency preservation rules of all transformations in $\mathbb{t}$ are total functions, then the algorithm implements an optimal application function.*

Proof. We show that the algorithm does not return $\bot$ when the input models are consistent, thus an orchestration is always found. This is even stronger than optimality, because it means that for every input with consistent models a consistent orchestration exists.

Since optimality allows the algorithm to return $\bot$ when the input models are inconsistent, returning $\bot$ in Line 3 is valid. The algorithm returns $\bot$ in Line 7 if PROPAGATE returns $\bot$, thus we show that PROPAGATE does not return $\bot$. PROPAGATE returns $\bot$ in Line 16 and Line 24 if the application of a $\mathfrak{t}_{candidate}$ in Line 14 or Line 22, respectively, returns $\bot$, which cannot occur because transformations are total by assumption. PROPAGATE returns $\bot$ in Line 20 if a recursive call returns $\bot$. If the loop in that recursive call is executed, the arguments for not returning $\bot$ apply recursively. If the loop is not executed in the recursion, the input changes are returned, thus not yielding $\bot$.

Finally, PROPAGATE returns $\bot$ in Line 27 if the models yielded by applying the changes after the recursive call and reapplying $\mathfrak{t}_{candidate}$ are not consistent with the already executed transformations in $\mathbb{t}_{executed} \cup \{\mathfrak{t}_{candidate}\}$. Since the transformation set is reactive converging, this can only be the case if not all permutations of the transformations in $\mathbb{t}_{executed} \cup \{\mathfrak{t}_{candidate}\}$ have been executed yet. We first note that applying the PROPAGATE function to transformations $\mathbb{t}$ with $|\mathbb{t}| = m$, the result after the first $n \leq m$ loop iterations is the same as when executing PROPAGATE to the n transformations in $\mathbb{t}_{executed}$ after n loop iterations. We thus show that when reaching Line 26 in the last iteration of the loop, i.e., when the algorithm returns consistent models if the check in that line does not fail, every permutation of transformations in $\mathbb{t}$, and thus in $\mathbb{t}_{executed} \cup \{\mathtt{t}_{candidate}\}$, has been executed by induction. Applying PROPAGATION to a transformation set with $|\mathbb{t}| = 1$, the statement is trivially true, because the single transformation is executed in Line 14. Let us assume that the statement is true for a transformation set with $|\mathbb{t}| < i$, but that it is not true for a set with $|\mathbb{t}| = i$. Since the execution of the first $i-1$ iterations is equal to executing the algorithm on the $i-1$ transformations selected in these iterations, the algorithm cannot return $\perp$ in Line 27 in the first $i-1$ iterations by induction assumption. Thus, only in the last iteration not all permutations of transformations may have been executed and thus the check in Line 26 may only fail in that last iteration. This means that there is a permutation $[\mathtt{t}_1, \ldots, \mathtt{t}_i]$ of the transformations in $\mathbb{t}$ after the last loop iteration in which they have not been executed yet. Let $\mathtt{t}$ be the candidate $\mathtt{t}_{candidate}$ of the last loop iteration, and let k be the index of $\mathtt{t}$ in that sequence, i.e., $\mathtt{t} = \mathtt{t}_k$. Then per induction assumption, the sequence $[\mathtt{t}_1, \ldots, \mathtt{t}_{k-1}]$ has been executed in one of the previous iterations of the loop. Afterwards $\mathtt{t}$ was executed in Line 14. Then, the sequence $[\mathtt{t}_{k+1}, \ldots, \mathtt{t}_i]$ has been executed in the recursive call in Line 18 by induction assumption. Since during the last iteration the recursive call is performed with $\mathbb{t}_{executed} = \mathbb{t} \setminus \{\mathtt{t}\}$ and thus $|\mathbb{t}_{executed}| = |\mathbb{t}| - 1$, all permutations of transformations in $\mathbb{t}_{executed}$, including $[\mathtt{t}_{k+1}, \ldots, \mathtt{t}_i]$, are executed in the recursive call by induction assumption. This is a contradiction.

In consequence, PROPAGATE and thus PROVENANCEAPPLY do never return $\perp$, except for inconsistent input models. Since we have already proven that the algorithm terminates always and implements a correct application function, this shows that it implements an optimal application function. □

Optimality can, however, only be guaranteed under specific conditions. Apart from the necessity to be reactive converging, the transformations need to be able to handle every input, i.e., every combination of models and changes, as otherwise selecting a transformation may lead to PROPAGATE returning $\perp$, because the transformation cannot be applied. In practice, this assumption may not be fulfilled. Nevertheless, it is theoretically possible to define such transformations and, at least, it leads to well-defined conditions for when we can assume the algorithm to realize an optimal orchestration function.

Although this means that under such specific conditions the algorithm is able to decide the orchestration problem, the problem is actually trivially solved in that case, because for every input there is a consistent orchestration. Thus, the problem is actually non-existent under these assumptions.

Finally, it is an open question how far we can assume sets of transformations to be reactive converging in practice. We have, however, not introduced this as a property that should be fulfilled by transformations, as it is obviously hard to ensure or even analyze this property. In fact, it is only supposed to be a well-defined property that allows us to define a reasonable upper bound for the execution of transformations and thus to allow us to define an algorithm that always terminates without using a completely arbitrary upper bound for determining when to terminate.

7.4.4. Provenance Identification Improvement

We have motivated the provenance algorithm with the idea to improve the ability of a transformation developer or user to find the reason for the algorithm not to yield consistent models for certain inputs. The proposed Algorithm 7.2 only returns $\perp$ in these situations and does thus not directly support that process. The necessary information for improving the identification of provenance for the failure is, however, present in the algorithm and can be retrieved easily.

The algorithm may fail, because it is, at some point, not able to execute a candidate transformation (Line 16 or Line 24), or because after executing a new transformation consistency to the previously executed transformations cannot be achieved without letting one of the transformations react to the reaction of all other transformations to its own changes (Line 26), which we defined as the property of reactive convergence. In this case, we at least know that after the previous loop iteration consistency to all transformations that have been executed so far could be achieved.

Whenever the Propagate function fails and returns $\perp$, we know that for the current transformations in $\mathbb{t}_{executed}$ an orchestration exists that yields the current changes in $\delta_{\mathfrak{M}}$, for which we know that when applied to the original models the result $\delta_{\mathfrak{M}}(\mathfrak{m})$ is at least consistent to $\mathbb{t}_{executed}$. We also know that the algorithm was not able to ensure consistency to the current candidate transformation $\mathfrak{t}_{candidate}$. This is exactly the information for which we already discussed in Subsection 7.4.1 the benefits with respect to the underlying design principle of recursively ensuring consistency for subsets of the transformations for the ability to identify the reasons for not finding a consistent orchestration. Thus, implementing the algorithm such that it also delivers $\mathfrak{t}_{candidate}$, $\mathbb{t}_{executed}$, and the current changes $\delta_{\mathfrak{M}}$ reduces the necessary model states and transformations to consider for a transformation user or developer to identify why no consistent orchestration was found.

The algorithm and the ability to identify reasons for the algorithm to fail may be further improved by determining a reasonable order for the execution of transformations in the loop of the Propagate function. The loop at least ensures that no transformations are executed that are not yet affected by any change and thus would not produce changes. It can, however, also be reasonable to first select transformations for which both models have already been modified before selecting transformations for which only one model has been modified. This can further improve locality of the changes performed until the algorithm fails, because less models may have been modified until the algorithm fails. We also discuss these benefits as results of the evaluation in Section 9.3.

7.5. Summary

In this chapter, we have discussed how we can realize an application function for transformation networks. We have motivated optimality as a desired property, which ensures that an application function always delivers consistent models if there is an order of the transformations that yields them. From this optimality notion, we have derived the central orchestration problem, for which we haven proven undecidability even when restricting transformation networks. Finally, we have proposed strategies to reduce the cases in which no consistent models are found and an algorithm that executes transformations with a well-defined order and bound. Rather than improving optimality, it ensures that in cases in which no consistent models can be derived at least some information can be provided that helps developers or users of transformations to identify why no consistent models were found. We conclude this chapter with the following central insight.

Insight II.4 (Orchestration)

The *orchestration problem*, whether an orchestration of modular and independently developed transformations exists that restores consistency for given models and changes, is undecidable. We have shown that the problem stays undecidable even with impractical restrictions to the individual transformations, such that we need to accept undecidability of the problem. In consequence, every algorithm that realizes an application function for transformations can only implement a conservative approach to the orchestration problem. Due to this conservativeness, every algorithm will fail in cases in which actually an orchestration of the transformations exists that leads to consistent models. Thus, it is useful to find an algorithm that orchestrates the transformations in a way such that the state of executed transformations and generated changes can help the transformation developer or user to identify why the algorithm failed. This can be achieved with a strategy of iteratively restoring consistency, such that always a subset of the transformations for which consistency could be restored and a transformation for which it could not be restored anymore can be provided to ease reasoning about the cause for failing. We have proposed an algorithm that implements this strategy and is proven to fulfill the desired property.

8. Classifying Errors in Transformation Networks

In the previous chapters, we have introduced a notion of correctness for transformation networks and discussed how we can achieve or analyze different kinds of correctness for the different artifacts of a transformation network, namely consistency relations, consistency preservation rules, and the application function. It may, however, easily occur that transformation developers define transformations that do not adhere to all these kinds of correctness, be it because of missing knowledge about them or by accident.

In this chapter, we discuss what may happen if correctness was not achieved. The possible types of errors that can occur depend on the abstraction level at which the specification is performed. This depends on existing knowledge, i.e., whether the transformation is to be used in a transformation network or even in which network it is to be used, but also on the abstraction provided by the formalism or language to specify a transformation or transformation network in. We first propose a distinction of such knowledge levels for the specification of transformation networks. We then systematically derive a categorization of potential *failures*, i.e., the unwanted results the application algorithm may yield, the *faults* that led to the failures, i.e., the errors in the implementation of the transformation, and finally the causing *mistakes*, i.e., the errors made by a developer due to his or her knowledge that led to an implementation fault. Finally, we discuss how the possible types of mistakes, faults, and failures can be detected or avoided and how this relates to the correctness notions and the introduced approaches to achieve correctness.

This chapter thus constitutes our contribution **C 1.5**, which consists of three subordinate contributions: a separation of knowledge-dependent specification levels for transformation networks; a categorization of potential errors in transformation networks; and a discussion of the possibilities to detect and avoid errors with respect to the discussed correctness notions and measures to achieve them. It answers the following research question:

RQ 1.5: Which errors can occur in transformation networks, how can they be classified regarding their avoidability, and how severe are they?

As the central goal of this chapter, we categorize the possible types of errors to derive systematic knowledge about mistakes that can be made and failures that can arise from them. First, this helps transformations developers to identify the reasons for arising failures. Second, it allows us to identify which relevant errors we can avoid or detect with the approaches proposed in the previous chapters and how relevant the problems that we solve with them are. The latter will be part of our subsequent evaluation at case studies.

Several of the insights regarding errors in transformation networks are results of the two Master's theses by Syma [Sym18] and Sağlam [Sağ20], who investigated errors that occurred when combining independently developed transformations in two case studies. Essential results from the former thesis were published in previous work [Kla+19b] and will be presented in the following sections in revised form.

8.1. Knowledge Levels in Transformation Specifications

The process of specifying a transformation network can be considered at different conceptual levels depending on the knowledge a developer must have to ensure correctness at that level. For example, at the lowest level a developer may only know that a transformation shall be used within a network without knowing the actual network, which only allows to avoid specific errors, whereas further errors are relevant and need to be considered when having knowledge about the other transformations to combine it with. In addition, depending on the level of abstraction that a specification formalism, such as a transformation language, provides, the developer must only deal with some of these levels as the language abstracts from the others, which determines the resulting challenges a developer has to deal with. In consequence, these levels are supposed to mean that specific kinds of mistakes can be made at each of them and that a formalism may ensure correctness with respect to one of those levels and the ones below, whereas

Level	Name	Correctness	Knowledge
1	Transfor-mation	Synchronizing transformations	Individual transformation
2	Network Relation	Compatible consistency relations	Consistency relations of complete network
3	Network Rule	Interoperable consistency preservation rules	Transformations of complete network

Table 8.1.: Distinguished levels in the transformation network specification process with their correctness criteria and required knowledge.

the transformation developer is still responsible for avoiding mistakes at the levels above.

We distinguish three such levels, which we summarize in Table 8.1 together with their properties and discuss them in the following. At the *transformation level*, we consider the specific properties of a single transformation to be used in a, or more precisely any transformation network, especially involving synchronization. At the *network relation level*, we consider the interplay of the binary consistency relations of a concrete set of transformations. At the *network rule level*, we consider the interplay of the consistency preservation rules of a concrete set of transformations. These levels depend on each other, because, for example, consistency preservation rules cannot properly work together if each on its own is not at least synchronizing and thus correct at the transformation level. Nevertheless, a transformation can be correct at the transformation level without being correct at the network relation and network rule level.

These levels especially differ in what knowledge they require to be able to deal with and even avoid potential errors. For the transformation level, it is sufficient to know that a transformation may be used in a transformation network without knowing the actual network. For the network relation level, at least the relations of the other transformations in the network must be known. Finally, for the network rule level, the transformations of the complete network must be known. This influences how far errors at the different levels can be avoided, first, because of the required knowledge to do so and, second, because of the possibility to ensure correctness at all.

8.1.1. Knowledge-Dependent Specification Levels

In the following, we introduce the three mentioned levels more precisely. They represent a revised version of the three levels we have presented in previous work [Kla+19b]. In that work, we have discussed the *global* level, which considers the global knowledge in terms of the overall, multiary relation between all involved models. We have, however, discussed different correctness notions in Section 4.2 and argued why we do not consider a *monolithic* notion of consistency, which conforms to the *global* specification level, as we do not assume this global knowledge to be represented explicitly, such that it would not make sense to explicitly consider correctness according to it.

Level 1 (*Transformation*): At the first level, we only consider the knowledge that a transformation shall be used within a transformation network. According to our formalism presented in Section 4.3, this means that the transformation needs to be *synchronizing*. We have discussed in Chapter 6 how synchronization can be achieved with ordinary transformation languages. Correctness at this level is given by the fulfillment of the synchronization property for a transformation.

Level 2 (*Network Relation*): At the second level, we consider the knowledge about the actual network in which the transformations shall be used, but restricted to their relations. In consequence, it would be possible that the relations between all models are known, e.g., because there is a common understanding of the relations, which may also be documented. We have discussed in Chapter 5 that *compatibility* is a relevant property of the consistency relations in a transformation network to ensure that the transformations are able to find consistent models after changes. Correctness at this level is thus given by compatibility of the consistency relations.

Level 3 (*Network Rule*): At the third level, we consider the knowledge about the complete transformations of an actual network, thus especially also the consistency preservation rules that preserve consistency. In Chapter 7, we have discussed the problem of orchestrating these rules and also discussed several issues that may prevent an algorithm from finding a consistent orchestration, such as the selection of different, conflicting options provided by a consistency relation to restore consistency.

8.1.2. Abstraction to Specification Levels

All three levels are relevant during the specification process of a transformation network, and potential mistakes that can be made at each of them need to be avoided. As mentioned before, a specification formalism, usually a transformation language, provides a specific level of abstraction associated with one of the conceptual levels introduced above, which relieves the developer from dealing with potential problems of the lower levels. He or she must, however, still ensure correctness with respect to all higher levels.

At the lowest level, a transformation language may not ensure correctness regarding any of the levels. For example, an imperative, unidirectional transformation language requires the developer to ensure synchronization of transformations at the transformation level, compatibility of the relations at the network relation level, as well as interoperability of the consistency preservation rules at the network rule level. Some declarative, bidirectional transformation languages already relieve the developer from specifying consistency preservation rules and lift the abstraction to consistency relations, from which consistency preservation rules are automatically derived. Some languages even relieve the developer from manually ensuring synchronization, for example, by using keys for matching existing elements in QVT-R. In this case, the transformation engine ensures correctness at the transformation level, but the developer still has to ensure it for the other levels. Then, the developer only needs to deal with problems at the higher levels. Integrating an analysis for compatibility, such as the one proposed in Chapter 5, into QVT-R could thus also abstract from the network relation level.

To the best of the author's knowledge, languages that ensure correctness at higher levels than the transformation level are currently uncommon. This would either require the specification of multidirectional transformations, i.e., a less modular or even monolithic notion of consistency (see Section 4.2), or at least additional analysis functionality integrated into the languages to, for example, ensure compatibility and thus correctness at the network rule level. Multidirectional QVT-R [MCP14] or extensions of TGGs to multiple models [KS06; TA15; TA16] provide means to define rules between multiple models, from which then consistency preservation rules between two models are derived, thus abstracting from the problems of ensuring rule compatibility and interoperability of consistency preservation rules. The Commonalities language [Gle17], which we present in detail in Chapter 11, lifts the abstraction such that the network relation and network rule levels do not have to be considered by the transformation developer. This is, however, achieved by a specific network topology induced by that language, which avoids several of the problems that we discussed for networks of arbitrary topologies.

Correctness at the higher conceptual levels always requires correctness at the lower levels. Especially the interoperability of transformations at the network rule level requires the transformations to be synchronizing, i.e., correct at the transformation level, and the relations to be compatible, i.e., to be correct at the network relation level. In fact, compatibility of the relations does not require the transformations to be synchronizing, thus the network relation level does, theoretically, not require correctness at the transformation level. From a knowledge perspective, it does, however, not make sense to ensure compatibility of relations when their transformations are not even synchronizing, because synchronization of a transformation can already be ensured independent from the other transformations to combine it with, whereas this knowledge is required for ensuring compatibility.

8.2. Categorization of Errors in Transformation Networks

In this section, we identify and categorize potential *failures* that can occur when executing transformation networks, which are derived from the failure cases of the application algorithm discussed in Chapter 7. We consider the *mistakes* and the resulting *faults* in the transformation specifications, which a transformation developer can make. The mistakes are specific for the introduced knowledge levels, thus we derive them from those levels. We finally relate mistakes to the failures that can occur when transformation networks containing faults caused by those mistakes are still executed.

8.2.1. Mistakes, Faults and Failures

Errors in transformation networks can occur in different contexts, for example in terms of the transformation networks, more precisely the application algorithm, producing an incorrect result, or in terms of a transformation developer defining an erroneous transformation. To be able to distinguish these contexts, we have already used the terms *mistakes*, *fault* and *failure* with a short introduction of their distinction, as specializations of the general term *error*. They are supposed to describe erroneous or inappropriate knowledge of a developer (mistakes), erroneous implementations (faults) and erroneous execution results (failures). These different types of errors depend on each other, as a mistake can lead to a fault, which can then lead to a failure.

Mistake: A mistake is made by a transformation developer. It is based on missing or erroneous knowledge about either the actual transformation or the necessity to ensure certain properties. For example, the missing knowledge that transformations must be synchronizing leads to a mistake in the conceptualization of transformations, as they do not ensure this required property. The missing knowledge that compatibility is required as well as the missing knowledge about the other transformations of a network can lead to the mistake that incompatible transformations are realized. If a transformation language abstracts from a conceptual level and relieves the developer from ensuring that no mistakes at that level are made, such mistakes can also be made by the transformation language developer and then manifest in a faulty implementation of the language. We do, however, not consider that case explicitly.

Fault: A fault is the manifestation of a mistake in the implementation of transformations. For example, the missing knowledge about the necessity to have synchronizing transformations can lead to the fault that the implementation does not properly identify existing elements instead of creating new ones. A fault is, thus, always the consequence of a mistake. It is also made by a transformation developer but can be seen within the implementation explicitly, whereas a mistake can only be detected by the fault in the implementation to which it led.

Failure: A failure occurs at execution time of transformations and is the manifestation of a fault when executing a faulty transformation network. A failure is the incorrect result of the execution of transformations. Whenever the transformations in a network have a faulty implementation, failures such as the termination in inconsistent states or non-termination of the application algorithm can occur. Since the occurrence of a failure depends on the scenario in which the transformations are executed, not every fault leads to a failure. On the other hand, a fault can also lead to several failures, e.g., because a transformation is executed multiple times.

Several similar terms like errors, mistakes, faults, bugs, defects, and so on are used in software engineering and especially in software testing. They are sometimes used interchangeably and sometimes with specific meanings. One common notion is the distinction of faults, errors, and failures in software testing, however also with different meanings, of which at least one is comparable to ours using the term *error* for what we call *mistake*. We decided to avoid the overloaded term *error* and make the human *mistake* explicit.

8.2.2. Possible Failure Types

Failures are the manifestation of faults during transformation execution and thus the final result of mistakes made by a transformation developer. A failure means that the execution of the transformation network, or more precisely the application algorithm, reached an unwanted state. We have already discussed in Subsection 7.2.2 that the application algorithm can fail by not implementing a correct application function, thus either returning models that are inconsistent or not terminating at all. Additionally, the algorithm may fail to deliver consistent models and return $\perp$ instead. Returning $\perp$ is actually desired behavior to deal with the undecidability of the orchestration problem. It can, however, mask that the transformations in the network contain faults that lead to the algorithm not being able to find an orchestration that yields consistent models.

Termination in an inconsistent state, non-termination, and returning $\perp$ already form the three general failure types that can occur when executing faulty transformations. They can be further specialized in different dimensions, e.g., regarding determinism of inconsistent termination or regarding whether too many or too few elements (or combinations of them) exist for being consistent. The latter could manifest in missing corresponding condition elements or the existence of too many condition elements for which no consistent models can be found by adding further ones. We have, however, found in previous work [Sağ20, Tab. 5.7] that this distinction regarding elements does not provide any insights and benefits when tracing the failures back to the causal mistakes. We do, however, consider *duplications* as one specific additional failure type, which can finally lead to any of the other

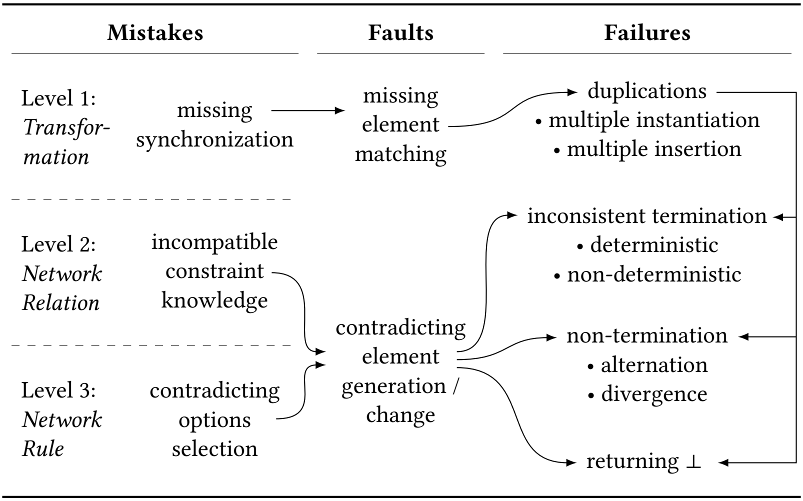


Figure 8.1.: Categorization of mistakes, faults and failures. Adapted from [Kla+19b, Fig. 3].

failures, depending on whether the application algorithm aborts or not. Duplications of elements are of particular importance, because they are the essential manifestation of missing synchronization in transformations, as we have discussed in Section 6.4.

In Figure 8.1, we depict the different failure types with their specializations, which we discuss in the following. Note that we do not assume a specific application algorithm when discussing failures. Whether a potential failure occurs or not highly depends on the used algorithm. For example, using the provenance algorithm proposed in Section 7.4 will neither lead to non-termination nor to inconsistent models, at least if the consistency check is implemented properly, but may only lead to returning $\perp$. Having an artificial upper bound for the number of transformation executions, of course, always prevents from non-termination. Only if transformations are executed without checking consistency afterwards or without defining an execution bound, the discussed failures can actually occur. Whenever an algorithm returns $\perp$, this can, however, be an indicator whether the algorithm fails because an artificial execution bound was reached or because a transformation cannot be applied anymore as it is not able to process the given changes. We will discuss that in Section 8.3.

We further distinguish the already discussed failure types as follows.

Inconsistent Termination: Inconsistent termination means that the application algorithm terminates and the models it returns are inconsistent. This can only occur if the algorithm does not check the models, which the application of the transformations yields, for consistency. Furthermore it can terminate *deterministically* or *non-deterministically*, depending on whether each execution delivers the same inconsistent models or different ones, because different execution orders of the transformations are selected.

Non-Termination: Non-termination means that the application algorithm does not terminate but executes transformations indefinitely without achieving a consistent state of the models. We can further distinguish between *alternation* and *divergence* as defined in Definition 7.6. Alternation means that the same model states are produced repeatedly, which can, for example, be because a feature, such as an attribute or reference, alternates between two or more values. In other cases, divergence occurs, which means that some feature values are changed indefinitely, such as a number counting up, a string being appended repeatedly, or an infinite number of elements being created. While an alternating algorithm can easily run endlessly, a diverging algorithm will abort at some point in time in many cases, because endless element creation or string concatenation can lead to an overflow of available memory.

Returning $\bot$: The application algorithm may terminate and return $\bot$ to indicate that it was not able to find an orchestration that yields consistent models. This may either be because no such orchestration exists or can be found even though no mistakes were made, or because the transformation network actually contains faults that prevents the algorithm from finding a consistent orchestration. For example, if transformations are not synchronizing, the application algorithm will, in general, not be able to execute them in a way that they deliver consistent models. This kind of failure is different from the others, as it is intended behavior of the algorithm to return $\bot$ rather than returning inconsistent models or not terminating at all, but it is still not the intended result as it is caused by an actual fault.

Duplications: As a more specific failure case, we have introduced element duplications, which can especially arise if transformations are not synchronizing and thus do not match existing elements rather than creating new ones. We can further separate this into *multiple instantiation* and *multiple insertion*. Multiple instantiation can occur because different consistency preservation rules instantiate an element multiple times, although all of them represent the same one. Multiple insertion can occur because an element is inserted into a reference or attribute list several times, although it should be inserted only once. In fact, such duplications can ultimately lead to inconsistent termination, non-termination, or returning $\perp$, either because the algorithm returns after a finite number of transformation executions without checking consistency or returning $\perp$, or because the transformations are not able to restore consistency and the algorithm does thus not terminate. Duplications, however, represent a special case, which, as we will see in the evaluation in Chapter 9, is one of the most important error cases for transformation networks. Thus, identifying such duplications in the generated models can ease finding the causal mistake in terms of missing synchronization.

We have discussed that if an application algorithm checks consistency and has an artificial execution bound, it will only return $\perp$ rather than producing any other type of failure, especially not the more specific duplications. Knowing the other failure types and their relation to the causal mistakes is still important. First, when a transformation network with such an application algorithm yields $\perp$ in most execution scenarios, there will likely be a fault in the transformation implementations. Temporarily replacing the algorithm with a less restrictive one can help to find the reasons, because then, for example, duplications may be detectable that help to identify missing synchronization. Second, in many transformation languages consistency relations are not represented explicitly, thus consistency checks are performed by executing the transformation and checking whether changes were performed. Then, if transformations are non-synchronizing, they return an actually inconsistent state, which may, however, not be identified by the transformation as such. This is due to the fact that these transformations do not expect to be used in the synchronization scenario and thus assume that consistency is achieved by construction, i.e., that only changes for one model are given and must be processed and, thus, that the models are consistent after executing the consistency preservation rules.

8.2.3. Mistake and Fault Types

Developers can make different kinds of mistakes at each of the specification levels, which lead to faults in the implementation of transformations and eventually to different kinds of failures during transformation execution. In the following, we derive mistakes and faults from the specification levels, as depicted in Figure 8.1.

We explicitly focus on conceptual mistakes and faults concerned with the development of transformation networks. This especially excludes the following two types of mistakes.

Technical Mistakes: We do not consider technical and careless mistakes that are due to misuse of the transformation language, a coding error such as a missing handling of *null* values, or comparable mistakes.

Transformation Incorrectness: We do not consider any kinds of mistakes that lead to incorrect transformations. We assume that transformations are correct, i.e., that the consistency preservation rules produce results that are consistent to their consistency relations. Thus any mistake related to the transformations handling changes in only one of the models are out of scope, as these scenarios are part of research regarding the individual bidirectional transformations on their own. However, mistakes regarding synchronization of transformations, i.e., the case that changes were performed to both models, are relevant.

In fact, technical mistakes eventually lead to incorrectness of the transformations.

Transformation Level

Correctness at the transformation level requires each transformation to be synchronizing. We have discussed in Section 6.4 that the essential requirement to make ordinary transformations synchronizing is the matching of existing elements, because transformations that were not developed for the synchronization case do usually not assume elements to be already existing but to be either added by changes that are processed by the transformation or created by the transformation itself.

The mistake a transformation developer can make at this level is not to consider that synchronization is necessary, potentially because he or she does not even know that it is necessary. Then the transformation may be correct but not synchronizing. In the implementation, this manifests as the absence of necessary matchings of elements. We have already discussed that this finally leads to the duplicate creation or insertion of model elements when executing such transformations.

Network Relation Level

The network relation level concerns correctness of the consistency relations in a transformation network. In general, we can distinguish two notions of correctness for them, as discussed in Section 4.2. First, relations must reflect an intended, probably informal notion of consistency. If the relations miss to reflect constraints of that notion or if they reflect additional constraints that are not part of that notion, the relations may be considered incorrect. Second, the relations must be compatible. As discussed in Chapter 5, this is necessary to enable the consistency preservation rules to find consistent models at all. In the worst case, there may not be a single tuple of models that is consistent to all consistency relations when they are incompatible.

The first correctness notion, however, only concerns a single consistency relation rather than the combination of them. We thus assume it to be correct, as we assume each transformation to already be correct. Finally, such incorrectness would not even be interesting. Defining additional constraints does not lead to failures but, in the worst case, only to not finding consistent models although they exist, and missing constraints simply leads to inconsistent models, as the result does not fulfill the constraints of the existing, informal notion of consistency.

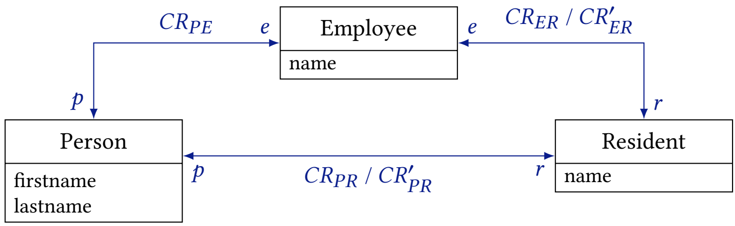
The relevant correctness notion is the one of compatibility. One or more transformation developers can make the mistake of having incompatible knowledge about the consistency constraints encoded into the transformations. This, in consequence, leads to a fault in the implementation of transformations, which may perform a contradicting generation or modification of model elements, for which no orchestration of the transformations may yield consistent models. Depending on the operation of the application algorithm, this can lead to different types of failures. If the transformations are executed with an artificial execution bound, the algorithm will terminate with inconsistent models, which may be returned or not depending on whether it checks consistency. The inconsistency will be deterministic or not, according to whether the execution order of transformations is fixed or not. If the algorithm does not implement such an artificial bound, such a fault can also lead to non-termination of the algorithm, because the execution of transformations will never lead to consistent models. Finally, if the algorithm implements an artificial execution bound and consistency checks, it may also return $\perp$ in this case.

Network Rule Level

The network rule level concerns correctness of the complete transformations of a network. We did not give a precise definition of what this correctness means. In Chapter 7, we have discussed assumptions to transformations to enable an application function to solve the orchestration problem, which could be a reasonable correctness measure. We have, however, also discussed that we cannot make any practical assumptions to the transformations such that they improve the ability of the application algorithm to find a consistent orchestration if it exists.

We only know from Subsection 7.2.4 that consistency relations providing multiple options for corresponding elements to consider models consistent can lead to consistency preservation rules that always select elements that are not in the overlap of these options between different transformations. In consequence, if transformation developers decide to implement consistency preservation rules that make such contradicting selections or generations of elements, the transformations may fail due to the same reasons as discussed for the network relation level. In this case, the causing mistake is that the transformation developers make contradicting selections of available options to restore consistency.

We did not find a property that a transformation set and especially its consistency preservation rules have to fulfill and instead concluded to deal with the orchestration problem by means of a conservative application algorithm. Thus, we cannot give a reasonable or even complete overview of potential mistakes developers can make at this level.



$$CR_{PE} = \{\langle p, e\rangle \mid p.\mathit{firstname} + \text{"}\textvisiblespace\text{"} + p.\mathit{lastname} = e.\mathit{name}\}$$

$$CR_{PR} = \{\langle p, r\rangle \mid p.\mathit{firstname} + \text{"}\textvisiblespace\text{"} + p.\mathit{lastname} = r.\mathit{name}\}$$

$$CR'_{PR} = \{\langle p, r\rangle \mid p.\mathit{lastname} + \text{"}\textvisiblespace\text{"} + p.\mathit{firstname} = r.\mathit{name}\}$$

$$CR_{ER} = \{\langle e, r\rangle \mid e.\mathit{name} = r.\mathit{name}\}$$

$$CR'_{ER} = \{\langle e, r\rangle \mid e.\mathit{name}.\mathit{toLower} = r.\mathit{name}\}$$

Figure 8.2.: Adaptation of consistency relations from the extended running example in Figure 5.1. Adapted from [Kla+19b, Fig. 5].

8.2.4. Causal Chains

We have already discussed the relevant causal chains between mistakes, faults, and failures when introducing the relevant mistake types. The full overview of these dependencies is given in Figure 8.1. Mistakes at the network relation and network rule levels can always lead to any kind of failure, namely non-termination, inconsistent termination, or returning $\bot$, depending on how the application algorithm operates. Thus, these dependencies do not give any insights regarding which mistakes may have caused an occurring failure. Mistakes at the transformation level, however, produce a specific kind of failure that can be distinguished from the general failure types. Thus, knowing these causal chains is especially useful for identifying mistakes at the transformation level. We further discuss the detection and avoidance of mistakes in the subsequent section.

In Figure 8.2, we depict slightly modified consistency relations from the running example. Based on these consistency relations, Figure 8.3 depicts three scenarios of transformation executions with mistakes at each of the three introduced levels. Each scenario assumes a person to be introduced by a user change. Then transformations are executed and produce changes in the order depicted by the numbers at the transformation executions. The

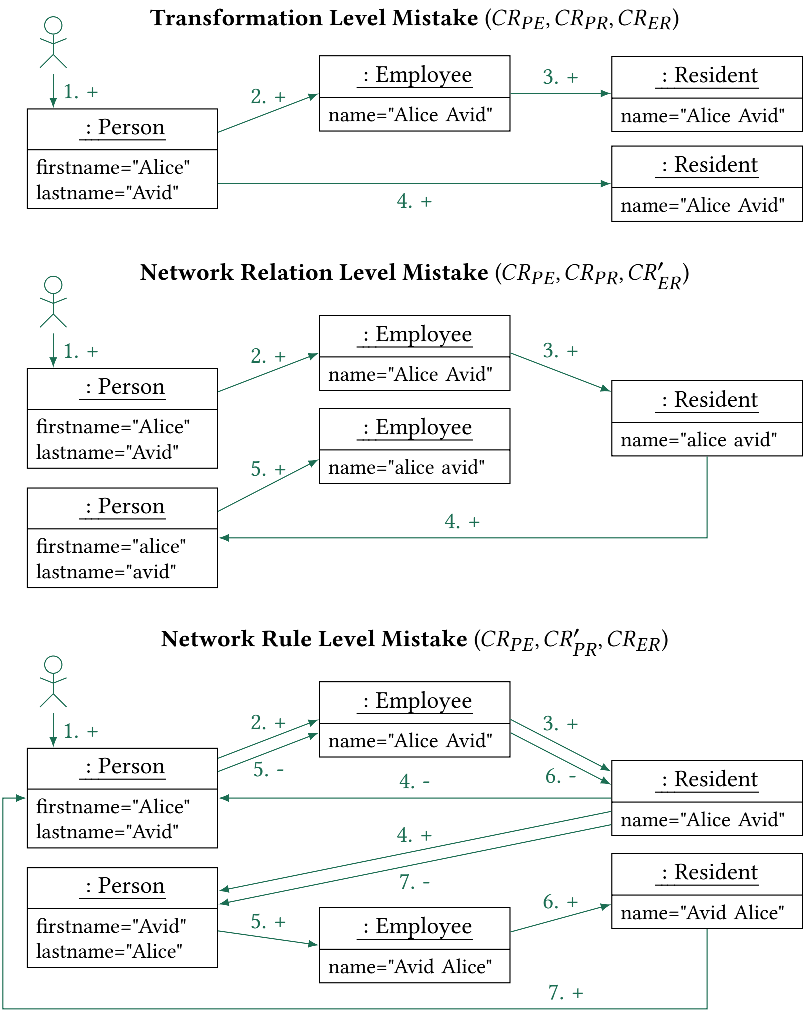


Figure 8.3.: Examples for transformation executions based on the consistency relations given in Figure 8.2 with mistakes at each of the three specification levels. Arrows denote user changes and transformation executions with numbers indicating their order and +/- indicating element addition and removal. Adapted from [Kla+19b, Fig. 5].

creation and deletion of an element is denoted by a "+" and "-", respectively. In one transformation step, multiple elements may be created or deleted. The arrows indicate that the change of the source element leads to the creation or deletion of the target element.

The example for the transformation level considers the compatible consistency relations CR_{PE}, CR_{PR}, and CR_{ER}. It assumes that the transformation developer made the mistake of not considering the necessity of synchronization, thus not implementing a matching of existing elements. This can lead to the depicted failure that two residents with the same *name* may be created by both the transformation between employees and residents, as well as the one between persons and residents. In consequence, the transformations may not be able to process the occurring situation, or, as discussed before, assume consistency by construction and thus identify the models as consistent although they are not.

The example for the network relation level considers the incompatible consistency relations CR_{PE}, CR_{PR}, and CR'_{ER}. Thus, the transformation developers made the mistake of not having a compatible knowledge about consistency constraints. In consequence, the developed transformations may try to resolve the occurring inconsistencies by adding further elements required to fulfill the consistency relations. This results in the depicted models, which are not consistent to CR'_{ER}, because both employees correspond to the resident without the possibility to add a further resident to which one of the employees corresponds. In fact, the transformations would need to remove the initially added person and the first employee to restore consistency. Due to incompatibility, there is no consistent tuple of models containing the initially added person. The algorithm may fail at the depicted state because the transformation between employees and residents is not able to restore consistency.

Finally, the example for the network rule level considers the consistency relations CR_{PE}, CR'_{PR}, and CR_{ER}. The relations require that for each person, employee, and resident, one with swapped *firstname* and *lastname* exists. Whether or not these are reasonable relations, they can be fulfilled by simply adding the appropriate elements. If, however, the transformation developer decides to resolve an inconsistency after adding an element to one model by adding the corresponding one to the other model and removing other elements in the other model for which no corresponding element exists, this leads to the repeated insertion of persons, employees, and residents with *firstname* and *lastname* concatenated in one order and the removal of them with the inverse concatenation, as depicted in Figure 8.3. In fact, the depicted process would proceed after Step 7 from the beginning endlessly, unless the application algorithm stops after a fixed number of transformation executions. In this case, although transformations were developed synchronizing and relations are compatible, finding consistent models after a change fails, because the transformations are not properly aligned with each other. This is analogous to the example depicted in Figure 7.3, for which no orchestration exists. In fact, this is also a problem of selecting incompatible options, as discussed in Subsection 7.2.4, because each transformation always restores consistency in a way that is not consistent with the other transformations, thus selecting an option from the consistency relations that is not in the overlap with consistency relations of the other transformations.

Whether incompatible constraints or a contradicting selection of options to restore consistency leads to a fault and thus potential failures during execution can often not be distinguished. This is especially the case when consistency relations are not explicitly defined but assumed to be implied by the image of the consistency preservation rules. If then the execution of transformations fails because the consistency relations induced by the consistency preservation rules are incompatible, it is unclear whether the, only implicitly known, consistency relations according to which the transformation developer defined the transformations are actually incompatible, or whether the defined transformations only make a contradicting selection of options for restoring consistency, which then implies such incompatible consistency relations. This is due to the reason that even if a transformation developer knows about different options in the consistency relations, he or she can only express one of them in the consistency preservation rules. Thus, the consistency relations implied by consistency preservation rules can always only be a subset of the originally intended consistency relations. For example, when the developers know that two options for name mappings are actually valid and for two transformations they select different of these options, then the consistency relations implied by the implemented consistency preservation rules are actually incompatible, because they contain incompatible name mappings, although in the original knowledge the consistency relations contained both these options, but the consistency preservation rules can only reflect one of them.

Level	Name	Avoidance	Detection
1	Transformation	By construction	Duplicate element creation
2	Network Relation	By analysis	Any network failure
3	Network Rule	-	Any network failure

Table 8.2.: Avoidance and detection of mistakes at the different levels in the transformation network specification process.

8.3. Detection and Avoidance of Errors

Two ways to deal with the possibility of errors in transformation networks exist. First, mistakes can be avoided (a priori), which was the major goal of the discussions and approaches presented in the previous chapters, such that no failures can occur when executing a transformation network or at least failures due to specific mistakes are avoided. Second, mistakes can be detected (a posteriori) by identifying failures during transformation execution. We have already discussed that how a mistakes manifests depends on the used application algorithm. An algorithm without an artificial execution bound may fail by non-termination, one without proper consistency checks may fail by returning inconsistent models, and a conservative algorithm, such as the provenance algorithm proposed in Section 7.4, may return $\perp$.

In Table 8.2, we depict the possibilities of avoiding and detecting mistakes at the different levels in the transformation network specification process. Avoidability is derived from the discussions in the previous chapters, whereas the detection is a result of the preceding categorization of mistakes and resulting failures.

8.3.1. Error Avoidance

In the best case, no failures occur in a transformation network, which means that no mistakes were made at all or at least none of them leads to a failure in a specific scenario. In fact, a network without mistakes does not mean that no failures occur, because the application algorithm can always fail because of undecidability of the orchestration problem. Thus, the absence of failures indicates the absence of mistakes but not vice versa.

To avoid mistakes, we have already discussed different approaches in the previous chapters. Associated with the identified specification levels, we can identify at which levels mistakes can be avoided by construction, by analysis, or not at all. At the transformation level, correctness requires transformations to be synchronizing. As discussed in Chapter 6, this property can be achieved by construction, because it is a property of a single transformation and does not depend on the other transformations to be combined with. We have also proposed techniques, especially the matching of existing elements, to achieve this correctness by construction. At the network relation level, correctness requires consistency relations to be compatible. As discussed in Chapter 5, this property can be validated by analysis of the transformations and their consistency relations. It can, however, not be avoided by construction. Finally, at the network rule level, we do not have a precise notion of correctness, which makes it impossible to define criteria for avoidance.

Since we assume transformations to be developed independently and reused modularly, it is especially relevant that mistakes at the transformation level, for which the required knowledge exists, can be avoided by construction. The necessary knowledge for avoiding mistakes at the network relation level does actually not exist with that assumption, thus we may not even consider them as actual mistakes. Finally, the mistakes that cannot be avoided by construction are handled by the proposed use of a conservative application algorithm anyway. As we have discussed before, consistency checks of transformations may be based on the assumption that consistency is achieved by construction. Thus, it is important that correctness at the transformation level is achieved by construction, as otherwise the application algorithm may apply non-synchronizing transformation without detecting that the yielded models are inconsistent, thus returning inconsistent models.

In Chapter 10, we will discuss how network topologies affect how prone a transformation network is to the possibility of containing faults. We will show that an appropriate topology excludes faults such that transformation developers cannot make mistakes at the network levels. Thus, it is also possible to avoid such mistakes by construction, but this limits the networks we can define to specific topologies. We also discuss in Chapter 11 an approach to construct networks of such a topology, which mitigates the restrictions induced by the necessity of having a specific topology by introducing auxiliary models.

8.3.2. Error Detection

Whenever mistakes are not avoided by construction or analysis, they can be detected by failures of the application algorithm. The insights regarding relations between mistake and failure types may at first not sound interesting, because all mistakes at the two network levels can lead to any kind of failure. And even if a duplication occurs, which is in particular the result of a mistake at the transformation level, this can also be a consequence of a mistake at the two network levels. Additionally, the algorithm may not only fail because of mistakes but also because of undecidability of the orchestration problem. Still, we can make some relevant conclusions for the detection of errors.

Insights about the causing mistakes can especially be derived from an inconsistent state of the models that the algorithm produced, e.g., by investigating whether this inconsistent state contains duplications of elements. This is why we proposed the provenance algorithm in Section 7.4, which is supposed to support the identification of problems in the transformations that lead to the application algorithm not being able to find a consistent orchestration. Thus, in case the algorithm fails for specific inputs, it is up to the transformation developer to investigate the state of the models in which the algorithm failed to identify the reason for that.

Whenever the application algorithm fails, it can be useful to exchange it with one with different properties. If the algorithm does not terminate, introducing an artificial execution bound can produce an insightful inconsistent state of the models. These inconsistent models can also be retrieved from a conservative algorithm as proposed in Section 7.4.

The occurrence of duplications is a specific indicator for missing synchronization. They can occur in inconsistent returned models produced by the algorithm and will most likely occur because of missing synchronization. In our evaluation in Chapter 9, we will see that in the investigated case study duplications occurred because of missing synchronization in most cases or can at least be distinguished from duplications caused by other mistakes.

If the algorithm fails for most inputs in any way, this may be an indicator that the algorithm is not only unable to yield consistent models because of the orchestration problem but because some essential mistakes prevent it from from finding consistent models, such that, in the worst case, no consistent orchestration exists at all. Thus, an often failing algorithm may be an indicator for, among others, incompatibilities.

It may make a difference whether a conservative algorithm fails returning $\perp$ because the maximal number of executions was reached or because a transformation could not be applied anymore. While the inability to apply a transformation can be seen as an indicator for an actual mistake within the transformations (such as the network relation level error in Figure 8.3), the abortion because of reaching the execution bound can also just result from the conservative behavior to avoid non-termination.

Finally, in the best case errors are avoided by construction, especially potential mistakes at the transformation level. At the network levels, mistakes cannot be avoided but, in the best case, analyzed. Since we need a conservative application algorithm anyway, it also ensures that such mistakes do not lead to unwanted results. In the worst case, the algorithm will only be able to yield consistent models in few or even no cases. Then the transformation developer must investigate the state of the models with which the algorithm fails to identify the reasons. Although there are several indicators for the existence of faults, it cannot be uniquely distinguished whether the application algorithm fails because of undecidability of the orchestration problem or because actually the transformations contain a fault. Since we assume independent development and reuse of transformations, the focus on avoiding mistakes at the transformation level and the handling of mistakes at the network levels by a conservative algorithm fits well to that context assumption.

8.4. Summary

In this chapter, we have discussed the separation of the transformation network specification process into three levels, we have categorized the possible mistakes, faults, and failures that can occur in such a network, and we have discussed which of them can be avoided or detected. We have considered the avoidance and detection of errors at a rather conceptual level, emphasizing what a transformation developer has to do to achieve correctness by construction and what he or she has to do if a transformation fails. We did, however, not propose a concrete process for the resolution of errors when they occur in a productive environment. This involves a system developer, who uses the transformations to keep models consistent and faces failing transformations, as well as the transformation developer, who is responsible for correcting potential faults in the implementation. Such a process discussion is out of the scope of this thesis and referred to as future work (see Subsection 9.3.4). We conclude this chapter with the following central insight.

Insight II.5 (Errors)

Errors in transformation networks can be classified regarding mistakes made by the transformation developers when thinking about consistency and its preservation, faults made during their implementation in transformations, and failures, which are the manifestation of faults when executing the transformations. We found that we can assign different kinds of mistakes to three different conceptual levels in the specification process, depending on the necessary knowledge about the transformation network. We derived that mistakes regarding a single transformation cover missing synchronization, which can and has to be avoided by construction. This is particularly necessary if transformations assume consistency to be achieved by construction, because then non-synchronizing transformations produce faulty results that they assume to be consistent. All other types of mistakes concern the network of transformations, either restricted to the relations or also concerning the consistency preservation rules. While consistency relations can at least be analyzed for compatibility, further mistakes cannot be avoided but only be detected by the application algorithm failing in specific scenarios. Due to the assumption of independent transformation development and modular reuse, it fits well that a conservative application algorithm is necessary anyway and also covers mistakes concerned with the network of transformations. Only if the transformation network fails in many scenarios, e.g., because of transformations with incompatible consistency relations, the transformation developers need to investigate the reasons for the algorithm to fail.

9. Evaluation and Discussion

In the preceding chapters 4–8, we have discussed several aspects of a well-defined notion of consistency and correctness of its preservation in transformation networks. Based on the assumptions we made, we were able to prove several statements regarding decidability of problems, correctness, and the properties and effects of approaches we proposed, such as the analysis of compatibility or the construction of synchronizing transformations. Thus, several insights presented so far have been validated by proof. Still, there are several interesting and relevant questions that we will validate by empirical evaluation at case studies. These especially concern the applicability of our approaches and also, at least implicitly, the appropriateness of our formalism, which we evaluate in case studies.

We do not provide an evaluation of the consistency and correctness notions proposed in Chapter 4. That formal foundation was derived from our motivation and assumptions by argumentation. Thus, a meaningful evaluation would be a user study in which the reasonability of the assumptions we made regarding the process of defining consistency in transformations networks is validated. Since we have based our work on well-motivated assumptions and since such an evaluation would be overly complex, we have decided not to perform it as part of this thesis and focus on statements that we can derive from the assumptions in Chapters 5–8.

The compatibility notion and the formal approach to validate consistency relations for compatibility that we have proposed in Chapter 5 is proven correct. The practical approach was derived from the formal one such that it is also supposed to be correct, although this is not formally proven. We apply the approach to a case study of several sets of consistency relations to first evaluate correctness, which especially concerns correctness of the implementation but also validates the construction of the practical out of the formal approach. Second, we evaluate applicability in terms of the degree of conservativeness, i.e., how often the approach does not prove compatibility although compatibility is given.

The properties of a bidirectional transformation to be synchronizing were proven to be correct in Chapter 6. The approach to achieve these properties was, however, derived by argumentation. In a second case study, we thus combine existing transformations, which were not supposed to be used in a transformation network and thus are neither synchronizing nor fulfill other correctness notions of transformation networks. We use this case study to evaluate completeness and correctness of the categorization of errors presented in Chapter 8 and also identify the relevance of the different mistake types regarding how often they occur and thus how prone they are to be made by transformation developers. We also evaluate practical relevance of the orchestration problem by investigating how often the orchestration fails because of that problem instead of actual mistakes in the transformations. Additionally, we apply our approach for making ordinary transformation synchronizing, depicted in Chapter 6, regarding correctness, i.e., whether it actually resolves failures due to transformations not being synchronizing. We validate its applicability regarding whether it is able to resolve all faults due to missing synchronizing. We will especially find that transformations not being synchronizing is the most relevant mistake type, that most other mistakes are due to incompatibilities, and that, at least in the considered case studies, the orchestration problem is not practically relevant. Finally, our approach for achieving synchronization of ordinary transformations is able to resolve most of the issues, at least in the considered case studies.

Finally, we haven proven several statements regarding the orchestration of transformations in Chapter 7, especially the undecidability of the orchestration problem. We have also proven correctness of the proposed conservative application algorithm. The fulfillment of the motivational property of the algorithm to support the process of finding errors when the algorithm fails to find consistent models is, however, only argued. We thus provide a scenario-based discussion to evaluate the usefulness of the strategy.

For each of these topics, we provide a plan according to the Goal Question Metric (GQM) approach, for which the original idea was presented by Basili et al. [BW84]. We define goals that we want to achieve with our evaluation, derive questions that we answer to identify whether we have achieved the underlying goal, and define metrics whose results we use to get a quantitative measure for answering the questions.

We have published parts of these evaluations in previous work [Kla+19b; Kla+20; GKB21]. The case studies for our error categorization and achievement of synchronization have been conducted in two Master's theses [Sym18; Sağ20]. The case study for validating the approach to prove compatibility has been conducted in another Master's thesis [Pep19]. We will explicitly refer to the according publications in the individual evaluations.

9.1. Compatibility

In Chapter 5, we have presented a formal notion of compatibility, a formal approach to prove it, and a practical realization of this approach for consistency relations defined in QVT-R. The compatibility notion is well-defined, based on our formalization of transformation networks and a correctness notion for them. The formal approach to validate compatibility of consistency relations of a transformation network is based on the insights that specific consistency relation trees are inherently compatible and that the addition and removal of consistency relations fulfilling a specific notion of redundancy preserve compatibility, thus removing redundant relations until a tree remains validates compatibility. We have proven correctness of this formal approach with Theorem 5.11, Theorem 5.6, and Corollary 5.12, such that we do not need to evaluate it. We thus focus on correctness of the practical realization of the approach as well as its applicability. The presented evaluation is based on and in parts taken from the evaluation that we presented in previous work [Kla+20] and that was developed in the Master's thesis of Pepin [Pep19].

9.1.1. Goals and Methodology

A tool for proving compatibility could be easily integrated into the process of developing a transformation network in order to assist transformation developers, as it operates fully automated and thus introduces no further developer effort, and it improves the ability of the transformation network to find consistent models after changes. Thus, the correctness and the applicability of the approach are of particular importance.

In the subsequently presented empirical evaluation in terms of a case study, we apply the practical realization of the approach to several sets of consistency relations, which are designed to be compatible or not according to Definition 5.3. We then apply the algorithm to prove compatibility to these

Goal 1: (Compatibility)	Show that the analysis can be used by transformation developers to find incompatibilities in consistency relations of a transformation network.
Question 1.1: (Correctness)	Is compatibility always given if the analysis finds it?
Metric 1.1.1:	*Precision: Ratio between true positives and the sum of true and false positives*
Question 1.2: (Applicability)	How often does the analysis not prove compatibility although it is given?
Metric 1.2.1:	*Recall: Ratio between true positives and the sum of true positives and false negatives*

Table 9.1.: Goals, questions, and metrics for compatibility evaluation.

consistency relation sets and analyze whether it properly identifies them to be compatible or not. We denote the cases in which the algorithm proves compatibility as *positives* and the ones in which it is not able to do so as *negatives*. Since the algorithm operates conservatively, a negative result does not mean that incompatibility is proven but only that compatibility could not be proven. The goal of this evaluation, the answered questions, and the evaluated metrics are summarized in Table 9.1.

First, the application of the algorithm to multiple scenarios allows us to validate correctness of the practical realization of the approach according to Question 1.1. Correctness of our approach means that it is able to classify a given set of consistency relations as compatible or otherwise does not reveal a result. This especially means that it operates conservatively and does not classify a set of consistency relations as compatible although it is not. The algorithm is thus not allowed to produce false positives, which is why we consider the *precision* metric:

$$precision = \frac{true\ positives}{true\ positives + false\ positives}$$

This metric needs to be 1, as otherwise the algorithm produces false positives and would be incorrect per definition. In consequence, correctness of the algorithm directly correlates with this metric. Analyzing this metric serves as an indicator that the mapping of our formal approach and the underlying formalism to the practical approach realization and the used QVT-R language is correct, and especially that it operates conservatively.

Second, the application of the algorithm to multiple scenarios allows us to validate its applicability according to Question 1.2. The approach uses a fully automated algorithm, thus it does not require any inputs apart from the QVT-R relations to check. Applicability may thus be restricted if the algorithm operates too conservatively, i.e., if it produces false negatives too often. In those cases, the algorithm operates actually correctly, but if it was not able to prove compatibility in most cases in which it is actually given, applicability would be reduced, as the usefulness of the results for a transformation developer is limited. For that reason, we analyze the *recall* metric:

$$recall = \frac{true\ positives}{true\ positives + false\ negatives}$$

The higher the number of false positives, the more consistency relations could not be identified as compatible by the algorithm although they actually are, thus reducing the usefulness of the algorithm. In consequence, applicability of the algorithm directly correlates with the recall metric. For that reason, we analyze this metric and the reasons for the cases in which the algorithm was not able to prove compatibility, i.e., in which it produced false negatives. In particular, it is relevant whether they are caused by conceptual issues of the formal approach, such as a too restricted notion of redundancy, or a limitation of the practical approach that may be fixed by a different implementation or a different realization approach.

9.1.2. Prototypical Implementation

The approach that we have presented in Section 5.4 resulted in the implementation of a prototype, which is available in a GitHub repository [GitDec]. The prototypical implementation is specific to QVT-R and OCL expressions used in that language. It expects a set of QVT-R transformations and returns a list of redundant QVT-R relations. Thus, if removing the returned redundant relations from the initial set of transformations yields a set of transformations whose relations do not contain any cycles, i.e., if they form a consistency relation tree, compatibility is proven. If cycles within the relations remain, compatibility could not be proven either because of an actual incompatibility or because of the algorithm not being able to find redundancies to prove compatibility.

Additionally, the implementation validates the given inputs. They may be invalid because of two reasons. First, they can contain transformations that are not well-formed, i.e., they are syntactically incorrect. In that case, the transformation cannot be processed by the compatibility analysis algorithm at all. Second, transformations can be well-formed but invalid, e.g., because two transformations have the same name or a QVT-R domain pattern uses a nonexistent class. Although the algorithm can still be applied to such an input, it may not produce appropriate results, which is why such errors are displayed to the transformation developer when applying the algorithm in the parsing step. Some errors, such as two transformations having the same name, could even be mitigated by automatically renaming them if such a clash occurs. In the evaluation, we, however, only consider valid inputs anyway. Finally, the implementation operates non-intrusively, thus not altering the transformations in any way.

The selection of QVT-R for the practical realization and implementation of the approach was, on the one hand, driven by the recommendation of the MDA [MDA] to use QVT-R for defining transformations, and, on the other hand, by the fact that consistency relations are explicitly defined in QVT-R, especially in comparison to imperative languages. We have based the implementation on the EMF and its Ecore meta-metamodel (see Subsection 2.2.2) as one of the most common and technically mature modeling frameworks. Within the EMF, implementations of transformation languages are provided through the *Eclipse MMT* [EcMMT] project. In particular, the contained QVT Declarative (QVTd) [EcQVT] language provides a parser for QVT-R transformations, which, in turn, uses *Eclipse OCL* [EcOCL] as an implementation of OCL.

For finding redundant relations, their OCL constraints are transformed into logic formulae, whose satisfiability is then to be validated by an SMT solver. Many such solvers are based on SMT-LIB [BFT17], which is an initiative that provides a common input and output language for SMT solvers. Our prototype uses the Z3 theorem prover [dB08], which is an SMT solver that can be used in Java code and supports a large number of theories.

#	Scenario Description	Compatible
1	Three equal string attributes of three metamodels	yes
2	Six equal string attributes of three metamodels	yes
3	Concatenation of two string attributes	yes
4	Double concatenation of four string attributes	yes
5	Substring in a string attribute	yes
6	Substring in a string attribute with precondition	yes
7	Precondition with all primitive data types	yes
8	Absolute value of integer attribute with precondition	yes
9	Transitive equality for three integer attributes	yes
10	Inequalities for three integer attributes	yes
11	Contradictory equations for three integer attributes	no
12	Contradictory inequalities for three integer attributes	no
13	Constant property template items	yes
14	Linear equations with three integer attributes	yes
15	Contradictory linear equations for three int. attributes	no
16	Emptiness of various OCL sequence and set literals	no
17	Equal string attributes for four metamodels	yes
18	Transitive inclusions in sequences	yes
19	Comparison of role names in three metamodels	yes

Table 9.2.: Consistency relation scenarios and their compatibility. Taken from [Kla+20, Tab. 3].

9.1.3. Case Study

We have applied our prototypical implementation in a case study to 19 scenarios, which are also available at GitHub [GitDec]. Each of these scenarios consists of three or four metamodels and comprises especially primitive data types and operations. They contain pairwise transformations between the metamodels defined in QVT-R, more specifically its implementation QVTd.

The scenarios are listed in Table 9.2. It also depicts whether the relations of the transformations in these scenarios are compatible or not. In total, 15 of these scenarios contain compatible consistency relations according to Definition 5.3, whereas the other four are incompatible. Thus, we know for each of the scenarios by construction whether it is compatible or not, which constitutes the ground truth for our evaluations. The application of the prototypical implementation to these scenarios yields the results *positive* if it considers the relations compatible, or *negative* if it was not able to prove compatibility. Comparing these results with the ground truth in Table 9.2 allows us to identify them as true or false positives or negatives.

The scenarios were specifically developed for the evaluation of the approach, thus reflecting as many kinds of relations as possible that can be expressed with QVT-R and also reflecting edge cases. The implemented QVT-R relations used for the case study are also available in the GitHub repository containing the prototypical implementation [GitDec].

9.1.4. Results and Interpretation

We have applied the prototypical implementation of our practical approach introduced in Subsection 9.1.2 to the case study explained in Subsection 9.1.3. The results of the scenario classification as compatible or not by the implementation are summarized in Table 9.3.

9.1.4.1. Correctness

Correctness for the formal approach has been proven. Since the practical approach is derived from this formal approach, correctness is also given by construction as long as the following requirements are fulfilled.

1. All relevant QVT-R relations are considered as consistency relations to be checked, i.e., all relations are represented in the property graph.
2. All constructs referring to expressions in QVT-R relations have to be considered. QVT-R relations are defined using variables, so all constructs referring to these variables have to be considered. In particular, all template expressions need to be represented, namely property template items, preconditions, and invariants.

The construction of the approach presented in Section 5.4 ensures that these relevant elements are considered. Additionally, the results of the case study further validate that we did not miss any relevant parts of QVT-R relations.

The results depicted in Table 9.3 show that the implementation did not yield any false positives. Thus, the implementation operates conservatively as

	Classified Compatible	Unclassified
Compatible	12	3
Incompatible	0	4

Table 9.3.: Compatibility classification of scenarios depicted in Table 9.2 by our approach. Corrected from [Kla+20, Tab. 4].

intended and does not identify consistency relations as compatible although they are not. This results in a precision value of 1:

$$precision = \frac{true\ positives}{true\ positives + false\ positives} = \frac{12}{12 + 0} = 1$$

On the one hand, this indicates that the practical approach actually conforms to the formal approach, so that the correctness proof applies as well. On the other hand, this indicates that the implementation is correct and does not miss any relevant QVT-R constructs. If this were the case, constraints would have been missed, which could have resulted in identifying consistency relations as compatible although they are not. Thus, as an answer to Question 1.1, the results indicate that we can expect the analysis to operate correctly.

9.1.4.2. Applicability

We have discussed that applicability of the approach especially depends on how often it fails in terms of not proving compatibility although the given consistency relations are compatible. In particular, conservative behavior of the approach can occur for the following two reasons.

Redundancy Notion: Compatibility of consistency relations is proven by identifying relations that follow the definition of left-equal redundancy (see Definition 5.9). Since this redundancy notion is not the weakest one that is compatibility-preserving, it may be a too strong requirement for identifying compatibility-preserving consistency relations.

Redundancy Undecidability: Definition 4.18 for consistency relations relies on an extensional specification of consistency, which enumerates usually infinite sets of elements. Since such sets cannot be compared programmatically, our practical approach relies on intensional specifications in

OCL as used by QVT-R, which describe how consistent element pairs can be derived. OCL is, however, in general undecidable, because it can be transformed into first-order logic [BKS02].

In particular, the higher the number of quantifiers within a formula, the more likely its satisfiability will be undecidable. Since variables in consistency relations are translated to existentially quantified formulae, the number of variables in a consistency relation is crucial for deciding satisfiability. Not all available OCL constructs may be necessary to describe relevant consistency relations, still constructs involving operations on collections, which are transformed into quantified formulae, and strings are especially problematic. For example, *toUpper* and *toLower*, which we have also used in our running example, cannot be transformed into formulae for state-of-the-art SMT solvers like Z3 and thus cannot be considered for detecting redundancies. Additionally, SMT solvers use heuristics, which prevents a systematic evaluation of the kinds of consistency relations that can be analyzed by the approach.

According to the results in Table 9.3 from applying our prototypical implementation to the scenarios introduced in Table 9.2, consistency relations were correctly classified as compatible in twelve out of the 15 scenarios, whereas the implementation was not able to prove compatibility in the remaining three scenarios, thus delivering three false negatives. This leads to a recall value of 80 %.

$$recall = \frac{\textit{true positives}}{\textit{true positives} + \textit{false negatives}} = \frac{12}{12+3} = 0.8$$

This is a first indicator for high applicability of the approach, as it could prove compatibility in most of the cases in which the relations were compatible.

The Scenarios 8, 18, and 19 introduced in Table 9.2 were not identified as compatible although they actually are. In all cases, the SMT solver should have returned *unsatisfiable* but instead returned *unknown*. In each scenario an actually redundant consistency relation was not removed, thus not identifying the relations as compatible. In detail, in Scenario 8 a precondition ensures that an element is included in the intersection of two set literals, but the solver was not able to check that properly. In Scenario 18, the transitive inclusion of sets was defined, and in Scenario 19, role names of classes with equivalent identifiers were considered, which the solver was both not able to check properly as well. In summary, all observed false negatives were caused by undecidability of satisfiability of the first-order formulae that were derived from the OCL constructs.

In conclusion, the evaluation has shown that basic operations on primitive data types, even with non-trivial constraints involving integer equations and string operations, were treated correctly. This led to a success rate of 80 %. As an answer to Question 1.2, the approach was unable to prove compatibility in only 20 % of the cases, in which more complex operations and structures requiring many quantifiers were involved, for which satisfiability could not be proven by the used SMT solver. Most importantly, however, this limitation only concerns the chosen SMT solver approach but neither the general concept of the formal framework and approach nor the practical realization itself. In particular, we did not find a scenario in which our redundancy notion was too strict for proving compatibility. Using a different SMT solver or, more generally, even a different approach to validate redundancy of consistency relations can even improve the applicability results.

9.1.5. Discussion and Validity

The evaluation of our compatibility analysis approach has shown that in the scenarios considered in the case study it operates correctly and shows a low degree of conservativeness, i.e., it is able to validate compatibility in most cases. This indicates correctness and high applicability of the approach. Still, there are some threats to the validity of these results, which we discuss after general conclusions on the benefits of the proposed approach.

9.1.5.1. Benefits

In general, the approach is supposed to support transformation developers in designing transformation networks by checking compatibility of transformations during their individual development or when combining them to a network. With the example depicted in Figure 5.6, we have shown that incompatible consistency relations can prevent the transformations from finding consistent models. Thus, incompatibilities eventually lead to failing executions of transformation networks, which, in turn, require transformation developers to find the reasons for that. Our approach provides a benefit by preventing such issues or at least by supporting the developer in finding their causes when running the analysis after a failure occurs. Due to its full automation, it requires no further effort than running the analysis. Additionally, a manual process of ensuring compatibility or finding incompatibilities requires manual alignment of transformations or the definition of test cases, which only validate but do not verify compatibility. Thus, such manual techniques can only make existentially quantified statements about the existence of incompatibilities, whereas our approach makes universally quantified statements about their absence.

Finally, even if the proposed approach had a high degree of conservativeness, i.e., if it produced a higher number of false negatives in other scenarios than in our evaluation, the approach still provides benefits. First, the approach would still be able to prove compatibility at least in few cases. Second, even if the approach cannot prove compatibility, it may at least detect some redundant relations and thus reduces the effort for the transformation developer to find incompatible relations. It would even be possible to define an interactive approach in which the removal of redundant relations by proof and by user decision is combined, which we propose as future work in the subsequent section. In such a process, the user could be asked to manually declare redundant relations when the automated approach does not find further ones. Afterwards, the automated approach can proceed.

9.1.5.2. Threats to Validity

We have designed the evaluation carefully, such that it gives appropriate insights regarding correctness and applicability of the approach. Still, due to limited complexity of the considered scenarios, threats especially regarding external validity of the results exist.

The evaluation scenarios of the case study were developed specifically for the evaluation of the approach. Thus, they may potentially not sufficiently represent actual transformation networks. On the other hand, the scenarios were designed to test different aspects of the approach and thus represent an extensive set of consistency relations and also consider edge cases. Scenarios not developed for the evaluation may not or only rarely cover specific and edge cases. In fact, most meaningful results could potentially be achieved with a combination of externally developed scenarios and evaluation-specific scenarios. However, the limited availability of scenarios, especially of scenarios developed with the tools we have used for the prototype and contain incompatibilities, prevents this.

The defined scenarios only contain OCL constructs that the approach currently supports. Thus, unsupported constructs are not covered by the evaluation, which may be a bias. The algorithm would, however, not yield a result in such scenarios anyway, thus this would not give further insights. Additionally, this is only a limitation of the implementation and not a conceptual limitation of the approach. The actual threat is that more complex relations, which are currently not supported by the implementation, may not be covered by our definition of redundancy. That would be an actual limitation also of the formal approach. In consequence, this has to be further evaluated in subsequent evaluations.

The considered scenarios only contain up to four metamodels with pairwise consistency relations. Actual transformation networks will probably contain more and larger metamodels and consistency relations. This is, however, not a threat to validity regarding correctness, because the inductive definition of the approach makes it independent from the number of metamodels and relations to consider. It may only affect applicability, as increasing size may lead to logic formulae which the SMT solver is not able to resolve. The size of scenarios may especially affect the performance and scalability of the approach, which we did not analyze in our evaluation and discuss in the subsequent limitations.

In consequence, our evaluation gives an initial indicator for the correctness and applicability of our approach based on well-selected evaluation scenarios but is potentially restricted in external validity due to the limited set and complexity of scenarios. To improve evidence in external validity, applying the approach to further and larger transformation networks would be beneficial. However, acquiring such networks is difficult. Especially, transformations in existing networks can be expected to be aligned with each other, thus not containing incompatibilities and limiting the evaluation to positive cases. A possibility to reduce that problem would be the manual extension of such networks by adding transformations with redundant or incompatible consistency relations. This would directly deliver a ground truth against which the results of the approach on these modified networks can be validated.

9.1.6. Limitations and Future Work

We discuss two types of limitations of our approach. First, we consider limitations of the current state of implementation. Second, we discuss limitations of the current state of evaluation, which may have masked limitations of the current concept. In addition, we discuss the opportunities for future work that these limitations as well as the conceptual core of the idea to prove compatibility and processes to use it provide.

Practical Approach Realization The proposed practical approach for QVT-R has fundamental as well as technical limitations. First, SMT solvers are limited such that they cannot analyze all kinds of formulae regarding satisfiability. Thus, even if we can transform all kinds of QVT-R and OCL constructs into logic formulae, they cannot necessarily be checked for satisfiability, as we have shown in the applicability evaluation. Second, we do not yet support all kinds of QVT-R relations, as we do not yet provide a transformation for all kinds of OCL constructs into logic formulae. This is, however, only a technical limitation that can be solved by additional implementation effort.

In future work, we will thus extend the operations for which translations to logic formulae are defined, so that we can apply the approach to more sophisticated case studies. This will provide further indicators for the general applicability of the approach. In addition, we will consider alternative realizations of the approach that circumvent the limitations of SMT solvers in general. The limitation of cases that a theorem prover can analyze can restrict applicability of our approach, and in the scenarios considered in our evaluation in Section 9.1, it was even the only limitation regarding applicability. To circumvent or mitigate this limitation, it is possible to implement the approach in Section 5.4 by means of other formal methods. For example, interactive theorem provers can potentially prove redundancy of consistency relations in more cases. Another possibility is the use of multiple formal methods next to SMT solvers, as some formal methods can provide proofs in cases in which others can not. Although this improves the effort for developing the translations, the simultaneous use of different symbolic computation tools can increase the chance of finding redundancy proofs. Additionally, it may even be beneficial to simplify the OCL statements transformed into logic formulae where possible, like discussed by Cuadrado [Cua19]. On the one hand, this can improve the chance of success of the SMT solver. On the other hand, it can make it easier for a transformation developer to understand the reasons why the algorithm failed if the checked expressions are simpler.

Benefits Evaluation and Development Process We have not provided an evaluation for the benefits that we claim for our approach. First, to the best of our knowledge, there are no competitive approaches to compare our one with. Second, it automates a manual process without requiring additional effort, thus compared to the baseline of performing the process manually, it provides an inherent and essential benefit. Thus, further empirical evaluation in a user study could only provide a quantitative measure of the benefits rather than the qualitative one we gave by argumentation. Such an evaluation could especially consider a development process in which the approach is used and evaluate whether that whole process improves by using our approach.

Such a process specification and evaluation should be part of future work. Our approach is only able to prove compatibility but not to prove incompatibility. If the approach does not identify a network as compatible, it may be incompatible or not. For that reason, we aim to define a holistic process for applying the approach, which integrates further information given by the user into the process of proving compatibility. Since the approach operates inductively, it can simply allow the transformation developer to perform single induction steps. If the algorithm is not able to prove compatibility, i.e., if it does not find further redundant relations, it can present the network, in which the algorithm already removed some redundant relations, to the transformation developer. He or she is then asked to declare a cycle of consistency relations as compatible, for which the algorithm is not able to prove it or which are even not compatible intentionally. Afterwards, the algorithm could proceed with finding further redundant relations to prove compatibility, based on the decision of the user. As a result, the approach would be applicable to more scenarios in which compatibility is intentionally not given or in which the algorithm on its own is not able to prove it.

Compatibility Notion and its Effects The notion of compatibility was derived from the goal of finding contradictory consistency relations that can prevent transformations from finding consistent models after changes. Additionally, it prevents the specification of contradictory and thus unintended consistency relations. Although we have shown at examples that our notion of compatibility fulfills both these notions, it is unclear whether this notion is kind of optimal in the sense that there exists no other notion that covers even more unwanted cases.

Evaluating the central purpose of the approach to improve the ability of transformations to find consistent models, i.e., to improve dealing with the orchestration problem, is part of our future work. In fact, compatibility ensures that the ability of not finding a consistent orchestration due to the orchestration problem decreases, thus reducing the ability that transformation networks fail or do not terminate. While we have shown this at examples in this work, we will empirically evaluate in future work how compatibility affects the ability of transformation networks to find consistent models and, if possible, even formally prove and analyze that effect.

Relaxation of Redundancy Notion We have defined the notion of left-equal redundancy (see Definition 5.9), which is proven to preserve compatibility. It is, however, unclear whether a more relaxed notion of redundancy exists that is still compatibility-preserving. Our implementation follows an even stricter notion of redundancy and still no limitations of applicability occurred in the case study. If, however, other case studies reveal the necessity of a weaker redundancy notion to be able to prove compatibility in more cases, either the notion used in the implementation needs to be relaxed or even the formal foundation needs to be adapted. Thus, we still aim to find the weakest possible notion of redundancy that is still compatibility-preserving, if it exists, in future work. This especially involves finding scenarios in which our notion of left-equal redundancy is too restrictive.

Performance and Scalability We have neither measured nor formally evaluated the performance and scalability of our approach and especially its practical realization. Applicability may be affected if the approach required too much time to be executed. SMT solvers, such as the used Z3 solver, depend on heuristics, which makes their performance hardly predictable. Thus, it would be important to evaluate performance of the approach in a case study. In our case study, we did not observe any time-consuming scenarios. However, transformation networks with more and larger transformations and especially many cycles of consistency relations need to be investigated to make generalizable statements on the performance and especially the scalability of the approach. Since the approach is applied as an offline analysis, which does not require instant feedback, it does not have to fulfill real-time requirements. Results should, however, still be delivered in an acceptable amount of time to achieve acceptance of the approach.

9.2. Errors, Orchestration and Synchronization

In Chapter 8, we have presented and discussed a categorization of errors in transformation networks. Such errors can occur when different kinds of mistakes are made when developing transformation networks, especially involving missing synchronization of the individual transformations as discussed in Chapter 6, but also because an algorithm that applies the transformations is not able to find consistent models because of the orchestration problem as discussed in Chapter 7.

We empirically evaluate different aspects of errors, their categorization, and their avoidability as well as resolvability by the proposed approaches in a case study. In that case study, we utilize a set of independently developed transformations, which were not supposed to be used in a transformation network. In consequence, executing them in a network leads to several failures. We analyze these failures and their causes to improve evidence of correctness and completeness of our categorization and to make statements about the relevance of the different failures and causing mistakes by their numbers of occurrences. Additionally, we apply our proposed approach for developing synchronizing transformations to resolve the failures to evaluate the correctness and applicability of that approach.

Since the orchestration problem can always lead to the situation that an application algorithm for a transformation network cannot find consistent models, we also utilize this case study to investigate how problematic the orchestration problem actually is in practice. We know from the halting problem that undecidability of an essential problem in software engineering does not have to be that relevant in practice.

9.2.1. Goals and Methodology

To evaluate both our proposed categorization of errors as well as our presented approach to avoid or find errors, we have conducted two case studies in which we combined existing transformations, of which two were not developed to be used in transformation networks, whereas one was designed to be synchronizing to be used in networks. In consequence, their combination revealed several errors to evaluate our categorization with, and by applying our approaches for constructing correct transformation networks, we were able to evaluate the approach for synchronizing transformation construction and the relevance of the orchestration problem as a source of errors.

The general process we followed in these case studies looks as follows. We combined independently developed transformations and executed existing test cases developed for the individual transformations, which we extended by validations of the further models generated by the additional transformations. We then validated the failures occurring in the test case execution. We used the information about the failures to trace back to the causing faults and mistakes, such as missing matchings of elements when multiple instantiations occur. For each identified failure, we fixed the causing fault and re-executed the test cases to validate whether the failure was resolved by the fix.

The process was applied iteratively until no more failures occurred. Since failures due to one mistake can hide failures caused by another, it was possible that after fixing all faults that led to the failures in one iteration still failures occurred afterwards. For example, incompatible consistency relations may not lead to any failure because the scenario fails earlier due to missing element matchings. Then, after adding the element matchings, the scenario may still fail, now because of the incompatible consistency relation. We explain in more detail which transformations we combined in which order in the subsequent section about the case studies. In the following, we discuss which evaluation goals we aimed to achieve with this process and which metrics we employed to answer different questions for achieving these goals.

9.2.1.1. Categorization and Orchestration

For the evaluation of our error categorization and the relevance of the orchestration problem, we depict the evaluation plan in Table 9.4. We evaluate completeness of the categorization in Question 2.1, i.e., that we did not miss any relevant mistakes in the categorization. This is covered by measuring how many occurring failures could be classified, i.e., traced back to mistakes they were caused by according to the categorization. The following according metric relates the number of classified to the number of totally identified failures, thus indicating a higher degree of completeness with a higher value with a maximum of 1:

$$\textit{classified failure ratio} = \frac{\#\ \textit{of classified failures}}{\#\ \textit{of total failures}}$$

Goal 2: (Categorization)	Show that the categorization of mistakes, faults and failures covers all relevant cases and identify relevance of the individual mistake types.
Question 2.1: (Completeness)	Can all failures be traced back to mistakes according to the categorization?
Metric 2.1.1:	*Classified failure ratio: Ratio between classified failures and identified failures*
Question 2.2: (Correctness)	Are identified failures caused by mistakes to which they are related according to the categorization?
Metric 2.2.1:	*Resolved failure ratio: Ratio between resolved failures and total number of failures*
Question 2.3: (Relevance)	How relevant is each type of mistake, i.e., how likely is it to be made?
Metric 2.3.1:	*Mistake type occurrence ratio: Ratio between occurrences of faults due to each type of mistake and total occurrences of faults*
Goal 3: (Orchestration)	Determine how relevant undecidability of the orchestration problem is in practice.
Question 3.1: (Relevance)	How often does an algorithm for orchestration fail due to the orchestration problem?
Metric 3.1.1:	*Fail ratio: Ratio between algorithm failures due to the orchestration problem and all failures*

Table 9.4.: Goals, questions, and metrics for categorization and orchestration evaluation.

Correctness of the categorization, i.e., that failures are actually caused by mistakes they are traced back to in the categorization, is identified by validating whether there are further mistakes that caused the failures in the case study, denoted as Question 2.2. This is covered by measuring the number of failures that were resolved by fixing the implementation fault as a consequence of the mistake it was traced back to according to the categorization. For example, when a failure of multiple instantiations occurs, we search for missing element matchings that are the fault caused by the mistake of missing synchronization, to which such a failure can be traced back according to our categorization. We then measure whether the failure was resolved when we fix the fault in the implementation, e.g., by adding the missing element matching. This is reflected by the following metric, again indicating a higher degree of correctness with a higher value with a maximum of 1:

$$\textit{resolved failure ratio} = \frac{\#\ \textit{of resolved failures}}{\#\ \textit{of total failures}}$$

While we actually expect correctness and completeness to be given by construction of the categorization, it is unclear without empirical evaluation how relevant the different types of mistakes are, i.e., how often they lead to faults in actual projects, as defined in Question 2.3. This especially influences how important it is to avoid or identify specific mistake types. Therefore, we measure how often each type of mistake leads to a fault in the transformation implementations and compare it to the total number of faults to evaluate their ratio of occurrence. We reflect this in a metric for each mistake type representing the percentage of all faults it caused in the case study:

$$\textit{mistake type occurrence ratio} = \frac{\#\ \textit{of faults due to mistake type}}{\#\ \textit{of total faults}}$$

Finally, directly related to completeness of our categorization is the relevance of the orchestration problem, discussed in Chapter 7. We have seen that a transformation network cannot only fail in delivering consistent models after a change because mistakes led to faults in the single transformations or their combination to a network, but also because the problem of finding a consistent orchestration is, in general, undecidable. Since our categorization only considers actual mistakes made during network specification and does not reflect the orchestration problem, some failures may not be traceable to such mistakes, leading to a reduction of completeness as analyzed for Question 2.1. We have, however, already discussed in Chapter 7 that it is still unclear how relevant the orchestration problem is in practice. Thus, we use the results of our case study to evaluate this relevance, as asked in Question 3.1. We measure how often the application algorithm fails to yield consistent models only due to the orchestration problem. To identify that case, we validate whether an alternative orchestration would yield consistent models whenever the algorithm fails. In fact, not finding such an order would not prove that it does not exist, but we will see that this situation does not occur anyway. We thus measure the following metric for the ratio of failures due to the orchestration problem:

$$\textit{fail ratio} = \frac{\#\ \textit{of failures due to orchestration problem}}{\#\ \textit{of total failures}}$$

9.2.1.2. Synchronization

In addition to the evaluation of our categorization, we also used the case studies to evaluate our approaches for constructing correct transformation networks. We traced all failures back to the causing mistakes and fixed them according to our proposed approaches. The analysis of compatibility was already evaluated independently in Section 9.1. Since incompatibilities were obvious in all cases in which they occurred, we fixed them without running an explicit analysis. For all failures that could be traced back to missing synchronization, however, we applied our approach presented in Subsection 6.4.2 for making the transformations synchronizing. This enabled us to evaluate correctness and applicability of our approach to make transformations synchronizing and thus to fix or avoid mistakes at the transformation level, which we summarize in Table 9.5.

We have first measured whether the proposed approach for matching existing elements is correct, i.e., whether it leads to synchronizing transformations. This is covered by Question 4.1. To measure this, we counted the test cases in which failures occurred because of faults that were made at the transformation level in terms of missing synchronization and that we could fix by adding missing element matching. We applied our approach, i.e., we added the missing element matchings, and counted in how many cases this resolved all failures due to faults at the transformation level. This is covered by a metric that represents the success rate of the approach:

$$\textit{success ratio} = \frac{\#\ \textit{of tests with resolved failures after approach application}}{\#\ \textit{of tests due to which approach was applied}}$$

In fact, we only count the test cases after applying the approach that failed before due to faults at the transformation level, because we are only interested in test cases that failed before. Otherwise the metrics might exceed 1.

In the correctness evaluation, we only count the tests in which we were able to apply our approach. This was on purpose, because it may be possible

Goal 4: (Synchronization)	Show that the approach for matching elements avoids failures due to transformation level mistakes by construction.
Question 4.1: (Correctness)	In how many cases does the approach lead to correct synchronizing transformations?
Metric 4.1.1:	*Success ratio: Ratio between changes for which no failure due to faults at the transformation level occurs after applying the approach to all changes for which consistency was not preserved before applying the approach because of faults at transformation level*
Question 4.2: (Completeness)	In how many cases can the approach be applied?
Metric 4.2.1:	*Application ratio: Ratio of faults at transformation level that can be resolved by the approach to all faults at that level*

Table 9.5.: Goals, questions, and metrics for synchronization evaluation.

that the approach cannot be applied in all cases. First, this can be due to the fact that there is no unique information to match existing elements (see Subsection 6.4.2). Second, we may have missed further reasons than missing matching of existing elements preventing the transformations from being synchronizing. Both cases would restrict completeness of our approach, as considered by Question 4.2, because it would not be possible to resolve or avoid all possible failures due to missing synchronization by adding matchings for existing elements. To measure this, we counted the number of faults at the transformation level that we could resolve to the total number of faults:

$$application\ ratio = \frac{\#\ of\ \ resolved\ faults\ at\ transformation\ level}{\#\ of\ \ total\ faults\ at\ transformation\ level}$$

Although we applied the approach for achieving synchronizing transformations after identifying them as non-synchronizing rather than applying the approach to specify transformations that are synchronizing by construction, the results regarding correctness and completeness still apply if the approach is applied during transformation construction.

9.2.2. Prototypical Implementation

For conducting the case studies presented in the subsequent section, we have used a prototypical implementation in the Vitruvius framework (see Subsection 2.3.2) [Kla+21]. It supports the view-based development of consistent systems by managing a consistent representation of all information about a software system, from which views can be derived to be modified by the users. Internally, the system is represented as a set of models of existing or newly defined languages, which are kept consistent by means of bidirectional model transformations. The transformations operate in an incremental and delta-based way. They are incremental, because they update the existing models rather than creating new ones upon changes. They operate delta-based, as they do not receive the modified state of a model but a delta between the old and the new state. This conforms to our notion of changes (see Definition 4.3). To achieve this, the framework records atomic changes to the models, i.e., element creations and deletions as well as attribute and references changes, as discussed in Subsection 6.4.3 and depicted in Figure 6.7, and passes them to the transformations. Currently, it lacks support for the combination of multiple transformations for keeping multiple models consistent, which is why we implemented our approaches in a case study with that framework.

In our case studies, we use the Reactions language defined for the Vitruvius framework, which we have already introduced in Subsection 2.4.3. It allows to define unidirectional consistency preservation rules according to Definition 6.1. Defining such unidirectional rules for both directions between two metamodels yields a bidirectional transformation according to Definition 6.3. These transformations only have an explicit representation of the consistency preservation rules, whereas the consistency relations are only implicitly defined as the fixed points of the application of the consistency preservation rules.

The Reactions language uses the so called *correspondence model* of the Vitruvius framework to identify corresponding elements according to the implicitly defined consistency relations and thus implements a witness structure according to Definition 4.19. It consists of *correspondences*, of which each relates two sets of elements. It enables to trace when elements were changed to update the corresponding elements rather than deleting and creating them. We have discussed in Subsection 4.4.1 that this still conforms to our formalism, although we omitted any kind of trace model there.

```
reaction {
   after element pcm::Component inserted in
      pcm::Repository[components]
   call {
      val component = newValue
      createClass(component)
   }
}

routine createClass(pcm::Component component) {
   match {
      require absence of uml:Class corresponding to component
      val componentsPkg = retrieve uml::Package
         corresponding to component.repository
         tagged with "componentsPackage"
   }
   action {
      val class = create uml::Class and initialize {
         class.package = componentsPkg
         class.name = component.name + "Impl"
      }
      add correspondence between component and class
   }
}
```

Listing 9.1: Reaction creating a UML class for a PCM component. Adapted from [Kla+21, Lst. 2] and extended from Listing 2.1.

In Listing 9.1, we depict an extension of the example in Listing 2.1, which we have explained in Subsection 2.4.3. The extended Reaction is also triggered by the insertion of a PCM component and calls a routine that is responsible for restoring consistency for a consistency relation between PCM components and UML classes. It thus checks in the *match* block whether the change affects that consistency relation and in that case, in addition to the original implementation, checks that no corresponding class already exists to avoid multiple instantiation for the synchronization scenario. It then creates a corresponding UML class in a retrieved package for components.

In Chapter 7, we have discussed different options for the orchestration of transformations in an application algorithm. In the Vitruvius framework, we have implemented a simple depth-first execution of transformations without an artificial execution bound. This means, for a given change all transformations involving that changed model are executed consecutively. After the execution of each transformation, this approach is recursively applied to the model changed by that transformation, which implements the depth-first execution. If the model is not changed, i.e., if the models are already consistent, the recursion aborts. Finally, this leads to termination of the algorithm. This results in an algorithm comparable to the provenance algorithm proposed in Section 7.4, as it implements a similar recursion strategy. In contrast, the implemented strategy does not only consider already executed transformations in the recursion and does not define an execution bound. In consequence, the implementation may not terminate.

Since the transformations defined in the Reactions language only contain implicit consistency relations by the fixed points of their consistency preservation rules, checking consistency for the recursion to abort is conducted by checking whether the transformation performed any changes. If this is not the case, the models are considered consistent by construction. We have discussed this as an option for the realization of a CheckConsistency function within an application algorithm in Subsection 7.2.1. The implementation of the framework with the Reactions language is available at GitHub [GitVit].

9.2.3. Case Studies

We have performed two case studies based on one set of metamodels and transformations between them defined in the Reactions language. The case studies employ the metamodels PCM for component-based software architecture descriptions, UML for object-oriented software design, and Java for source code development, as introduced in Section 2.5. Transformations are defined between each pair of these metamodels, based on two sets of consistency relations that we have also introduced in Section 2.5. This covers relations between PCM and object-oriented design, applied to both Java and UML, and relations between UML and Java.

We haven chosen these metamodels and transformations for our case studies because except for one transformation they were explicitly developed independently without the goal of using them within a transformation network, yielding the possibility to evaluate our categorization and error resolution approaches. The transformations even assumed that they are only executed in one direction after a user change. It is difficult to find further comparable examples, because we require transformations whose induced graph contains cycles, as otherwise most of the discussed problems do not occur at all. If such transformations exist, however, they were usually defined in a way that they properly work together, as otherwise they would not be usable at all. They would have to be developed in a scheme similar to the one proposed by Kramer et al. [Kra+16] to exclude different types of possible biases.

The preservation of consistency between PCM and Java according to these relations (see Table 2.1) using the Reactions language was implemented in the Master's thesis of this thesis' author [Kla16] in the context of the dissertation of Langhammer [Lan17]. At that point in time, the transformation was only defined to be executed once in one direction and, in particular, not to be used in a transformation network. In addition, Syma defined the bidirectional transformation between PCM and UML in his Master's thesis [Sym18]. He also proposed a formal specification of those relations and their preservation [Sym18, Sec. 5]. This transformation was defined to be used in a transformation network and therefore implements the matching of existing elements according to Subsection 6.4.2 to achieve synchronization of the transformation.

PCM models can also contain *service effect specifications* as abstract specifications of the behavior of services provided by a component. Consistency between behavior specifications in PCM and their implementation in Java is one of the reasons why, in general, consistency between PCM and Java cannot only be expressed across UML class models. We do, however, not consider that consistency relation in this case study, because we focus on structural consistency relations, as motivated in Subsection 3.1.2. Since these behavioral descriptions share an isolated relation between PCM and Java, it is not relevant for our considerations on transformation networks anyway.

The preservation of consistency between UML and Java according to these relations (see Table 2.2) was implemented using the Reactions language within a Bachelor's thesis supervised by the author of this thesis [Che17]. Like for the transformation between PCM and Java, this one was implemented to be used in one only direction and was, thus, especially not intended to be used in a transformation network.

The implementations of all transformations are available in a corresponding GitHub repository of the Vitruvius project [GitApp]. Each of them also contains a sophisticated set of test cases, which were supposed to test each

↓ **From** / **To** →	PCM	UML	Java
PCM	-	57	40
UML	68	-	63
Java	16	49	-

Table 9.6.: Complexity of the case study transformations in terms of the numbers of Reactions in each consistency preservation rule, i.e., the number of change types it is able to react to.

transformation only executed in one direction after changes to one model. We reused and extended these test cases for our case study. This setup of independently developed transformations and test cases ensures that there is only low risk of the transformations and test cases to be initially aligned with each other, which could result in a bias of the results.

To give an impression of the complexity of the transformations, we depict the numbers of Reactions in each of the six unidirectional consistency preservation rules in Table 9.6. They conform to the numbers of change types each of these consistency preservation rules reacts to. The lower number of Reactions between Java and PCM is mainly caused by several elements of the PCM being mapped to the same elements in Java. For example, components and all kinds of data types are mapped to classes in Java, such that the Reactions in Java react to less change types and instead make more distinctions within the routines to separate the affected consistency relations.

The scenarios used for our case study, i.e., the changes to which we applied the transformations for preserving consistency, are twofold. They consist of existing test cases for the implemented bidirectional transformations and of the simulated construction of an existing, comprehensive system model.

We have reused the test cases that were already implemented for the existing bidirectional transformations between PCM and UML as well as between UML and Java. These test cases implement fine-grained tests for all possible types of changes according to the consistency relations, i.e., all kinds of relevant insertions, removals, and modifications of involved elements. They set up minimal models and then perform the changes to be tested. Afterwards, they validate that the expected models exist. The according test cases are summarized in Table 9.7, expressing the number of test cases for each underlying consistency relation. We have split the test cases between PCM and

Consistency Relation	Test Cases
PCM ↔ UML Core	
Repository	4
Interface	2
System	2
Composite data type	4
Repository component	2
Provided role	2
Assembly context	2
Total	**18**
PCM ↔ UML Additional	
Signature	6
Parameter	6
Attribute	5
Required role	3
Total	**20**

Consistency Relation	Test Cases
UML ↔ Java	
Package	8
Class	25
Class method + parameters	29
Field + association	19
Enumeration	14
Interface	10
Interface method	9
Total	**114**

Table 9.7.: Numbers of test cases for the different consistency relations in the case studies.

UML into two categories, because the second case study only uses the first of these categories. In total, we used 38 existing test cases between PCM and UML, as well as 114 test cases between UML and Java. The gap between these number has two reasons. First, UML and Java share more information, such as visibilities and modifiers of fields and methods. Second, the granularity of the test cases differs, because they were developed by different persons, thus a test case between PCM and UML validates more scenarios than one between UML and Java.

In addition, we have used the Media Store system model [SK16], which is a comprehensive case study system for the PCM. It represents the architectural description of a system for managing different types of media files, i.e., uploading and downloading them to a database via a web server. It consists of several components, data types, and interfaces, which are provided and required by the components. For this system, representations as a PCM model as well as in Java code exist. We have simulated the construction of that system model by producing a change sequence as if the system was developed from scratch and applied the transformation network to these changes to create the other two models. This conforms to the *Reconstructive Integration Strategy (RIS)* proposed by Langhammer [Kla+21; Lan17] and implemented in the Vitruvius framework. Afterwards, we have validated that the expected models, conforming to the consistency relations, were created. This is covered by two additional test cases.

Based on these test cases, we have performed two case studies. In the first *linear network study*, we have realized a linear network by combining two bidirectional transformations. This network does not contain any cycles of bidirectional transformations. This study was conducted in the Master's thesis of Syma [Sym18] and published in previous work [Kla+19b]. In the second *circular network study*, we have realized the network of all three bidirectional transformations, thus also containing a cycle of transformations. This study was conducted in the Master's thesis of Sağlam [Sağ20]. Both studies were conducted in the previously explained iterative process of identifying failures and resolving them by fixing the causing faults. We have tagged the states before and after the iterations of these studies in the according GitHub repository [GitApp].

Linear Network Study

In the first study, we restricted ourselves to a linear network by combining the transformations between PCM and UML as well as between UML and Java. In this situation, no synchronization of transformations would be necessary, because there is always only one path of transformations between two models across which changes can be propagated. Thus, it would be sufficient to execute both transformations in one direction after changes in one model to achieve consistency, as long as the transformations are correct. A synchronizing bidirectional transformation, however, can require its consistency preservation rules to be executed multiple times, as discussed in Subsection 6.3.4, to let them react to changes in both models and achieve a fixed point by improving partial consistency in each step. This means, executing a synchronizing bidirectional transformation in a linear network should terminate after executing one consistency preservation rule once, as the one in the other direction should not react to the generated changes and no further changes that need to be synchronized can exist. Since the existing transformations were not developed to be synchronizing, we could expect errors to occur here, although no synchronization would be necessary at all. For example, if the consistency preservation rule in one direction creates an element and the one in the opposite direction processes this creation without considering that this may not be a user change that needs to be processed, it will create another corresponding element, due to missing matching of existing elements.

The transformation between PCM and UML was developed in the context of this case study and, purposely, designed to be synchronizing. This allowed us to get an impression of whether it is possible to develop a transformation that is synchronizing by construction. The transformation between UML and Java pre-existed and was designed to be executed in only one direction. Thus, it neither matched existing elements using implicit unique information to be synchronizing nor using explicit unique information, i.e., correspondences, to be executed in both directions without duplicating elements.

Based on these transformations, we conducted the already depicted process of executing the scenarios of existing test cases and case study system, identifying the occurring failures, tracing them back to the causing mistakes and then fixing the faults in the implementation to resolve the failures. We employed all test cases summarized in Table 9.7 without further modifications and, in addition, the construction simulation of the Media Store system.

Circular Network Study

In the second study, we started with the results of the first one, i.e., we employed the transformations that were already improved due to the identified faults in the first study. In addition, we considered the transformation between PCM and Java to induce a cycle in the graph of the transformations. Consequently, in this study a synchronization scenario occurs, because changes can be propagated across multiple paths of transformations.

Again, we reused existing test cases, but in this study we extended them to validate consistency of all three models rather than only the two they were developed for. We used the *PCM ↔ UML Core* tests depicted in Table 9.7, which perform different types of changes in PCM and UML models, and extended them to also validate the Java models.

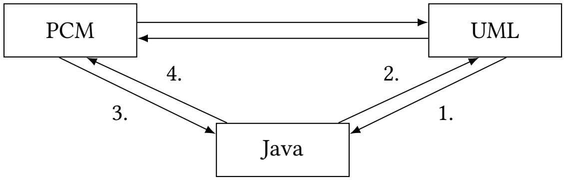


Figure 9.1.: Phases of the circular network study by depicting the transformations that are incrementally added in each of the phases.

Instead of a big bang integration of all transformations, we incrementally added the unidirectional consistency preservation rules to the network to evaluate and resolve the occurring failures in multiple phases. These phases are depicted in Figure 9.1. We started with a network of the transformation between PCM and UML as well as the unidirectional consistency preservation rule between UML and Java. In the further phases, we completed the bidirectional transformation between UML and Java by adding the consistency preservation rule in the reverse direction, then first added the rule between PCM and Java, and finally added the one in the opposite direction. This also allowed us to evaluate how the topology of the network affects the types of mistakes that lead to failures. Although the first two phases were already covered by the linear network study, we still conducted them again because of the extension of the test cases to the third model, which revealed further errors that were not detected before.

9.2.4. Results and Interpretation

We present the results of the introduced iterative process of identifying failures and the causing faults and mistakes, as well as of fixing the faults to resolve the failures. In Table 9.8 we summarize the numbers of faults we found in each case study because of the different mistake types, as well as the numbers of failures they resulted in when executing the test scenarios. The detailed analyses can be found in the theses of Syma [Sym18] and Sağlam [Sağ20]. In the following, we discuss the aggregated and interpreted results and only go into the details where relevant.

The presented numbers of faults represent the actual parts of transformations that needed to be fixed. For example, each fault due to missing synchronization manifests as a missing matching of existing elements, which needs to

Case Study	Mistake Type	Faults	Number / Type of Failures	
Linear network study	Missing synchronization	25	154	Multiple instantiations
	Incompatible constraints	1	3	Non-terminations (divergence)
	Contradicting options selection	1	2	Non-terminations (alternation)
	Total	**27**	**159**	**Failures**
Circular network study	Incorrect transformation	12	37	Inconsistent terminations
	Missing synchronization	13	57	Multiple instantiations
	Incompatible constraints	4	24	Multiple instantiations
	Contradicting options selection	0	0	-
	Total	**29**	**118**	**Failures**

Table 9.8.: Mistakes, numbers of faults, and number and type of faults in the case studies.

be added at one place within the transformations. The counted numbers of failures are not that meaningful but are only supposed to give an impression of the extent of failures. This is due to the fact that these numbers are highly dependent on the kind and number of the used test scenarios, as they determine how often a fault manifests in terms of a failure. Additionally, faults interfere, as one fault may hide another one when it leads to a failure before the transformation with the other fault was applied. This does also explain why there are more failures than there are test cases, especially in the circular network study. There, some missing element matchings only led to failures after another was fixed, thus leading to the same test failing twice because of two faults. In consequence, the types of failures in the overview are more relevant than the actual numbers of occurrences.

Linear Network Study

We performed two iterations in the linear network study. In each iteration, we fixed all faults that we could identify because of the test scenario failures. After two iterations, no further failures occurred. In total 159 failures occurred, of which 154 occurred in the first iteration in terms of multiple instantiations due to missing element matchings. These 154 failures correspond to all test scenarios, as we had in total 152 existing test cases and two scenarios with the Media Store case study system. These failures did, in fact, only occur because the transformations between UML and Java did not even contain element matchings using explicit unique information, i.e., correspondences. Thus, when Java elements were created by the transformation execution from UML to Java, the execution in the reverse direction treated the creation changes as if they were performed by a user and created elements in the UML model again. This could already be fixed by checking the correspondence model for the existence of correspondences, i.e., by applying element matching based on explicit unique information according to Subsection 6.4.2.

In the second iteration, five further failures occurred. In all cases, the execution did not terminate, which was because of divergence in three cases and because of alternation in two cases. The reasons for these failures were incompatible constraints and contradicting options selections by the transformations. The Java model contains the fully qualified name of a class, whereas the UML model only contains the simple name, which was correctly propagated from UML to Java, but the namespace prefix was not removed in the opposite direction. Thus, the considered consistency relations for both directions were incompatible, leading to a repeated addition of the namespace and thus divergence due to an endless extension of the class name. This shows that already within a single bidirectional transformation the unidirectional consistency relations can be incompatible. The alternation occurred in terms of endlessly swapping visibilities of methods between UML and Java, because different options for mapping default visibilities exist, for which the consistency preservation rules in both directions chose contradicting ones.

Most importantly, all faults occurred in the transformation between UML and Java. Thus, the transformation between PCM and UML, which was developed to be synchronizing with our proposed approach to match existing elements, operated properly by construction.

Circular Network Study

In the circular network study, we performed 29 iterations, which conforms to the 29 identified faults. This is because we decided to fix one fault in each iteration. We investigated the failures, traced one of them back to a fault, fixed that fault, and validated how many failures this resolved. Finally, we were able to resolve all failures by fixing the identified faults, such that all test scenarios can be executed successfully. Details about the failures resolved by the fix for each fault are described in the Master's thesis of Sağlam [Sağ20].

Across these iterations, 118 failures occurred. In contrast to the first study, we also counted incorrectness of the transformations in this study, which is actually out of the scope of our evaluation, because we assumed the transformations to be correct, as correctness of individual transformations is a separate and well-researched topic. It was, however, interesting to see that some faults because of incorrect transformations are only detected when using a transformation within a network rather than using it in an isolated way. This is due to the reason that other transformations produce edge cases that were not covered by the transformations and their test cases before. For example, the transformations implicitly assumed specific naming schemes within the models, which are not guaranteed to be followed. If other transformations then produce models that do not follow this naming scheme, this leads to failures that reveal incorrectness of the transformation. In total, twelve faults within incorrect transformations were revealed by 37 failures during their execution in a network. Seven of these faults were revealed in the first two phases of the case study, in which the transformation between UML and Java was added (see Figure 9.1). They were first revealed in this case study, and especially not in the linear network study, because of further validations added to the test cases.

The majority of 57 failures were multiple instantiations of elements due to missing synchronization. In 13 cases, matchings of existing elements were missing. Additionally, four faults because of incompatible constraints led to 24 failures in terms of multiple instantiation. This is particularly interesting because in this case multiple instantiation was not caused by missing synchronization, which we expected to be the main reason for multiple instantiation. In this case, the incompatible constraints were caused by different, incompatible naming schemes. For example, all transformations assume a single UML model to exist, but they assume it to have different names, which results in multiple UML models being instantiated. In practice, such cases can be distinguished from multiple instantiation due to missing synchronization, because although there are multiple elements where there should only be one, they can be distinguished by differences in their names or other key information used to identify them.

In the following, we use these results to evaluate the defined metrics for answering our evaluation questions depicted in Table 9.4 and Table 9.5.

9.2.4.1. Categorization and Orchestration

All failures that we identified in the test scenarios were covered by our categorization and could thus be traced back to potential mistakes and faults they were caused by. Additionally, we were able to fix all faults to which the occurring failures were traced back. We achieved that all test scenarios can be executed successfully after fixing the causing faults. Although not part of our categorization and contributions, we also fixed the incorrect transformations, as they could otherwise hide other failures due to further faults. Finally, whether we also count these failures or not, we were able to classify and resolve all occurring failures, thus leading to:

$$\textit{classified failure ratio} = \textit{resolved failure ratio} = 1$$

We introduced these metrics as indicators for the completeness and correctness of our categorization in Question 2.1 and Question 2.2. Since none of the occurring failures was caused by any other mistake than we expected according to our categorization, we assume this a valuable indicator for its completeness and correctness.

Most importantly, we aimed to identify the relevance of the different types of mistakes according to Question 2.3 by counting the numbers of faults they caused. We summarize the results of this analysis, depicting the according metrics values, in Figure 9.2. We found that most faults were caused by missing synchronization. Across both studies, more than 85 % of the faults were caused by missing synchronization, and even if only considering the circular network study they made up more than 75 % of all faults. Incompatible constraints led to the second highest numbers of faults, namely about 10 % when considering both case studies and about 25 % when only considering

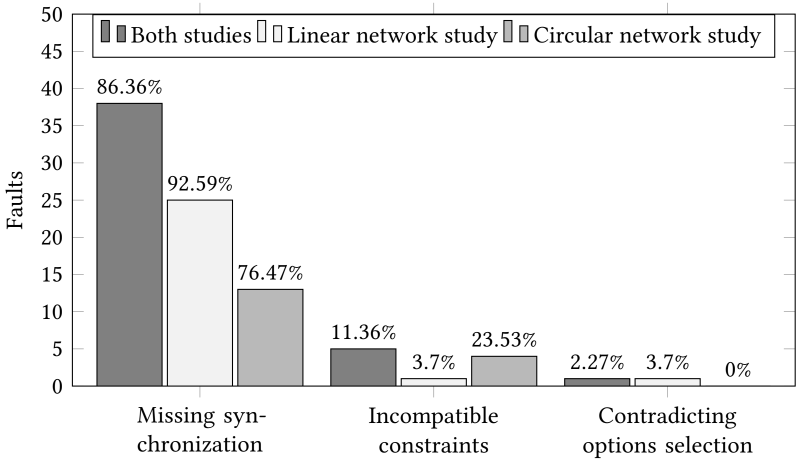


Figure 9.2.: Absolute numbers of faults due to different mistake types in both case studies. Percentages are relative to total number of faults in the particular case study.

the circular network study. Finally, the contradicting selection of options only led to a single fault in the linear network study.

The actual numbers must be assumed to be rather imprecise due to the low numbers of faults. For example, only five faults due to incompatible constraints were detected in total. Nevertheless, the relations between the numbers of fault occurrences show that missing synchronization was by far the most important reason for faults in transformation networks. Since synchronization can be achieved by construction without knowing about the other transformations in a network, this indicates that most errors in transformation networks can already be avoided by construction of the individual transformations. Incompatibilities, as the reason for the second highest number of faults, can at least be analyzed when developing the network, which means that it can at least be detected at design time without and before productively executing the transformations.

Finally, we also aimed to evaluate the relevance of the orchestration problem in practice. We have discussed that its evaluation is directly related to completeness of our categorization, because if we are able to classify each failure and trace it back to a fault covered by our categorization, there are no failures actually caused by the orchestration problem. Since we were able to resolve all failures by fixing mistakes covered by our categorization, undecidability of the orchestration problem did not lead to the situation that the Vitruvius framework was no able to find a consistent orchestration in any scenario. Consequently, the according metric measuring the fail ratio evaluates to 0:

$$\textit{fail ratio} = 0$$

In particular, we selected a simple recursive strategy for the orchestration, which was still able to always find a consistent orchestration. In answer to Question 3.1, this indicates that the order in which transformations are executed may not be that relevant in practice, thus leading to the orchestration problem not being particularly relevant in practice. We must, however, consider that the orchestration problem is especially relevant if multiple options for preserving consistency exist, like we have discussed as a possible restriction in Subsection 7.2.4. We have, however, seen that contradicting selection of options to restore consistency was not even a relevant fault in the case study, which may indicate that it is either not that problematic in practice or that the case study does not contain many cases in which multiple options for restoring consistency exist.

9.2.4.2. Synchronization

Most faults in both case studies were caused by missing synchronization. In total, 38 faults led to 214 failures, and even if only considering the circular network study, still 13 faults could be identified. We were able to fix all these faults by adding matchings for existing elements by explicit and implicit unique information, i.e., using correspondences as well as key information, as proposed in Subsection 6.4.2. Thus, all 214 scenarios that failed due to missing synchronization, i.e., mistakes at the transformation level, and in which we could apply our approach, succeeded after applying the proposed approach for constructing a synchronizing transformation by matching elements. Thus, our approach operates correctly according to Question 4.1, as its application always leads to correct synchronizing transformations in the case studies, as reflected by the success ratio metric:

$$\textit{success ratio} = 1$$

In addition, we were able to apply our approach in all cases in which faults at the transformation level led to failures during execution. More precisely, there we no cases in which we were not able to perform matching of elements due to unique information, thus requiring us to use heuristics and having the possibility to fail. Additionally, there were no failures due to missing synchronization that occurred for other reasons than missing element matchings. This indicates completeness of our approach according to Question 4.2, as there are no cases in which the approach could not be applied and resolve failures due to faults at the transformation level, which is also reflected by the according metric:

$$\textit{application ratio} = 1$$

We have used the transformation between PCM and UML, which already applied our approach for matching existing elements to achieve a synchronizing transformation by construction. Since we detected no failures due to missing synchronization of that transformation in either of the case studies, it serves as an additional indicator for the correctness and completeness of the approach, in addition to the measured metrics.

In conclusion, we found the proposed approach for constructing synchronizing transformations to be correct and complete in the considered case studies. This serves as an indicator for its general correctness and completeness and thus the possibility to use it as a constructive approach for achieving synchronizing transformations. Since we found missing synchronization to be the most important reason for failures in transformation networks, concluding that we can achieve synchronization by construction means that we are able to avoid most of these failures by construction.

9.2.4.3. Topology Effects

We have performed the circular network case study in a four-phase process, as explained at Figure 9.1, adding a unidirectional consistency preservation rule in each phase to analyze how the topology affects the types of faults that are revealed by failures when applying our test scenarios to the network of each phase. We depict the numbers of faults as consequences of the different mistakes types in the different phases in Table 9.9.

Phase → / **↓ Mistake Type**	PCM ↔ UML → Java	PCM ↔ UML ↔ Java	PCM ↔ UML ↔ Java, PCM → Java	PCM ↔ UML ↔ Java, PCM ↔ Java
Incorrect transformation	5	2	4	1
Missing synchronization	0	0	6	7
Incompatible constraints	0	0	2	2
Incompatible options selection	0	0	0	0

Table 9.9.: Numbers of faults due to different mistake types by the phase of the circular network case study with the stepwise addition of unidirectional consistency preservation rules.

In the first two phases, the consistency preservation rules of the transformations between UML and Java are added. Since these two phases were already covered by the linear network study, it was likely that only few further faults are found by extending the test cases. In this case study, we extended the test cases to also validate the generated Java model, whereas in the linear network case study the test cases validated only the PCM and UML, or the UML and Java models, respectively, but not the third model. Interestingly, in these phases only faults due to incorrect transformations were found as reasons for failing test scenario executions. On the one hand, this shows that it seems to be difficult to construct correct transformations that consider all possible scenarios. In this case, the combination of transformations to a network revealed incorrectness due to cases that were not considered for a transformation on its own before. On the other hand, this indicates that it may already be sufficient to validate pairwise consistency of models in multiple scenarios when executing a transformation network rather than validating consistency of all models, since no faults due to the combination of transformations to a network could be found in these phases. As we have seen in the linear network study, such faults can actually occur already in a linear network.

As expected, in the last two phases especially faults due to missing synchronization are revealed by the occurring failures. This is due to the reason that these phases introduce a cycle in the transformations, which leads to the situation that transformation need to synchronize changes, as both models may have been changed across two paths of transformation executions. Even in these phases, failures occur due to incorrect transformations.

As the essential takeaway, it is important to not only consider mistakes specific to the combination of transformations to a network but also to consider correctness of the individual transformations. The results of our case study indicate that assuming transformations to be correct may not be reasonable in practice, thus transformations may fail when combined to a network because of faults that they already contained before, but which never led to failures when executing them in an isolated way.

9.2.5. Discussion and Validity

From the two discussed case studies, we can derive several important insights. This covers correctness of our categorization and our synchronization approach as well as, and in particular, regarding the relevance of different mistake types and the relevance of the orchestration problem.

9.2.5.1. Insights

We found that most faults in the case study were due to missing synchronization. Synchronization is, however, achievable by construction, as we have also validated in the case study. The proposed approach for synchronizing transformations can be applied to a single transformation without knowing about the other transformations to combine it with. In consequence, a high number of faults in transformation networks can already be avoided by construction of the single transformations.

In the iterative process of the case studies, we found that the first occurring failures were multiple instantiations because of missing synchronization. Adding the element matchings for synchronization then revealed further faults, for example, because of incompatible relations. First, this is not surprising, because multiple instantiation occurs upon creation of elements, which is the first step in consistency preservation. Thus, faults due to missing synchronization lead to early failures. Second, this shows that faults due to missing synchronization can hide further faults. Thus, it is important to resolve errors at the transformation level first, or, in the best case, to avoid them by construction.

In the circular network study, we detected multiple instantiations due to incompatible consistency relations rather than missing synchronization. We have discussed in Chapter 8 that this can, theoretically, be the case, but still multiple instantiations are expected to be the consequence of missing synchronization in most cases. While this is still given in the case studies, we also found that two kinds of multiple instantiation can be distinguished to identify their cause. In case of missing synchronization, an element with the same key information, such as the name or other information, is created. For matching existing elements, we proposed to use unique key information, such as names, to identify the existence of an element. On the contrary, if the elements differ in their key information but still should be the same, there is a fault in the transformations in terms of incompatible consistency relations, as they use different ways of relating the key information although it should actually be the same.

Finally, we found undecidability of the orchestration problem not to be relevant in our case studies. This does not validate that it is not relevant in practice at all but at least serves as an indicator that it is not such a central problem that transformation networks are unlikely to ever work properly. Still, external validity of that statement has to be improved by further studies.

9.2.5.2. Threats to Validity

In the following, we discuss different threats to the validity of the discussed results. The limited set of case studies especially limits external validity. We discuss how we have mitigated validity threats and for which reasons validity of the statements may be actually restricted, distinguished by construct, internal, conclusion, and external validity [Woh+12].

Construct Validity If transformations are in some way aligned with each other a priori, certain errors would not occur at all, thus reducing the number of faults and influencing their distribution. We have mitigated this threat by developing each transformation in an isolated project without knowing that it is supposed to be combined with other transformations and by giving the development tasks to different students. The only bias may be that the author of this thesis supervised the students that developed the different transformations. Still, this is a situation comparable to practice, because developers may also exchange information but not have an explicit representation of common knowledge.

We employed the Reactions language to implement the transformations. The language may affect how likely specific faults are to be made. For example, the language and the Vitruvius framework it is embedded into use a delta-based approach to consistency preservation, which may already prevent problems that may occur with a state-based approach to consistency preservation. We did, however, purposely use a language that provides a rather low level of abstraction to reduce the chance that this influences how prone the implementations are to specific faults. For example, using QVT-R, which already provides the ability to define keys for matching existing elements, would have prevented specific faults already by construction.

Internal Validity Using transformations that were not initially synchronizing and fixing them during the case studies leads to two threats to validity. First, this process obviously leads to a high number of faults and failures due to missing synchronizing, which would not have been the case when using transformations that are synchronizing a priori. Since we wanted to evaluate how important it is to have synchronizing transformations, this setup was reasonable. Still and second, it would be valuable to conduct a case study in which transformations are already synchronizing. This can give further and more precise insights regarding the relevance of the other types of faults and, more importantly, the process of fixing faults rather than avoiding them may introduce a bias. When fixing the faults, additional fixes beyond the application of our synchronization approach may have been performed until the test scenarios succeeded, which cannot occur if transformations are already developed to be synchronizing. We mitigated this threat by constructing at least one of the transformations to be synchronizing and found that it did actually not lead to any failures because of missing synchronization. Still, we plan to perform a case study to further validate how well synchronization can be achieved by construction and how this influences the relevance of other mistakes, as we discuss in Subsection 9.2.6.

Conclusion Validity The central threat to conclusion validity is the low amount of data. Some fault types occurred only once in the case studies, thus potentially reducing the significance of the results. This especially means that the actual values, especially for the relevance of the mistake types, cannot be considered representative. Still, we expect the general conclusions regarding relevance to be correct, because the number of test scenarios was high enough and led to a significant number of failures.

External Validity External validity in terms of generalizability of the results is especially affected by the representativeness of the case studies. To this end, a threat may be the low number of performed case studies. Our results are, however, not highly dependent on the actual contents of a case study, i.e., the contents of the models and the transformations. They rather depend on the existence of specific patterns, such as the possibility for transformations to select from multiple options to restore consistency, and potentially on the size of a transformation network. Especially the evidence of our results regarding the relevance of faults at the network rule level needs to be further validated in additional case studies. This is, however, difficult due to the limited availability of evaluation scenarios. Regarding the size of transformation networks, we do not expect larger networks to reveal further problems, because the problematic situations are those in which changes to the same models are performed across two paths of transformation executions, which already exist with a cycle of three transformations. In addition, only the number of case studies is rather low, but the number of considered scenarios within the case studies represents a comprehensive set of scenarios.

The selection of scenarios for the case studies may have influenced whether specific kinds of mistakes can occur at all. In particular, the used transformations can all rely on unique key information for identifying matching elements. Thus, we may have identified the synchronization approach to be correct and complete, because the case study scenarios do not reflect problematic cases. This is, however, essential complexity that cannot be solved with any comparable approach, because if no unique key information exists, only heuristics to identify elements can be applied. To circumvent that problem, it would only be possible that transformations know each other and use trace links generated by the other transformations, such that they can rely on meta data attached to these links to uniquely identify elements. This does, however, break the assumption of independent development and thus cannot be achieved by construction of a single transformation but essentially requires transformations to be aligned with each other or to be defined as multidirectional transformations.

Finally, the consistency relations in the case studies do not provide many different options for models to be consistent. Thus, the chance that transformations decide to use different, contradictory options to restore consistency may be unlikely. This may have led to only few faults at the network rule level and may thus have biased the results. It does especially also influence the ability that undecidability of the orchestration problem leads to a failure. It is, however, also a consequence of using a transformation language that does not explicitly define consistency relations that led to this result. Since consistency relations are only implicitly defined by the consistency preservation rules, a contradictory selection of options manifests as an incompatibility of the implicit consistency relations, as the options to select from are not documented anywhere. Thus, to mitigate this issue, consistency relations would have to be defined explicitly.

9.2.6. Limitations and Future Work

In addition to the discussed results, the case studies revealed limitations of our approaches and insights, which represent our starting points for future work in terms of practical application improvements, conceptual progress, and additional necessary evaluations.

Element Matching Implementation Within the case studies, we have implemented the matching of existing elements manually, i.e., using the existing constructs provided by the transformation language. This is a costly and cumbersome task, which is also prone to errors in the accidental complexity of the mechanism due to repetitions of the same logic. Since the mechanism is always similar and only differentiates in the key information used to search for, it could be embedded into an API or language construct to be reusable.

In future work, we thus want to investigate how we can integrate the patterns for constructing synchronizing transformations into existing transformation languages, such as the Reactions language used in the evaluation. In particular, we want to investigate how well QVT-R fits for that purpose, as it already allows to define keys for matching existing elements [QVT, Sec. 7.10.2.].

Semantic Element Matching In the evaluation, we have detected cases that may be expected to be consequences of incompatibilities but are actually not. For example, the transformation between UML and PCM creates a repository with a name starting lowercase, whereas the transformation between Java and PCM generates a repository with a name starting uppercase. Then the repository created by one transformation is not matched by the other, which is correct as the transformations define consistency relations with different capitalizations of the repository. Thus, having two repositories is correct in this case, although it may not be expected but would intuitively be assumed to be incompatible. It is expected that both repositories are supposed to represent the same element (see [Sağ20, Fig. 6.4]), thus having the same semantics although their uniquely identifying information, the name, is not equal. In this case, however, a different notion of correctness is violated that we explicitly excluded for this thesis in Subsection 4.2.3. This notion assumes a common global knowledge to which the transformations must be correct, thus requiring knowledge about the semantics of the elements, for example, in terms of a global specification of consistency or a mapping to a common, verifiable formalism.

In future work, we want to consider how such a matching in terms of element semantics rather than plain syntactic matching can be performed. Although it requires the transformation developer to know about the semantics of the elements to define that they have to be syntactically matched, this process would be even more valuable if the matching was performed on more semantic information. One such example that we have considered in Chapter 5 was the swap of first name and last name by one consistency relation, which does not represent an incompatibility according to our definition but may intuitively be undesired. Mapping all elements to a common semantic representation could improve such a matching process. In Chapter 11, we present an approach that proposes to describe transformations in terms of descriptions of the common elements of the metamodels, thus representing their common semantics.

Interaction with Users In our assumptions in Subsection 1.3.2, we explicitly excluded semi-automatisms in consistency preservation from the considerations in this thesis. Actual transformations can, however, be semi-automated by integrating decisions of users. For example, a user may select whether an added class shall represent a component or not. In terms of consistency relations, such decision options can be represented by multiple consistency relation pairs that represent all options to select from. Within consistency preservation rules, such user decisions can, however, be problematic. If both the transformation between UML and PCM as well as the one between UML and Java ask the user whether a class shall represent a component, this, on the one hand, is annoying if the user is asked twice and, on the other hand, can even lead to conflicting decisions by the user. We have already discussed how the selection of different options by transformations can prevent the network from finding consistent models and in such a case, even worse, only one user decision can be correctly reflected in the result. Thus, it is part of our future work to find out how user decisions can be aligned across multiple transformations.

Alignment of Consistency Preservation Rules We have made important insights regarding synchronization of transformations and compatibility, thus correctness at the transformation and network relation levels. At the network rule level, however, we only found the selection of contradicting options for consistency to be problematic, but we were neither able to restrict them without reducing expressiveness nor to define any reasonable notion for correctness at all. Thus, it remains an open question how consistency preservation rules need to be aligned with each other in a transformation network such that a consistent orchestration of them always exists and, in the best case, that it can easily be found. While finding a consistent orchestration is difficult due to undecidability of the orchestration problem anyway, in this thesis we focused on how to conservatively deal with this situation. Although the evaluation indicated that the orchestration problem may not be highly relevant in practice, having a comprehensive, systematic theoretical understanding at that level, especially of how consistency preservation rules influence the ability to find consistent orchestrations and whether there are further issues except the selection of contradicting options, would still be beneficial, which is why we consider it as important future work.

Synchronization Transformation Construction Case Study Finally, we have discussed two case studies to validate different properties of our proposed error categorization as well as the approach for constructing synchronizing transformations. Although we were able to derive valuable conclusions, the case study was biased by the fact that two of three transformations were not designed to be synchronizing and, as part of the case study, fixed to be synchronizing during that study. Still, it would be valuable to perform a case study with a focus on the construction of synchronizing transformations to improve evidence on the ability of correctly and completely achieving synchronizing transformations with our proposed approach.

9.3. Orchestration Algorithm

In Section 7.4, we have proposed an application algorithm for transformation networks, which is proven correct, i.e., which returns only consistent models and terminates for every input. Thus, it conservatively approximates the orchestration problem. We have motivated the algorithm with its assistance in finding the reasons whenever it fails to deliver consistent models. Since this property is difficult to prove, we provide an evaluation in the following.

9.3.1. Goals and Methodology

The proposed provenance algorithm (see Algorithm 7.2) iteratively achieves consistency for subsets of the transformations. This is based on the idea that if the algorithm fails, we know that all but the last executed transformation were executed in an order that yields consistent models and only the last executed transformation introduced some decision such that no consistent orchestration could be found anymore.

We define the goal of our evaluation to show that the strategy helps transformation developers in finding the cause for a transformation network not to be able to find a consistent orchestration together with an according evaluation question and metric in Table 9.10. For meaningful results, evaluation scenarios need to comprise more than three metamodels. Failures especially occur due to cycles in the graph of transformations, and since a setting with three metamodels contains at most one cycle, there is no real value in the proposed orchestration strategy. Like we have discussed for the case studies in Section 9.2, such scenarios are difficult to find.

Most meaningful results for this goal and question could be achieved with a controlled experiment, in which participants are confronted with the information provided by the proposed strategy for a set of scenarios in which

Goal 5: (Orchestration)	Show that the orchestration strategy helps transformation developers to find the cause for a transformation network not being able to find a consistent orchestration.
Question 5.1: (Usefulness)	How far does the provenance algorithm improve the ability of identifying the reasons for a network not being able to find a consistent orchestration compared to an arbitrary strategy?
Metric 5.1.1:	*Considered transformations ratio: Ratio between the number of transformations to consider for finding a fault and the total number of transformations*

Table 9.10.: Goals, questions and metrics for orchestration evaluation.

it fails, as well as a control group to which the information delivered by other orchestration strategies is provided. Then, metrics like the time or the number of steps required to find the reasons for the transformation networks to fail could be measured and compared. Additionally, qualitative statements from interviews could be evaluated.

Since such an empirical evaluation requires high effort and, in particular, due to the absence of transformation networks to base the evaluation one, we decided not to perform such an empirical evaluation. Instead, we provide a scenario-based discussion that exemplarily shows the benefits of the proposed strategy in two defined but not yet implemented scenarios. We discuss two transformation networks with exemplary changes and how failures manifest with the proposed as well as alternative strategies and how this relates to the ability of identifying the reason for the failure. This allows us to evaluate the usefulness of the strategy in terms of Question 5.1 by measuring how many transformations have to be considered to identify a fault, according to the following metric:

$$\mathit{considered\ transformations\ ratio} = \frac{\#\ \mathit{of\ \ transformations\ to\ consider}}{\#\ \mathit{of\ \ total\ transformations}}$$

9.3.2. Scenarios

We consider two scenarios of transformations and changes to existing models that are to be kept consistent by the transformations. They represent

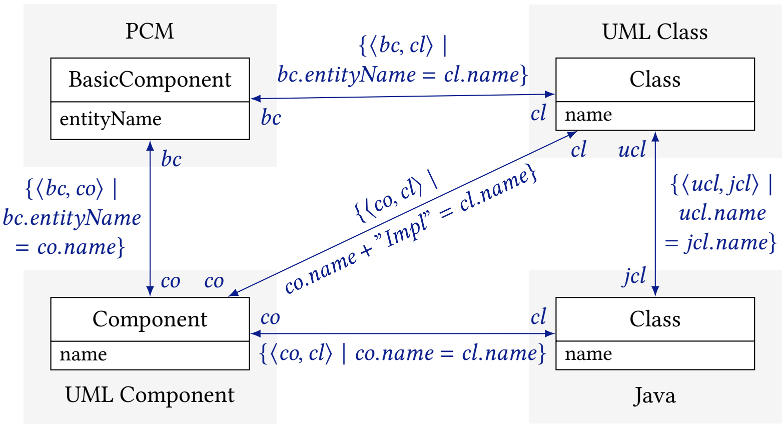


Figure 9.3.: Consistency relations between basic components in PCM models, components in UML component models, classes in UML class models and classes in Java models.

extensions of scenarios that we already considered within the last chapters. In both scenarios, our proposed strategy fails. In one scenario, no consistent orchestration can be found because of incompatible consistency relations. The other scenario contains a consistent orchestration. It may, however, take an arbitrarily long time to find it and no algorithm that is guaranteed to terminate can find it.

Incompatible Consistency Relations

We depict the first scenario in Figure 9.3. It consists of consistency relations between different representations of software components and their realizing classes. It comprises components in the PCM and in the UML as well as classes in the UML and Java. The consistency relations between them describe a simple one-to-one mapping of their names, such that for each class and component the according other elements with the same name need to exist. This is a simplification of the scenario that components have to be represented by classes but not vice versa. The only derivation from this mapping is the relation between UML class and UML component models, in which the class is specified to have the component name with an "Impl" suffix, according to the pattern proposed by Langhammer [Lan17].

Independent from the actual realization of consistency preservation rules that try to preserve consistency according to these relations, any application algorithm for those transformations will fail, because the relations are incompatible. In fact, the induced set of consistent model tuples contains only the empty models, as the relations cannot be fulfilled by any instances of the depicted classes. In consequence, adding any of the elements to a model will lead to an application algorithm that fails by either returning $\perp$, by returning inconsistent models, or by non-termination. While not terminating, either the “Impl” suffix is repeatedly added and removed from the elements to locally fulfill the individual consistency relations, or the suffix is repeatedly appended to newly created elements, leading to an infinite number of elements with arbitrary long names.

When failing, an application algorithm can be in an arbitrary execution state, in which any of the models can be inconsistent. The states in which the proposed provenance algorithm can fail can be divided into the following two categories.

1. If the first execution of the transformation between UML class and UML component models closes a cycle, i.e., two of the other transformations have already been executed such that the three form a cycle, the algorithm fails when adding that transformation. All transformations that were executed in advance are able to preserve consistency between all models, as they fulfill the consistency relations by adding the appropriate elements. When adding the transformation between UML class and component models, the transformations cannot find a consistent tuple of models anymore, which is due to the incompatibility of their consistency relations.
2. If the first execution of the transformation between UML class and component models does not close a cycle, e.g., because after adding a UML component it is the first transformation to be executed or because only the transformation between UML component models and PCM component models and/or the one between UML component models and Java code has been executed yet. Then the algorithm fails as soon as another transformation is executed that closes a cycle, such as the transformation between PCM component models and UML class model.

In either case, the algorithm fails as soon as the execution of transformations closes a cycle involving the transformation between UML class and component models. This does not necessarily mean that there is a fault in that transformation but that there is a fault within one of the transformations in the cycle closed by the added transformation, as consistency to all other transformations could be preserved. In fact, it is even impossible to say which transformation contains a fault, because it is unclear whether the consistency relation between UML class and component models is actually the one that should be adapted or whether, for example, the ones between PCM component models and UML class models and between UML component models and Java code should be adapted.

When the algorithm fails, the developer receives the information which addition of a transformation led to the failure together with the current state of the models. There is at least one consistency relation that is violated, which led to the abortion of the algorithm, and this relation must belong to one of the transformations within the cycle containing the fault. In consequence, the transformation developer only needs to consider the transformations in that cycle for finding the fault and, since he or she knows which consistency relation was violated, can restrict his- or herself to the elements concerned with the violated consistency relation. While in the example each metamodel pair only shares one consistency relation, in larger transformations more relations may be involved.

Regarding the number of transformations to consider for finding the fault, this means that at most three transformations need to be considered, as this is the largest simple cycle of transformations containing an incompatibility in its consistency relations:

$$\mathit{considered\ transformations\ ratio} = \frac{3}{5}$$

Even if the transformation between UML class and component models is the last to be executed, still only three transformations must be considered. Although there is an incompatibility in both simple cycles in which that transformation is contained, investigating one is sufficient, because the fault must be visible in both of the simple cycles involving the last executed transformation. Otherwise, the symmetric difference of the transformations in both cycles, which again forms a simple cycle, would also contain incompatible relations. This can not be the case, as consistency to these transformations was already achieved. In the example, if the cycle of transformations between UML component models, UML class models, and Java code did not contain

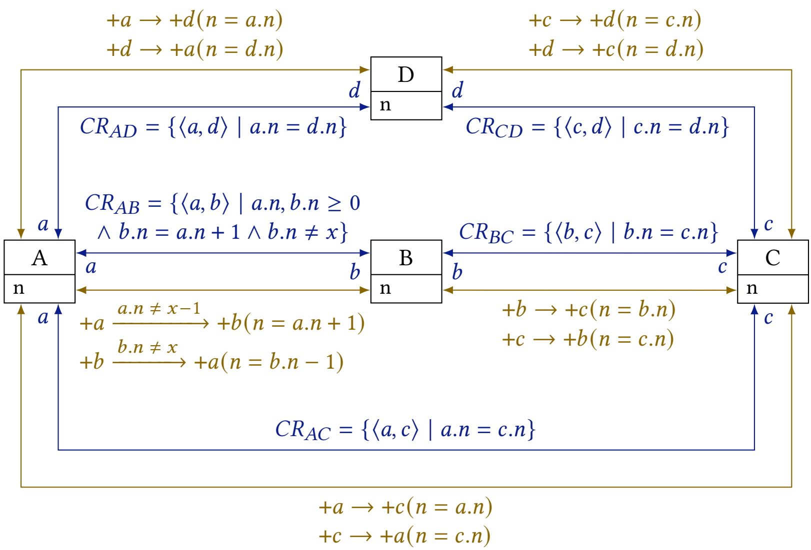


Figure 9.4.: Extension of the example in Figure 7.2 with consistency relations and associated transformations that require an arbitrary number of executions, depending on value x.

an incompatibility, either the consistency relation between UML component models and Java code or the one between UML class models and Java code would need to assume the “Impl” suffix as well. Then, however, the cycle of relations between all four metamodels would contain an incompatibility.

Orchestration Problem

Figure 9.4 depicts the second scenario. It is an extension of the abstract example depicted in Figure 7.2 as a demonstration for the non-existence of an upper bound for the number of necessary transformation executions in a transformation network. The extended example contains an additional metamodel, thus consisting of four metamodels, each containing one metaclass. Apart from that, it also contains consistency relations that require for each of the abstract elements A, B, C, and D other elements with the same value of n to exist. Only the relation between A and B requires the value n of B to be higher by one than the one of A, except for some value x of n, for which there must be no such element B for an existing A. Although these constraints make it difficult to find consistent models, they are actually compatible, as for each element there is a consistent model tuple containing it.

The depicted consistency preservation rules try to resolve this issue by adding elements to fulfill the consistency relations. This leads to the situation that adding an element A with value 1 at least $x - 1$ transformation executions are necessary (see Lemma 7.1). Thus, any application algorithm must either perform that many executions or fail returning $\perp$ or inconsistent models. When an algorithm performs that many executions, it can actually not be allowed to define any arbitrary execution bound, because the value of x can be arbitrarily high. Thus due to the orchestration problem, as discussed in Subsection 7.2.1, such a behavior leads to non-termination in other scenarios, which is not a competitive behavior compared to our proposed algorithm, since we want to avoid non-termination. In consequence, any useful application algorithm will fail in that example.

While an arbitrary application algorithm with an artificial termination criterion will fail in an unexpected state without any guarantee for usefulness of the state in which it fails to identify the reason for the failure, the provenance algorithm fails in the same cases and in the same way that we have already discussed for the first scenario. As soon as a transformation is executed that induces a cycles with the executed ones and contains the transformation between A and B, the algorithm will fail. In that case, the developer knows that the problem arises from the transformations in the cycle that was closed by the last executed transformation. This improves the process of finding the cause for the failure in the way as in the first example. In the worst case, the first cycle closed during execution containing the transformation between A and B is the one of length 4 between all metamodels. Thus, we have:

$$\textit{considered transformations ratio} = \frac{4}{5}$$

9.3.3. Discussion and Validity

The discussed scenarios give us specific insights about the usefulness of the proposed provenance algorithm, which we summarize in the following. In addition, we discuss threats to the validity of the results that especially arise from the construction of our scenario-based evaluation.

9.3.3.1. Insights

In the discussed scenarios, we have seen that using the provenance algorithm the number of transformations to consider for finding a fault that leads to a failure during execution is restricted by the length of the largest simple cycle of transformations that contains the faulty transformation. By construction of the algorithm, it fails as soon as a cycle of executed transformations is closed that contains a faulty one. In addition, by construction the fault can be found in each of the simple cycles of the already executed transformations that contain the last executed one. Thus, in the worst case the transformations in the largest simple cycle of transformations containing the faulty one need to be considered to find it. In consequence, as long as the transformation network does not only consist of one simple cycle of transformations, the algorithm does always ensure that not all transformations need to be considered in case of a failure. In fact, it ensures that in a network of n metamodels at most n transformations need to be considered.

This also shows that we can further improve the algorithm by determining a reasonable selection order for the transformations. Instead of choosing an arbitrary one to be executed next, cycles should be closed first, because this ensures that smaller simple cycles are closed early. As an example, consider the second scenario. If we first execute the transformation between A and B and then the one between B and C, it is better to then execute the one between A and C to close the cycle, as the algorithm then already fails. If the transformations to D are executed before closing that cycle, we first close a cycle of length 4 rather than one of length 3. Both lead to a failure, but we expect the effort to find the fault in the latter case to be lower.

Since we did not perform an empirical evaluation but only a scenario-based discussion, our finding only serve as an initial indicator for the usefulness of the provenance algorithm in terms of improving the ability to identify reasons for failing as asked by Question 5.1. Still, we found a criterion in the scenarios, which is induced by the construction of the algorithm, that limits the number of transformations that need to be considered to identify a fault. This is an improvement regarding any arbitrary other strategy, which, in the worst case, can require the investigation of all transformations. Whether or not this metric reasonably reflects usefulness of the approach does, however, remain a threat to validity until its validation in an experiment.

9.3.3.2. Threats to Validity

In the evaluation, we found a criterion that shows that the proposed algorithm improves the investigated metric in all cases. Still, there are threats to construct and external validity that need to be mitigated by further studies.

We assumed the number of transformations to consider to be related to the usefulness of the strategy in terms of the ability to find a fault. Whether this assumption holds is a threat to construct validity. To mitigate this threat, we did not only focus on the evaluation of that metric, but we also presented qualitative arguments and discussed further quantifiable improvements, such as the restriction of consistency relations to consider in case of a failure to identify the cause.

Finally, we have compared our proposed approach with an arbitrary strategy for transformation orchestration. In this comparison, we always guarantee an improvement in worst-case performance. There may, however, be another strategy that performs better or at least equal to the proposed approach in all cases. This can limit external validity of the results. We tried to mitigate this issue by systematically deriving a strategy for orchestration that performs better than other strategies in all cases with respect to a well-defined criterion. As discussed in Subsection 7.3.1, we have developed a simulator for evaluating different strategies, but unfortunately we found each strategy to be outperformed by at least one other strategy regarding their ability to find a consistent orchestration in at least one scenario. Thus, we do not expect another strategy to be systematically better than the one we proposed, but, in the best case, only to perform better in specific situations.

9.3.4. Limitations and Future Work

Limitations of the provenance algorithm especially arise from its focus on theoretic properties and the missing discussion and evaluation of its practical application. We discuss specific limitations in the following and derive opportunities for future work.

Evidence for Generalizability The most relevant limitation of the proposed algorithm concerns the validity of the evaluation results regarding the proposed properties of the approach. While statements on the correctness and well-definedness of the approach have been proven, its usefulness was only validated in a scenario-based discussion, which especially suffers from potential threats to construct validity, as it is unclear whether the metric we have investigated actually reflect usefulness of the strategy in terms of reducing the time and effort when identifying faults in transformation networks. Thus, we plan to perform a controlled experiment in which the information delivered by our approach and by other strategies are presented to different groups of developers. Evaluating how long they take to find and fix faults and how successful they are in both situations helps us to validate the expected properties and improve evidence of the results.

Well-Defined Design Property The provenance algorithm gives the guarantee of finding a consistent orchestration as long as the transformations fulfill the property of being reactive converging (see Definition 7.8). This property can, however, neither be easily guaranteed nor analyzed. We have argued why this is still a reasonable property, but a property that can at least be analyzed at design time to avoid failures during execution would still be beneficial. Such a property can, however, easily restrict expressiveness of transformations, as we have discussed in Subsection 7.2.4. Still, finding such a property would be a valuable contribution and thus serves as a starting point for future work.

Transformation Selection Order In the evaluation, we found that selecting transformations in an order such that smaller cycles of executed transformations are closed first may be beneficial, because it reduces the number of transformations that need to be investigated to find a fault whenever the algorithm fails. While the considerations in the evaluation scenarios indicate it to be reasonable, we want to systematically investigate such an order and, in the best case, prove its improvement in future work.

Holistic Application Process Finally, we have only discussed how our proposed approach supports a transformation developer in identifying faults in transformations. In practice, a failure may however not occur when a transformation developer tests a transformation network, which allows him to directly identify and fix the fault. Instead, a transformation network may be in productive use, thus a failure occurs when a user of that network applies the transformations to preserve consistency. Then, a holistic process is required for reporting and fixing such errors, which needs to define the responsibilities. Additionally, such a process will not be just-in-time, thus the project in which the transformation network is applied needs to be able to deal with the fact that consistency cannot be preserved for some time. Such a process is, for example, also part of the research in the Vitruvius project (see [Kla+21]), to which the results of this thesis contribute, and thus constitutes a general topic of future work. The author of this thesis also contributed to a group discussion in a Dagstuhl seminar that considered that topic [TK19].

9.4. Conclusions

In the presented evaluation, we have discussed and provided empirical evidence for several statements regarding the categorization of errors in transformation networks and our approaches for synchronization, analyzing compatibility, and orchestration to avoid such errors, which we could not prove. Arising from the assumptions that we made for this thesis and discussed in Subsection 1.3.2, our contributions and their evaluation have some general limitations, which we shortly discuss in the following together with a derivation of general topics for future work. We finally summarize the results of our evaluation.

9.4.1. Overall Limitations and Future Work

For the correctness of transformation networks, we have presented a formal notion based on a well-defined formalism and derived different properties of correct transformation networks. This thesis especially provides a general formalization of the overall problem and a division into smaller sub-problems, for which it provides individual contributions and insights. While we made some initial assumptions that lead to general limitations of our contributions, they also provide space for future work.

Binary Consistency As discussed in Subsection 1.3.2, we assume a development process in which modular transformations are developed and reused independently. In Chapter 4, we have then introduced our central formalism based on a modular notion of consistency, for which we defined correctness of transformation networks. We decided to focus on transformations that rely on a binary notion of consistency. While this is a limitation, since not every multiary consistency relation can be decomposed into binary ones [Ste20b], for most considerations we made this limitation is actually only for ease of understanding but without loss of generality. Thus most of our considerations and contributions also apply to networks of transformations of which each relates more than two models. Since we did not explicitly consider that case, however, we currently need to accept it as a limitation, until we validate whether and which statements generalize in future work. This also resolves the issue that our approaches can currently only be applied to relations that are denoted as *binary-definable* by Stevens [Ste20b].

Structural Consistency In addition, we restricted ourselves to structural consistency relations (see Subsection 3.1.2). We need to investigate how far our insights and approaches apply to behavioral and extra-functional consistency relations as well. In fact, there is no conceptual limitation in our formalism that prevents it from being applied to behavioral relations. A hypothesis from a Dagstuhl seminar [Cle+19] states that behavioral relations may be more likely to be multiary, whereas structural relations are more likely to be binary. That would reduce this limitation to the one regarding the restriction to a binary consistency notion and thus imply the same necessity for future work.

Concurrent Editing Finally, we assumed that a user only changes one model, for which consistency then has to be preserved. Thus, we do not consider concurrent edits to multiple models by one or more users. Although, from a conceptual point of view, networks of synchronizing transformations can also handle concurrent edits in multiple models, as the transformations need to be synchronizing anyway, the process of dealing with problems must be different. While failures that occur without concurrent user edits in different models indicate faults within the transformations, concurrent edits can also lead to failures just because conflicting changes were made and are thus invalid. These cases must at least be distinguished and potentially lead to the necessity of different processing. This topic requires further investigation in future work, also incorporating existing work on considering concurrent updates in single transformations [Xio+09; Xio+13].

9.4.2. Summary

In the preceding chapters, we have introduced a notion for correctness of transformation networks and identified three specific problems to be discussed in detail. We have proposed an approach to analyze compatibility of consistency relations, whose formal representation is proven correct and for whose practical realization we empirically validated correctness and completeness. Transformations must be synchronizing to be used in transformation network. We have derived properties that transformations which are specified in existing languages for bidirectional transformations need to fulfill, for which we haven proven that they ensure synchronization. In an empirical evaluation, we have shown that the proposed approach to fulfill these properties is correct and complete. Finally, we have discussed the orchestration problem of finding consistent orchestrations for transformations, for which we have proven undecidability. We have proposed an algorithm that conservatively approximates a solution to that problem, for which we have also proven correctness and completeness and validated usefulness in a scenario-based discussion.

In addition, we have analyzed what happens if correctness notions are not fulfilled. We have proposed a categorization of mistakes, faults, and failures, which assigns mistakes to different conceptual levels in the specification process of transformation networks and shows that specific failures can be avoided if certain mistake types are avoided. We found that mistakes due to missing synchronization can be avoided by construction of a single transformation without knowledge about the other transformations to combine it with. Mistakes due to incompatible consistency relations can be found by analysis, and other mistakes are only found by failures during execution. An empirical evaluation has shown that this categorization is correct. In particular, the evaluation has also revealed that most of the faults that are likely to occur in practice are due to missing synchronization and can thus be avoided by construction. Of the remaining faults, most are due to incompatible constraints and can thus at least be found by analysis at design time. This is a promising insight, because it fosters the independent development of transformations, as most failures can already be avoided without knowing about other transformations to combine it with. Thus, as a central takeaway, it is particularly important to ensure that transformations are synchronizing by construction.

Part III.

Improving Quality of Transformation Networks

10. Classifying Transformation Networks

In the previous chapters, we have discussed how correctness of transformation networks can be achieved under the assumption of independent development and modular reuse of the individual transformations. Artifacts of the software development process, and thus also transformation networks, have, however, further relevant properties than functional correctness. Other properties especially concern the quality of artifacts regarding several dimensions, as also defined by ISO standard 25010 [I25010]. For the operation of a piece of software, besides functionality also its performance, usability, reliability, and security are relevant, whereas for its development especially its maintainability and portability are of interest [I25010, Tab. 2].

These dimensions of quality properties are directly related to the stakeholders for which they are relevant. While most property dimensions are related to the operation of a system, which in our case is the transformation network, and are thus relevant for users, i.e., for the people developing a system whose artifacts are kept consistent with transformations (see Section 3.2), especially maintainability is important for those who develop and maintain a transformation network [I25010, Tab. 2]. Although all these properties are relevant and have to be considered when developing transformation networks, we explicitly put the focus on those that are relevant for developers of transformations and transformation networks (see Subsection 1.3.4). Thus, in the following, we particularly focus on properties regarding *maintainability* of a transformation network in addition to its functionality.

In our motivation in Chapter 1, we have derived several assumptions regarding the process of transformation network construction. In particular, we have identified independent development and modular reuse of transformations to be essential assumptions, which directly imply that consistency relations may be defined and preserved transitively and repeatedly across different paths of transformations, thus inducing a dense graph of transformations. Since different qualities properties highly depend on the topology of a transformation network, we aim to identify these dependencies and thus discuss which topologies of transformation networks should be distinguished, independent from the initial assumptions that we have made. We then discuss how these topologies influence quality properties and identify trade-offs between these properties, especially concerning functional correctness and reusability. Instead of assuming modular reuse and then deriving how to achieve functional correctness, as it was the goal of the previous chapters, we consider topologies with inherent correctness properties and investigate how to improve quality properties, such as their independent reuse.

This chapter thus constitutes our contribution **C 2.1**, which consists of two subordinate contributions: a discussion of quality properties and their manifestation in transformation networks; and a classification of transformation network topologies with a discussion about their impact on properties. It answers the following research question:

RQ 2.1: What are relevant properties and topologies of transformation networks and how are they related?

With the insights in this chapter, transformation developers and users become aware of further quality properties of transformation networks besides correctness. They understand how the topology of a network affects these properties and, thus, between which of them trade-off decisions for their improvement have to be made.

Parts of the contributions in this chapter have been published in previous work [Kla18]. This especially concerns the identification of general relations between topologies and quality properties of transformation networks as well as the implication of trade-offs between these properties.

10.1. Properties of Transformation Networks

The most essential property of transformation networks, which we have also considered in the last chapters, is correctness, or more precisely *functional correctness* according to ISO standard 25010 [I25010, p. 11]. In addition to its correctness, functionality can be regarded in terms of *completeness* and *appropriateness* [I25010, p. 11]. While completeness concerns the degree to which functions cover all intended objectives, appropriateness is the degree to which functions facilitate the conduction of tasks to achieve the intended objectives. In terms of a transformation network, completeness represents whether the network is able to preserve all consistency relations, which requires transformations for all existing relations to keep consistent to be defined. Since appropriateness especially concerns manual effort, it is not as relevant in a fully automated process. Appropriateness would especially be of interest if the user is involved in the consistency preservation process by clarifying its intent or making necessary decisions to adapt models for being consistent, which can influence how far the automation facilitates the process of consistency preservation. Thus, in addition to functional correctness, we also discuss functional completeness as a relevant property and relate it to our requirement of *universality*, as defined in Chapter 1.

In our work, we focus on properties of transformation networks that are relevant for their developers (see Subsection 1.3.4). Thus, in addition to functional properties of such networks, we especially consider properties regarding their *maintainability* [I25010, Tab. 2], which describe the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [I25010, p. 14]. Maintainability includes the properties *modularity*, *reusability*, *analyzability*, *modifiability*, and *testability* [I25010, pp. 14]. We have already covered the former two properties of modularity and reusability implicitly in our assumption of *modular reuse* as well as analyzability in the goal of *comprehensibility*. In previous work [Kla18], we have also discussed properties of transformation networks but without basing them on a common understanding defined by the mentioned ISO standard.

10.1.1. Correctness

According to ISO standard 25010 [I25010], functional correctness denotes to which degree a system, in our case a transformation network, provides correct results. We have intensively discussed this property in the previous chapters, starting with a definition of *correct results* in Chapter 4 and discussing how to achieve transformation networks that fulfill such a correctness notion in Chapter 5 to Chapter 7. Thus, we do not discuss this property again but emphasize its central importance for a transformation network to be useful, as an incorrect transformation network leading to models of a system description that are inconsistent will hardly provide relevant benefits.

10.1.2. Completeness

According to ISO standard 25010 [I25010], functional completeness describes to which degree provided functions cover all objectives. Applied to transformation networks, this means to which degree such a network can preserve consistency according to consistency relations, be they explicitly defined or only intended by a transformation developer. Completeness of the individual transformations as well as of the transformations are both covered by their notions of correctness (see Definition 4.8 and Definition 4.16). It does, however, assume an even broader notion of what we introduced as *universality* in Chapter 1. While we have introduced universality as the ability to process transformation networks of arbitrary topology, an even broader notion would require the applicability of transformation network to every project in which artifacts need to be kept consistent. Thus, it would first require that the artifacts to keep consistent are represented in a form that is required to define transformations between them. More precisely, the artifacts to keep consistent need to conform to some kind of modeling formalism, such as the one we have proposed in Section 3.3 based on the EMOF standard [MOF].

If the artifacts or, more generally, the models to keep consistent are not represented in a format conforming to such a modeling formalism, a metamodel for them needs to be defined, and their representation may need to be transformed into an instance of such a metamodel. This is especially the case for proprietary tools that do not use a common format to represent their artifacts. For many popular tools, however, metamodels based on the EMOF or Ecore have already been reverse-engineered, such as MATLAB/Simulink [HB13; Son+12; Arm+11]. In addition, the EMF as a popular modeling framework provides an importer for XML-based specifications of metamodels [Ste+09, pp. 86]. Tools, especially from engineering domains, often provide XML-based representations of their artifacts, such as the electronic circuit design tool EPLAN [Gis16] or the exchange format for automation system design in AutomationML [I62714]. Defining a metamodel for a specific modeling formalism, such as Ecore, and representing artifacts as models of it is always necessary when modeling tools for that formalism shall be applied, for which transformations are only one example. Frameworks for generating graphical editors or model analyses could be further tools to be applied [Kla+17]. Thus, such an integration of artifacts into model-driven processes is part of separate research.

In our research, we have also developed and proposed such an approach to integrate artifacts into model-driven processes [Kla+17; Kla+19a]. It is based on the insight that code often contains models implicitly. The tools, whose artifacts we want to keep consistent, usually have definitions of metamodels of their artifacts defined within their source code, but they are only represented as a simple structure of classes instead of am explicit metamodel according to some modeling formalisms. For example, Java graph libraries need to contain a metamodel for representing graphs, but this is usually just represented by a set of classes and not an explicit metamodel according to some modeling framework. This also applies to programming languages, for which parsers contain metamodels for their Abstract Syntax Tree (AST) representations. We have proposed an approach that makes these implicit metamodels explicit to apply modeling tools, such as transformations for consistency preservation, to them [Kla+17; Kla+19a]. Since this topic and especially the proposed approach is important for applying transformation networks but also has further, broader application areas, we do not further discuss it in this thesis but refer to our previous work for details about it.

10.1.3. Maintainability

We have identified maintainability as a dimension of quality properties with central importance for developers of transformations and transformation networks. According to ISO standard 25010 [I25010], maintainability includes modularity, reusability, analyzability, modifiability, and testability. We discuss for these properties how they manifest in transformation networks and especially how they are related to each other. We do not aim to measure these properties, which is why we do not propose specific metrics for them. For source code, it has been shown that it is hard to assess its quality, e.g., to measure modifiability in terms of a correlation with the number of defects [GFS05; PSM02], and that only few metrics provide a correlation to, for example, the number of defects. We only aim at identifying the influencing factors for these properties instead of a measure for them anyway, especially with respect to topologies of the transformation network.

Modularity: Modularity is the degree to which a program, and thus also a transformation network, is composed of components such that changes only influence a part of it [I25010, p. 14]. This property degrades when having multiple paths of transformations expressing the same consistency relations, as then these paths depend on each other and may be contradictory. Having such multiple paths can lead to incompatibilities (see Chapter 5) or situations in which no consistent orchestration of the transformations exists (see Chapter 7), and thus degrade modularity.

Reusability: Reusability is the degree to which assets, such as the single transformations of a transformation network, can be used in more than one system [I25010, p. 15]. In terms of a transformation network, reusability of a transformation is given if it is independent from the other transformations and can be used together with others in a different context. This conforms to our notion of independent development and modular reuse, given as assumptions in Chapter 1. Reusability profits from having all relations between the involved metamodels expressed explicitly, i.e., directly between each pair of metamodels and not only transitively across others. This leads to multiple expressions of the same relations transitively across different paths of transformations, but it allows subsets of the transformations to be used in a different context in which only a subset of the metamodels is used. For example, having the relation between PCM and Java expressed directly instead of only expressing it transitively across the UML enables its reuse in other system development scenarios in which the UML is not used at all. Thus, reusability degrades when modularity improves.

Analyzability: Analyzability is the degree to which the impact of a change can be assessed effectively and efficiently or to which defects can be identified afterwards [I25010, p. 15]. On the one hand, this is important for the single transformations, as analyzing the impact of a change especially concerns the intended change of the behavior of a transformation. That is, however, also a topic of dedicated research about transformation validation and verification [Cab+10; AW15; AZK17; Val+12]. On the other hand, this is important for the interplay of transformations, thus how a change to one transformation affects interoperability with the others. This is, again, directly related to the existence of multiple paths of transformations preserving consistency to the same relations, as it influences how many other transformations may be affected and potentially need to be updated due to the modification to one of them. Consequently, the more relations are preserved across multiple paths of transformations, the more transformations may be affected by a single change and introduce interoperability problems that may be hard to analyze (see Chapter 7). Analyzability is also related to the notion of comprehensibility that we have introduced in Chapter 1. The lower analyzability is, the harder it becomes for a transformation developer to comprehend what the combination of transformations actually does, how an intended change can be performed, and what its impact is. We have also used comprehensibility to motivate the design of our orchestration algorithm in Section 7.4, which is driven by the goal of easing the analysis of failures of the transformation network, analogous to analyzability. Thus, analyzability improves with modularity.

Modifiability: Modifiability is the degree to which a system can be modified without introducing defects or degrading quality [I25010, p. 15]. It is directly influenced by modularity and analyzability, as also stated by the ISO standard [I25010, p. 15]. In terms of a transformation network, this can include the adaptation of existing transformations or the extension of an existing network with further metamodels and transformations. The same arguments as for modularity and analyzability apply, and thus modifiability improves and degrades with modularity of the transformation network. For example, the complexity of adding a new transformation, which is covered by modifiability, depends on the number of transformations that already, and in particular transitively, preserve relations between the two metamodels related by the new transformation.

Testability: Testability is the degree with which test criteria can be effectively and efficiently established and evaluated by test cases for a product [I25010, p. 15], such as a transformation network. While there are many influencing factors for testability, such as encapsulation and coupling within the implementation, it is, again, also influenced by the number of transformation paths across which consistency relations are preserved. The more paths of transformations preserving the same consistency relations exist, the larger is the set of models to be considered and transformations to be executed for testing correctness of preserving consistency according to a certain relation. This increases complexity of the tests to perform. Testability is also highly related to the notion of comprehensibility that we have introduced in Chapter 1, as we have also discussed for analyzability. The higher the number of transformations that need to be executed to detect a failure, the more complex we can expect the process of identifying the causing mistake to be (see Chapter 8). Testability, just like analyzability and modifiability, thus improves with modularity.

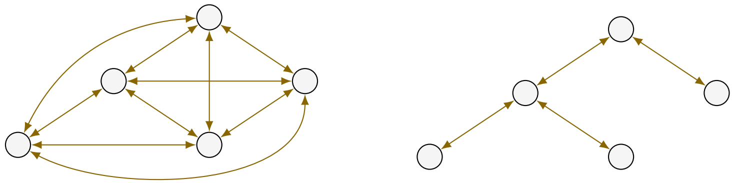
The discussion shows that the existence of multiple paths of transformations preserving consistency to the same consistency relations reduces modularity, modifiability, analyzability, and testability, while it improves reusability. This is because multiple representations of the same consistency relations induce dependencies, which reduce modularity, and can contain conflicts, which reduces modifiability. The increased complexity reduces analyzability and testability. Reusability is, however, improved, because relations are not only represented transitively. In the following, we identify relevant topologies of transformation networks that reflect the effects on properties of having multiple transformation paths preserving consistency to the same relations and discuss their impact on properties.

10.2. Topologies of Transformation Networks

Due to our assumption of *universality* (see Chapter 1), we have allowed arbitrary topologies of transformation networks in our approaches for achieving correctness of transformation networks. The topology of a transformation network does, however, directly influence how prone it is to incorrectness and also to the fulfillment of other quality properties, which we have introduced in the previous section. We consider the effects of a topology to different properties of transformation networks, for which we first discuss the extreme cases of topologies that have extreme effects on its properties.

10.2.1. Topology Categories

Transformation networks induce a graph of metamodels as nodes and transformations as edges. In general, this graph has an arbitrary topology, as there can be transformations between any pair of metamodels, and, in particular, there can be multiple paths of transformations between two metamodels in this graph. As we have discussed in the previous section, properties of transformation networks are especially influenced by the presence of multiple paths of transformations between the same metamodels. Thus, the density of the graph has gradual influence on the quality properties of the network. Two extremes of topologies contain the minimum and maximum numbers of paths between each pair of metamodels. They are given by complete graphs and trees, as exemplarily depicted in Figure 10.1. While complete



(a) Complete graph **(b)** Tree

Figure 10.1.: Examples for extreme topologies of transformation networks with five metamodels. Nodes depict metamodels and edges depict transformations. Adapted from [Kla18, Fig. 2].

graphs contain an edge between each pair of nodes, i.e., one transformation between each pair of metamodels, a tree contains only one path between each pair of nodes, i.e., only one sequence of transformations between two metamodels. We have already discussed the effects of these extremes in previous work [Kla18].

In a complete graph (see Figure 10.1a), each node is connected to each other by an edge. In consequence, each of the n nodes has $n-1$ edges to the other nodes, leading to a total of $\frac{n*(n-1)}{2}$ edges. This conforms to the number of transformations defined in a transformation network that induces a complete graph. In addition, the paths of transformations between two metamodels are given by paths of all lengths between 1 and $n-2$ involving all permutations of the remaining $n-2$ metamodels. This leads to $\sum_{i=0}^{n-2} \frac{(n-2)!}{(n-2-i)!} = \sum_{i=0}^{n-2} \binom{n-2}{i} i!$ transformation paths between each pair of metamodels.

In practice, the induced graph of a transformation network will, of course, usually not be complete but a graph of arbitrary density, in which there may be clusters of complete subgraphs. Imagine the development of an automobile, in which models from different domains, such as electrical engineering, mechanical engineering, and software engineering, are involved. While models within one domain may all be related by transformations, there may be specific interface models that are used to relate the models of one domain to those of the others, which avoids the necessity to have knowledge about the relations between all models across existing domain borders.

In a tree (see Figure 10.1b), there is only one path between each pair of nodes. Thus, a tree of n nodes has $n-1$ edges. A transformation network that induces a tree thus has a number of transformations reduced by a factor

Category	Property	Complete Graph	Tree
Functionality	Correctness	-	++
	Completeness	++	-
Maintainability	Modularity	-	+
	Reusability	++	-
	Analyzability	-	+
	Modifiability	-	+
	Testability	-	+

Table 10.1.: Effects of topology extremes on quality properties. "+" and "-" indicate whether a topology improves or degrades a property, "++" denotes inherent optimization of the property.

of $\frac{n}{2}$ in comparison to a complete graph and an even greater reduction in the number of transformation paths between two metamodels. This leads to significant advantages regarding interoperability of the transformations, which we categorize in more detail in the following.

A transformation network inducing a complete graph can naturally be achieved by expressing each consistency relation in a transformation. If two metamodels are not related at all, the according transformation does nothing. Defining a tree is, however, more complex, as it imposes severe restriction regarding the transformations in which relations have to be preserved to avoid having two paths of transformations between the same metamodels. In the following, we discuss the effects of these extreme topologies and derive which inherent property guarantees a specific topology can give.

10.2.2. Effects on Properties

We have discussed in Section 10.1 how the existence of multiple transformation paths between two metamodels affects quality properties of transformation networks. In the previous subsection, we have identified complete graphs and trees as two extremes of topologies of transformation networks that have particular effects on the existence of such multiple paths. These topology extremes have extreme effects on the quality properties of a network.

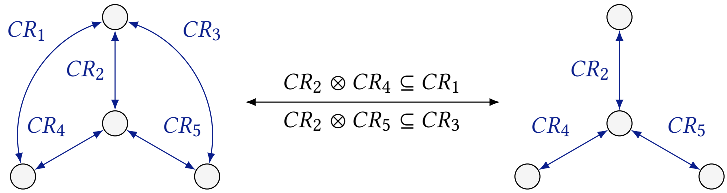


Figure 10.2.: Example for consistency relations in a graph that can be equally represented by consistency relations in a tree. Adapted from [Kla18, Fig. 3].

Table 10.1 summarizes the impact of topologies on quality properties. The classification is only based on the existence of multiple transformation paths between the same pairs of metamodels, as we have discussed in Section 10.1. There are, of course, more influencing factors that can improve or degrade these properties. In fact, we are particularly interested in properties that are inherently optimized by specific topologies, which are functional correctness and completeness as well as reusability.

Modularity, analyzability, modifiability, and testability all benefit from the absence of multiple transformation paths between the same metamodels, because the information about one relation is only located at one place, which can be a single transformation or a single sequence of them. But the information is not duplicated across several transformation paths. Since we expect a benefit from the absence of duplications for the mentioned properties, we classify them as improved by tree topologies and degraded by complete graphs. There are, however, further influencing factors that may mitigate this classification. For example, to achieve a tree it is necessary to express at least some of the relations indirectly across multiple transformations, as not each relation can be expressed directly. This can degrade properties like modifiability, as it gets more complicated to comprehend relations if they are defined across multiple transformations rather than in a single one.

Completeness and reusability are inherently given in networks inducing a complete graph. A complete graph of transformations allows to preserve consistency to any set of binary consistency relations, as the topology does not restrict between which metamodels transformations are allowed to be expressed. Trees, on the other hand, do not allow to express every set of relations, as we have already motivated in Section 1.2.2. If, for example, the PCM, the UML, and Java all share information pairwise, which cannot be expressed in instances of the third metamodel, there is no tree of transformations that preserves consistency for all this information. In general, of three metamodels there must always be one that is able to express the information shared between the other two to encode their consistency preservation in a tree of transformations. Transferred to model-level consistency relations (see Definition 4.1), this means that between three metamodels there must be a concatenation of two consistency relations that is a subset of the third. In that case, the third relation is subsumed by the concatenation of the others anyway and can thus be omitted. This situation is depicted in Figure 10.2.

In addition, reusability is given by complete graphs, because preserving consistency between two metamodels is always represented in a direct transformation between them, which can readily be reused. From a transformation network inducing a tree, only subtrees of transformations can be reused without loosing guarantees for consistency preservation. If, for example, PCM and Java models are kept consistent via the UML, it is not possible to reuse the (indirectly expressed) transformation between Java and PCM without reusing the UML. This significantly restricts reusability in tree topologies.

Correctness, on the other hand, is inherently given in networks inducing a tree topology. Between each pair of metamodel there is only one path of transformations. In consequence, there cannot be any incompatibility (see Chapter 5), as this requires multiple contradicting sequences of consistency relations encoded into transformations. In addition, transformations do not need to be synchronizing (see Chapter 6), as the situation that both models involved in a transformation have been modified is never given due to the missing situation of multiple transformation paths modifying the same models. Finally, only the orchestration of transformations (see Chapter 7) remains a challenge in such trees. Although there are no cycles of transformations that need to be orchestrated, and thus any topological order of transformations starting with the node representing the metamodel of the changed model may be selected, we have identified in Chapter 7 that it can be necessary to execute transformations multiple times, as they need to react to the changes performed by other transformations. This already occurs when two transformations are chained. Since this challenge does always occur when a transformation is able to change both involved models rather than only one of them, the only solution is to enforce transformations to only change one model, which may prevent relevant scenarios, as discussed in Chapter 7. The evaluation of our approaches for achieving correctness in Chapter 9, however, indicates that issues due to orchestration of transformations may not be that relevant in practice. Summarizing, apart from the discussed restrictions, this leads to inherent functional correctness as defined in Chapter 4. Thus, in a network that induces a tree, several severe challenges for correctness of transformation networks do not occur.

An actual transformation network will usually neither induce a complete graph nor a tree, although we have already discussed that complete graphs are at least easier to achieve. Thus, a network will not inherently optimize any of these properties but gradually optimize some of them, depending on the number of duplications of preservation for consistency relations within the transformations. This leads to a trade-off between different properties depending on the achieved topology. More duplications lead to higher completeness and reusability, whereas less duplications improve inherent correctness and also likely improve further discussed quality properties.

Although trees are not easy to achieve in practice due to the missing ability of transformation networks with such a topology to express all possible consistency relations, their inherent correctness guarantee is still interesting, as we have seen how difficult correctness is to achieve in networks of arbitrary topology in the previous chapters. In the following chapter, we thus identify and discuss how we can use this essential benefit of trees to construct networks that still provide a high level of completeness and reusability.

In fact, we have up to now discussed the topology of transformation networks at the level of complete metamodels and transformations between them. Transformations are, however, composed of rules that preserve consistency according to fine-grained consistency relations, such as the ones we have specified in Definition 4.18. Thus, we can even generalize the insights regarding topologies from complete metamodels and transformations to metamodel elements and fine-grained consistency relations, which then mitigates some of the drawbacks regarding completeness of trees. This conforms to the notion of *non-interference* defined by Stevens [Ste20b], which considers transformations to be non-interfering as long as they affect independent subsets of the metamodels and then can be executed in any order.

10.3. Summary

In this chapter, we have discussed which software quality properties, as defined in ISO standard 25010 [I25010], are relevant for developers of transformation networks. In addition, we have identified two extremes of transformation network topologies and discussed their impacts on quality properties. From this discussion, we were are able to derive necessary trade-offs between the properties induced by the topology of the network. We conclude this chapter with the following central insight.

Insight III.1 (Property Classification)

In addition to functional correctness of transformation networks, further quality properties can be relevant for developers and users of such networks. For developers of transformations networks, in particular functional completeness, i.e., the ability to apply transformation networks to any situation in which consistency between models needs to be preserved, and different aspects of maintainability, such as modularity, reusability, analyzability, modifiability, and testability, are important. Transformation networks induce a graph of metamodels and transformations between them that can, at one extreme, be a complete graph, in which each pair of metamodels is directly related by a transformation, and, at the other extreme, be a tree, in which each pair of metamodels is only related by one path of transformations. While networks inducing a complete graph inherently optimize completeness and reusability, those inducing a tree inherently optimize correctness. Although trees are particularly restrictive regarding completeness, and although in practice networks inducing a tree are thus hard to achieve, their inherent correctness guarantee makes them still interesting, as they avoid multiple challenges to achieve correctness.

11. Mitigating Trade-Offs with Commonalities

We have identified in the previous chapter that the topology of the graph induced by the metamodels and transformations of a transformation network directly influences several of its quality properties, such as functional correctness and completeness as well as maintainability in terms of modularity and reusability. The extreme topologies of complete graphs and trees imply extremes in the optimization or degradation of these properties, which induces a trade-off between these properties by means of the topology.

In Part II, we have focused on achieving correctness for networks of arbitrary topology, thus in general not inducing a tree but any graph topology that can be extended to a complete graph, which inherently optimizes reusability and completeness but requires high effort for achieving completeness. On the contrary, a tree structure, although not that easy to achieve, provides inherent correctness guarantees while reducing reusability and completeness (see Subsection 10.2.2). In this chapter, we discuss how a network having a tree topology can be constructed by introducing additional metamodels, such that correctness is still given but reusability and completeness is improved.

The idea of adding metamodels is not only a conceptual necessity to improve quality properties but also motivated by practical benefits. Since consistency relations define how common information is represented in several metamodels redundantly, we propose to represent this common information explicitly by means of additional metamodels. Then, only the manifestation of this information in the models to keep consistent has to be defined rather than an implicit encoding of common information in the consistency relations. These manifestation relations can, of course, again be represented by transformations. This way of specifying consistency with explicit metamodels representing common information can inherently lead to a transformation network with a tree topology.

This chapter constitutes our contribution **C 2.2**, which consists of four subordinate contributions: a discussion of how common information can be represented explicitly in dedicated metamodels and under which conditions this is reasonable; a proposal of the Commonalities approach to construct such metamodels and transformations for describing the manifestations of common information in the original metamodels; a discussion of the expected benefits of the approach, especially in terms of mitigating trade-offs between quality properties; and finally an outlook to processes of applying the approach and of combining it with other transformations. It answers the following research question:

RQ 2.2: How can topologies of transformation networks improve quality properties of transformation networks?

The insights in this chapter support transformation developers in constructing networks of correct, complete, and reusable transformations. It gives a different view on consistency and the possibilities to describe it besides consistency relations, which we expect to improve comprehensibility due to common concepts being represented explicitly rather than encoding them in consistency relations implicitly. The proposed construction approach for transformation networks inherently improves several quality properties by reducing the effort to achieve correctness of transformation networks as discussed in Part II and mitigating necessary trade-offs. It especially improves reusability and completeness in comparison to an ordinary construction of a network having a tree topology.

The initial idea for the contributions in this chapter has already been published [Kla18] as well as the proposed Commonalities approach with its expected benefits [KG19]. The approach along with a language that supports it, which we present in the subsequent chapter, has originally been developed in the Bachelor's thesis of Gleitze [Gle17], which was supervised by the author of this thesis.

11.1. Consistency of Common Concepts

In Chapter 1, we have motivated that models describing the same system share an overlap of information that leads to dependencies or, in particular, redundancies between the models. We have made these dependencies explicit

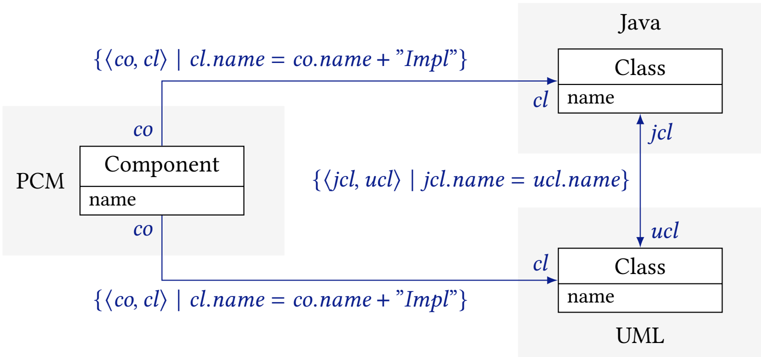


Figure 11.1.: Simple metamodel extracts for Java, the UML, and the PCM and consistency relations between them. Adapted from [KG19, Fig. 1].

by means of consistency relations. In the following, we discuss an alternative consideration of redundancies, as a special case of dependencies, by means of common concepts. We therefore provide an introductory example to be extended throughout the following considerations, explain the idea of *Commonalities*, and discuss in which cases it can be reasonably applied.

11.1.1. Introductory Example

We employ a running example from the case study introduced in Section 2.5 involving the PCM, the UML and Java. Consistency relations comprise the common and mostly one-to-one mappings between UML and Java as well as the mappings proposed by Langhammer et al. [LK15] to represent PCM architecture models in Java code and in UML class models.

In the following, we start with limited subsets of the metamodels, namely the one-to-one mapping between components in the PCM and classes in Java, whereby each component is mapped to a class but not vice versa, as depicted in Figure 11.1. Consistency relations require the existence of a class in the UML and Java for each PCM component having the component name with an “Impl” suffix by an according unidirectional consistency relation. In addition, the consistency relations require an equally-named UML class for each Java class and vice versa. We extend the example in the following sections to explain the introduced concepts.

11.1.2. Explicit Commonalities

In the given example, classes are redundantly represented in Java and the UML. This requires them to be kept consistent, which can, for example, specified by means of an according consistency relation. As an alternative, redundant classes in a Java and a UML model can also be considered representations of a *common concept*, more precisely the common concept of a class in general object-oriented design. Thus, rather than expressing this redundancy implicitly by means of a consistency relation and a transformation that preserves consistency to it, we propose to make the common concept explicit in an according metamodel and descriptions of how this concept *manifests* in Java and the UML. Then, instead of saying that each UML class should corresponding to a Java class and vice versa, we would say that classes in the UML and Java are both representations of the same concept of a class in object-oriented design.

We denote the actual metamodels that developers instantiate and want to keep consistent as *concrete metamodels*, whereas we denote metamodels that describe the concepts that such concrete metamodels have in common as *concept metamodels*. Figure 11.2 depicts the concrete metamodels UML and Java with their representations of classes. In addition, it contains a concept metamodel for object-oriented design, which contains the common concept of a class, shared by the UML and Java. We denote a single common concept, such as a class, as a *Commonality*. Further Commonalities in object-oriented design would be interfaces or methods. In general, a Commonality can be considered a metaclass with the specific semantics of describing the commonalities between elements of concrete metamodels. We say that an element in a concrete metamodel, such as a class in the UML and Java, is a *manifestation* of a common concept. The relation of a Commonality to these manifestations is denoted by a manifestation relation (*«manifests»*). In the example, the relations would especially define that each class manifestation conforms to a common class concept having the same name and vice versa, according to the relations in Figure 11.1.

In fact, these manifestation relations can be considered consistency relations that are preserved by ordinary transformations. Thus, in a first place the representation of common concepts in terms of explicit Commonalities introduces further effort, because it requires the definition of one metamodel and two transformations instead of a single transformation that relates the

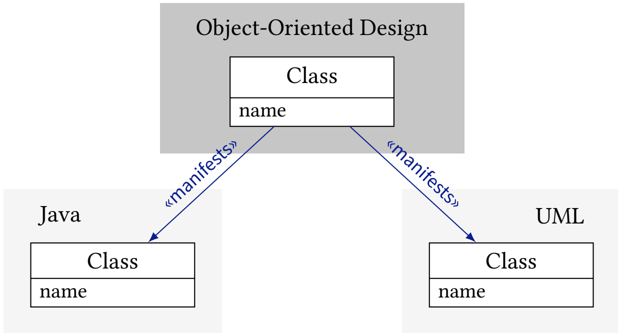


Figure 11.2.: Concept metamodel for object-oriented design with a `Class` Commonality and its relations to the concrete metamodels UML and Java. Adapted from [KG19, Fig. 2].

metaclasses directly. This drawback is, however, reduced by several benefits, which we discuss in Section 11.3, such as mitigating trade-offs between correctness and reusability as well as improving comprehensibility. Finally, such a specification can even reduce effort due to better scalability when adding further concrete metamodels to keep consistent. For example, if another object-oriented language such as C++ shall be kept consistent, no matter whether only with the UML or indeed even with Java, only the manifestation relation from Commonalities in the object-oriented design concept metamodel to C++ has to be added. This may only come along with some extensions of the concept metamodel for information shared between C++ and the UML as well as between C++ and Java that was not already shared between Java and the UML. This reduces the effort in comparison to defining both relations from C++ to the UML and to Java.

In general, a concept metamodel must contain Commonalities for all redundancies between the concrete metamodels to keep consistent. In a mathematical sense, this can be considered as the union of all pairwise intersections of the concrete metamodels. It can, however, not be precisely expressed as such, because elements may be similarly represented in the concrete metamodels, but they are not the same. One manifestation of the same Commonality may contain different information or encode it differently, such as using other units, than the others. This already illustrates the essential difference to approaches in which one central model unifies all information about a system, called a SUM (see Subsection 2.3.1), from which the models used by different tools are derived by projections. Such a SUM can be seen as the

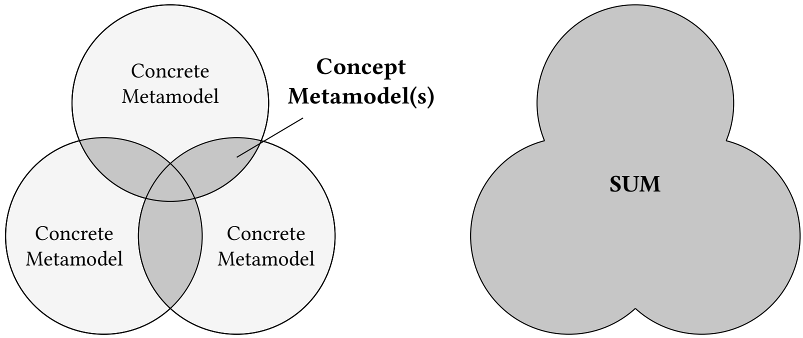


Figure 11.3.: Sketched comparison for the scope of contents of concept metamodels and SUMs.

union of all concrete metamodels, whereas concept metamodels represent the union of their pairwise intersections, as illustrated in Figure 11.3.

11.1.3. Consistency Specification Types

In Subsection 3.1.1, we have discussed the distinction of descriptive and normative specifications of consistency, which can be summarized as follows.

Descriptive Specification: Descriptive specifications describe consistency relations that are "naturally" given when two metamodels represent common concepts redundantly or with common or dependent properties. In that case, a notion of consistency already exists, formally or informally, to which the given specification must conform. This is, for example, the case for UML class models and Java realizing object-oriented design.

Normative Specification: Normative specifications prescribe consistency for metamodels for which no existing or common notion for consistency exists. This is especially the case if metamodels represent different abstractions or domains of a system, which have no implicit relations and for which different possibilities to relate them exist, such as an architecture description in the PCM and its implementation in Java.

While descriptive consistency relations between two metamodels are usually definite, such as those for object-oriented design between the UML and Java, normative consistency relations may vary depending on the project context. For example, several possible relations can be defined between an architecture description in the PCM and object-oriented design, such as the realization of each component as a class, as a bean in Enterprise Java Beans (EJBs), or as a complete project [Lan17].

Describing consistency by means of Commonalities and concept metamodels especially promises to be useful for descriptive consistency specifications, where a "natural" relation exists due to elements representing common concepts. It can, however, also be used to normatively define Commonalities in terms of a normative specification. A component Commonality can, for example, define that a component manifests as a component in the PCM and as a class in the UML and in Java, or, more generally, in an object-oriented design concept metamodel. This will, however, unlikely fit well for rather complex dependencies, such as a consistency relation requiring an implementation to fulfill some performance requirement. In such a case, the complexity is in the specification of the relation anyway, which would have to be replicated when defining a Commonality between performance requirement and the implementation. Finally, this conforms to our distinction of structural and behavioral consistency relations given in Subsection 3.1.2, in which the Commonalities fit well for structural relations, on which we focus in this thesis anyway.

In the following, we do not distinguish whether Commonalities are defined for common concepts that exist naturally or for those which are prescribed by the definition of concept metamodels and their Commonalities. We will see that even for normative specifications Commonalities can be reasonably defined. In Section 11.4, we also discuss how to combine ordinary transformations with the idea of concept metamodels.

11.2. The Commonalities Approach

We have motivated the idea of representing common concepts of different metamodels in terms of Commonalities in explicit concept metamodels rather than implicitly encoding them in direct consistency relations between the concrete metamodels. In the following, we discuss the specification of concept metamodels and the notion of manifestation relations in more detail. We also depict how further benefits can be generated by composing concept metamodels in terms of defining a hierarchy of them. We call this approach of defining and composing concept metamodels of Commonalities the *Commonalities approach*. The mitigation of trade-offs between quality properties as the central benefit of the approach is given by the inherent possibility to achieve a specific kind of tree topology, which we derive from the approach before discussing different options for its operationalization.

11.2.1. Concept Metamodels

The inherent benefits of the Commonalities approach are given by the definition of additional concept metamodels, across which consistency relations are expressed instead of defining consistency relations between the concrete metamodels. Conceptually, it is not that relevant how the structure of these concept metamodels and of the manifestation relations to the concrete metamodels actually is. Still, we discuss how elements can be represented as Commonalities in a concept metamodel and which relations beyond pure redundancies representing exactly the same information they may express.

Figure 11.4 depicts an extension of the example given in Figure 11.2. In addition to classes, it contains the representation of packages and associations. A package is represented as a dedicated metaclass in the UML, which references the classes contained in that package. Java, however, does not have an explicit representation of packages but encodes them into the package names specified within classes and, additionally, represents them in a folder structure in which the source code files of the classes are persisted. A concept metamodel used to preserve consistency between packages represented in the UML and Java must represent this information in any way such that changes in Java code can be propagated into a UML model to preserve their consistency and vice versa. To sketch an extreme, this could even be achieved with some string attribute in the concept metamodel that encodes this information in such a unique way that the necessary information for both instances of the concrete metamodels can be generated. Actually, a concept metamodel should represent such information in a reasonable structure, whose concrete characteristics have to be defined by the transformation developer. For packages, either the representation of Java as attributes of classes or the representation of the UML as a dedicated metaclass can be chosen. In the given example, we define packages in the concept metamodel as explicit metaclasses, as this makes the containment structure of classes in packages explicit. In addition, in the complete UML and Java metamodels

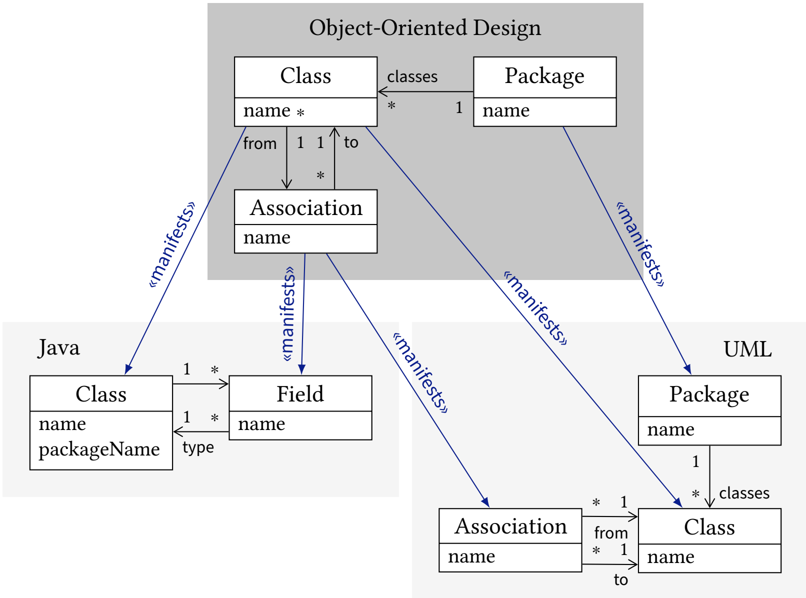


Figure 11.4.: Concept metamodel for object-oriented design with a `Class`, an `Association` and a `Package` Commonality and its relations to the concrete metamodels UML and Java with a different representation of associations as fields and packages as attributes of classes in Java.

packages are represented hierarchically, which is also easier to express as a relation between dedicated elements rather than their implicit encoding in the package names of classes.

Associations in the UML are used to define relations between classes. Each association references two classes, denoting from which class to which class the association is defined. Java does not provide an explicit representation of associations, which usually results in their implicit representation as fields of the class from which the association is defined and having the type of the class to which it is defined. In the example, we have chosen to represent an association in the concept metamodel explicitly. Fields can be related to further elements than associations in the complete Java and UML metamodels. Thus, having this distinction within the concept metamodel gives it more semantics. In addition, we have chosen that the class from which the association is defined references the association instead of having this reference in the opposite direction as in the UML metamodel. No matter whether this is beneficial or not, all information that is necessary to keep Java fields and UML associations consistent is represented by the concept metamodel. It shows that for a concept metamodel even a representation that differs from all its manifestations can be chosen.

As mentioned before, the only requirement to a concept metamodel is that it must be able to represent all information that is necessary for defining manifestation relations to the concrete metamodels, such that they are able to preserve consistency according to some consistency relation between the concrete metamodels. A general but rather informal rule, which has shown to be beneficial in the implementation of a case study for our evaluation, is to select the semantically richest among different representation options. In the example, we have thus chosen to represent packages explicitly instead of implicitly encoding them in package names of classes. This improves expressiveness of the concept metamodel and makes its information easier to use for defining manifestation relations without interpreting implicitly encoded information in each of these relations.

Instead of defining a new concept metamodel, it is, of course, also possible to use an existing metamodels as a concept metamodel. For example, the UML may be considered a suitable concept metamodel for object-oriented design. Doing so does not conflict in any way with the goals of the Commonalities approach. Such a metamodel can then either only be considered a concept metamodel whose instances are, by accident, also used by developers, or it can be considered both a concept metamodel and a concrete metamodel with a one-to-one manifestation relation between them. This is only a conceptual differentiation with no practical impact. Only for the approach operationalization, which we discuss later, it has to be considered whether instances of a concept metamodel may actually be relevant during productive use or not.

11.2.2. Composition of Concepts

We have so far discussed the idea of defining an additional concept metamodel to represent the common concepts of two or more concrete metamodels. For the depicted example for Java and the UML, it seems reasonable to group the common concepts in object-oriented design in such a metamodel. In Figure 11.1, we have also considered PCM components and their consistency relations to classes in the UML and Java. Although we could define a component Commonality for PCM components and classes in the UML and Java and consider this Commonality next to the class Commonality for UML and Java classes, we will likely not do so because of several drawbacks. First, a component Commonality does, semantically, not fit into the discussed concept metamodel for object-oriented design. Thus, the concept metamodel would have to be considered broader, potentially as one generic concept metamodel. Second, and more importantly, such a construction would introduce further redundancies, as the relation between classes in the UML and Java would be expressed via the two Commonalities for classes and components.

To solve the problem of a redundant specification of the relation between classes in the UML and Java via a class and a component Commonality, we could combine these two Commonalities to a single one, representing all necessary common information. If, however, further elements share information with classes and components, they also have to be merged into the same Commonality. In the extreme case, this could result in only having one large Commonality that is able to represent all related information. The manifestation relations would then have to make all kinds of distinctions based on the information given in such a monolithic Commonality.

An intuitive solution for the example scenario is to not consider classes in the UML and Java as manifestations of a component Commonality but to consider the class Commonality as a manifestation of the component Commonality. Then the relation between classes in the UML and Java is still represented across one specific class Commonality, whereas the manifestation relation of the component Commonality only has to be defined for the concept of classes instead of their concrete manifestations.

Abstracting from this concrete example, we propose to define hierarchies of Commonalities and concept metamodels, such that a manifestation of a Commonality does not have to be some classes of a concrete metamodel but can also be Commonalities of other concept metamodels. We depict such a structure for the example of classes and components in Figure 11.5. This allows to define one concept metamodel for each kind of concept, such as object-oriented design or component-based design, and then compose these concepts hierarchically. In consequence, it avoids the specification of a single concept metamodel that may become unmanageably large and again suffers from bad modularity, as it needs to combine information from as many concrete metamodels as have to be kept consistent.

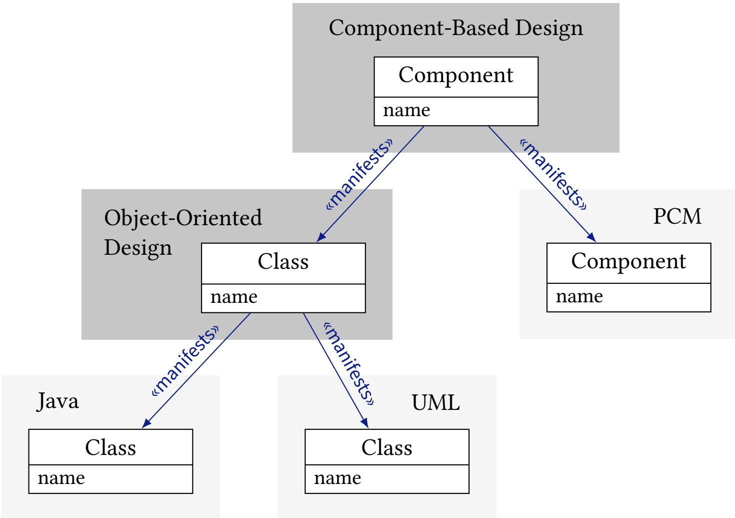


Figure 11.5.: Concept metamodels for component-based and object-oriented design and their manifestation relations between each other and to concrete metamodels for the example introduced in Figure 11.1. Adapted from [KG19, Fig. 3].

Since constructing such hierarchies induces a tree topology between the concrete and the concept metamodels, this construction suffers from the drawbacks regarding completeness, which we have already discussed in Subsection 10.2.2. Given two concrete or concept metamodels, there must be one that can be considered the manifestation of the other, or it must be possible to define a concept metamodel for them such that finally a tree of concrete and concept metamodels is achieved. First, this is an assumption and thus a limitation of the approach, for which we provide preliminary results regarding applicability in our evaluation in Chapter 13. Second, we further discuss these requirements regarding a tree structure in the following subsection to relax the restriction currently defined at the level of metamodels and consider a more fine-grained restriction at the level of metaclasses.

11.2.3. Tree Topology

In Subsection 10.2.2, we have discussed the benefits of a tree topology induced by the metamodels and transformations of a transformation network,

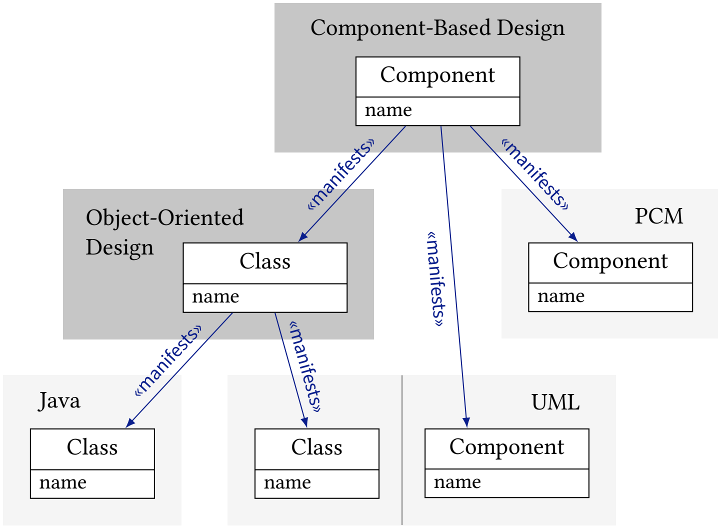


Figure 11.6.: Concept metamodels for component-based and object-oriented design and their manifestation relations between each other and to concrete metamodels for the example introduced in Figure 11.1 and extended with components in the UML. Adapted from [KG19, Fig. 3].

especially concerning inherent correctness. We have proposed the hierarchic composition of concept metamodels in the previous subsection to achieve a tree structure of manifestation relations in the Commonalities approach, which leads to a transformation network having a tree topology when realizing the manifestation relations as transformations.

This approach does, however, assume that such a tree topology of concept metamodels can always be achieved. Since we have up to now discussed the topology at the level of complete metamodels and transformations between them, it is easy to see that a tree cannot be achieved in many situations. This is always the case if one concrete metamodel contains concepts that are to be represented in multiple concept metamodels. For example, the UML contains concepts both from object-oriented design and component-based design, which easily conflicts with the goal of achieving a tree topology. Figure 11.6 depicts this example for classes and components in the UML. UML classes have a common concept with the concrete metamodels Java in object-oriented design, and UML components have a common concept with the concrete metamodel PCM in component-based design, which both, in turn, share a manifestation relation. This breaks the tree topology at the level of metamodels and transformations between them.

Although the bounds of metamodels are usually motivated by their necessity to fit for a specific purpose (see Subsection 2.1.1) and thus to represent specific concepts, metamodel bounds are, in general, arbitrary. Especially if metamodels have a rather general purpose, such as the UML or programming languages like Java, they may contain elements representing multiple different concepts, or the same elements may even be considered manifestations of multiple concepts. The former case leads to the situation that the elements of a metamodel may be separated by the different concepts they represent, thus virtually forming multiple metamodels. Usually, however, even elements representing concepts from different domains are still related, for example, by having the same super types like `NamedElement`, which makes their separation into different metamodels impossible.

The benefit of inherent correctness guarantees of transformation networks with tree topology arises from the fact that there are no two paths of transformations between the same metamodels, as discussed in Section 10.1. This is, however, already given if two paths of transformations affect disjoint sets of elements and thus do not interfere. Such a notion of *non-interference* has already been defined by Stevens [Ste20b], which specifies that two transformations changing the same model do not interfere if changing their execution order does not change the result. Since each transformation ensures consistency to its consistency relations and since the result is independent from the execution order of non-interfering transformations, it is guaranteed that the resulting models are consistent to both non-interfering transformations.

This informally stated notion of having all pairs of paths of transformations affect disjoint sets of elements, given, for example, by non-interference, conforms to our notion of *consistency relation trees* as specified in Definition 5.6 for proving compatibility of consistency relations. It defines that for each pair of concatenations of consistency relations either the left class tuples or the right class tuples must be disjoint, such that sequences of transformations preserving consistency to these relations affect disjoint sets of objects. In consequence, it is sufficient to ensure that the graph of consistency relations defined by the manifestation relations is a consistency relation tree to ensure compatibility of the network. Since Definition 5.6 assumes the consistency relations to be connected according to Definition 5.5, we may actually have multiple independent consistency relation trees, whereby *independent* means

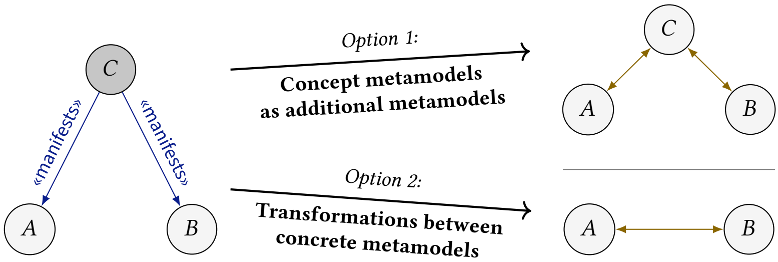


Figure 11.7.: Exemplification of alternatives to operationalize Commonalities specifications by using concept metamodels (such as C) as ordinary metamodels or by deriving direct transformations between the concrete metamodels (such as A and B) from them.

that the relations affect disjoint sets of classes. For reasons of simplicity, we relax that definition in our further discussions and consider multiple independent consistency relation trees as consistency relation trees as well. Due to the lack of multiple transformation paths affecting the same elements, it is also not necessary to ensure that transformations are synchronizing. Thus, even for this relaxed notion in comparison to trees at the level of metamodels and transformations, as depicted in Subsection 10.2.2, correctness guarantees for the transformation network are given.

Still, this relaxed notion represents a requirement for the Commonalities approach to provide specific benefits. We show at a case study in our evaluation in Chapter 13 that it is actually possible to achieve such a structure in practical scenarios, which serves as an indicator for its general achievability and thus the possibility to have inherent correctness guarantees when applying the Commonalities approach for preserving consistency of multiple models. Finally, the notion could even be further relaxed, as it must finally only be ensured that only one transformation path between two elements exists at runtime. Even if there are two possible relations defined in the transformations, it can be the case that further constraints ensure that at runtime only one path is relevant, because the constraints are mutually exclusive.

11.2.4. Operationalization

Up to now, we have discussed how to express consistency by means of concept metamodels with Commonalities and manifestation relations in the

Commonalities approach. To actually preserve consistency of instances of the concrete metamodels, such a specification must also be operationalized, such that executable transformations that can be applied after changes to these models are present or derived. We can distinguish the following two basic options for this operationalization, which are also depicted in Figure 11.7.

Concept Metamodels as Additional Metamodels: The concept metamodels are considered as ordinary metamodels and manifestation relations as ordinary transformations. Thus, we consider a transformation network of concrete and concept metamodels, whose instances are kept consistent by transformations for the manifestation relations.

Transformations between Concrete Metamodels: Concept metamodels and manifestation relations are only used as auxiliary specification artifacts from which direct transformations between the concrete metamodels are derived. For example, from the object-oriented design concept metamodel in Figure 11.2, a transformation between Java and the UML is derived.

The benefit of treating concept metamodels as ordinary, additional metamodels and the manifestation relations as transformations is easy achievability. No specific languages or generators are required to derive the necessary artifacts, but existing tools for defining metamodels and transformations can be used to define concept metamodels and manifestation relations that can be readily used to preserve consistency of their instances. A drawback of this approach is that it requires the management and persistence of additional artifacts, namely the instances of the concept metamodels, which are only auxiliary artifacts that should not be visible to the user. Hiding these artifacts can be achieved with an according framework, such that developers are still only confronted with the models of the tools they use. Such functionality is provided by tools like Vitruvius [Kla+21] (see Subsection 2.3.2) providing only views on instances of concrete metamodels.

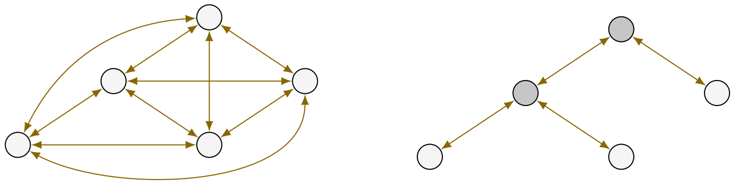
Deriving transformations between concrete metamodels from a specification of concept metamodels and manifestation relations benefits from not introducing further artifacts, such that a developer still only has to deal with instances of the concrete metamodels he or she is concerned with. This approach, however, suffers from reduced expressiveness, because not all multiary relations as expressed across additional concept metamodels (see [DKL18]) can be expressed by sets of binary relations and transformations preserving them [Ste20b]. In addition, it requires the implementation of generators that derive transformations from specifications of concept metamodels and manifestation relations.

Although with the second approach of deriving ordinary transformations the resulting transformation network contains cycles and does thus not provide correctness guarantees due to its topology, it still provides the guarantee due to the transformations being generated from a specification that ensures correctness. For example, since a specification of Commonalities cannot contain incompatibilities, the derived transformations cannot contain them either, as long as the generator produces transformations that actually preserve consistency conforming to the defined manifestation relations.

For the orchestration of the generated transformations, no matter whether they are defined between concept metamodels or derived between the concrete metamodels, it is still necessary to allow the execution of each transformation multiple times. Due to the situations identified in Chapter 7, in which it is necessary to execute transformations multiple times to “negotiate” a result and repeatedly react to the changes of other transformations, such a behavior is still relevant for the Commonalities approach. For example, propagating a class from Java across the object-oriented design concept metamodel and the component-based design concept metamodel to a component in the PCM can lead to further additions to the class as soon as it is identified as a representation of a component, which then needs to be propagated back to the class representation in Java. To support this, transformations should still be synchronizing and thus allowed to modify both involved models to support such situations that require this backpropagation of changes.

11.3. Expected Benefits

We expect several benefits from the Commonalities approach in comparison to defining ordinary networks of transformations. First, we claim to achieve better *comprehensibility* by making common concept explicit rather than implicitly encoding them in consistency relations. Second, we mitigate trade-offs between specific quality properties, in particular correctness and reusability, of the defined transformation network. Finally, it promises to reduce the specification effort at least in specific scenarios. While the improvement in comprehensibility is only a claim, we discuss the benefits of mitigating trade-offs and reducing specification effort in the following.



(a) Complete Graph

(b) Commonalities Tree

Figure 11.8.: Complete graphs as an extreme of transformation network topologies in comparison to the tree topology of a Commonalities specification. Nodes depict metamodels and edges depict transformations. In a Commonalities specification, leaves represent concrete metamodels whereas inner nodes represent concept metamodels. Adapted from [KG19, Fig. 4].

11.3.1. Improving Correctness and Reusability

We have discussed the benefits of the Commonalities approach regarding correctness guarantees in Subsection 11.2.3. This results from a transformation network defined with the Commonalities approach being intended to induce a tree topology. At the same time, a network defined by Commonalities also improves reusability, although the network forms a tree and reusability is actually a benefit of dense graphs as discussed in Subsection 10.2.2.

Figure 11.8 depicts the topology extremes of complete graphs and trees. In the tree topology of a Commonalities specification, the concrete metamodels are represented by leaves of the tree, whereas the inner nodes represent concept metamodels. This depiction is reduced to metamodels rather than metaclasses and Commonalities, as discussed in Subsection 11.2.3 for the tree structure, but would be the same if considered at the level of metaclasses.

In Subsection 10.2.2, we have discussed that reusability is improved in transformation networks inducing a dense or even complete graph, as a transformation exists for each metamodel pair and thus the transformations for any subset of the metamodels can be reused in other transformation networks relating a different set of metamodels. Having a tree topology, only subtrees can be reused, as otherwise consistency between some of the metamodels cannot be preserved because it was expressed transitively via transformations across metamodels that are not part of the subset to be reused.

This is, however, different for a network defined with the Commonalities approach. Although it forms a tree, the concrete metamodels to be reused in other networks are only leaves of that tree. Any subset of them can be reused without loosing transformations that preserve consistency between them by also reusing all concept metamodels on each path between two of the concrete metamodels to reuse. Since concept metamodels and their instances only represent auxiliary artifacts for describing consistency relations and their preservation, it is not a drawback that they have to be reused.

For these reasons, defining consistency with the Commonalities approach has the same benefits regarding correctness (and also other maintainability properties as discussed in Subsection 10.2.2) as defining a transformation network with tree topology, but at the same it improves reusability by allowing any subset of the concrete metamodels and the specification of consistency between them to be reused. The central limitation of the approach is regarding completeness, since the manifestation relations between metaclasses and Commonalities must induce a specific tree structure, namely a consistency relation tree according to Definition 5.6, to actually provide the benefits regarding correctness. It is part of our evaluation in Chapter 13 to validate the achievability of that property in practical scenarios.

11.3.2. Reducing Specification Effort

While the mitigation of the trade-off between correctness and reusability of a transformation network through the use of the Commonalities approach represents its major benefit, it can also reduce specification effort. This is achieved by the fact that each consistency relation must, in the best case, only be defined once, whereas in a transformation network inducing a dense or even a complete graph, there need to be redundant representations of the same relations if arbitrary parts of the network are supposed to be reusable.

Figure 11.9 depicts an the extension of the introductory example given in Figure 11.1, in which in addition to classes in the UML and Java a representation in C++ is added. In case of a transformation network, the relation between C++ and both Java and the UML needs to be defined. Using the Commonalities approach, only an additional manifestation relation to the concepts already defined in the object-oriented design concept metamodels has to be specified. In general, if n metamodels share common concepts, adding an n-1-th metamodel requires n transformations to be defined in ordinary networks, whereas the Commonalities approach, in the best case, only requires one addition manifestation relation to be defined.

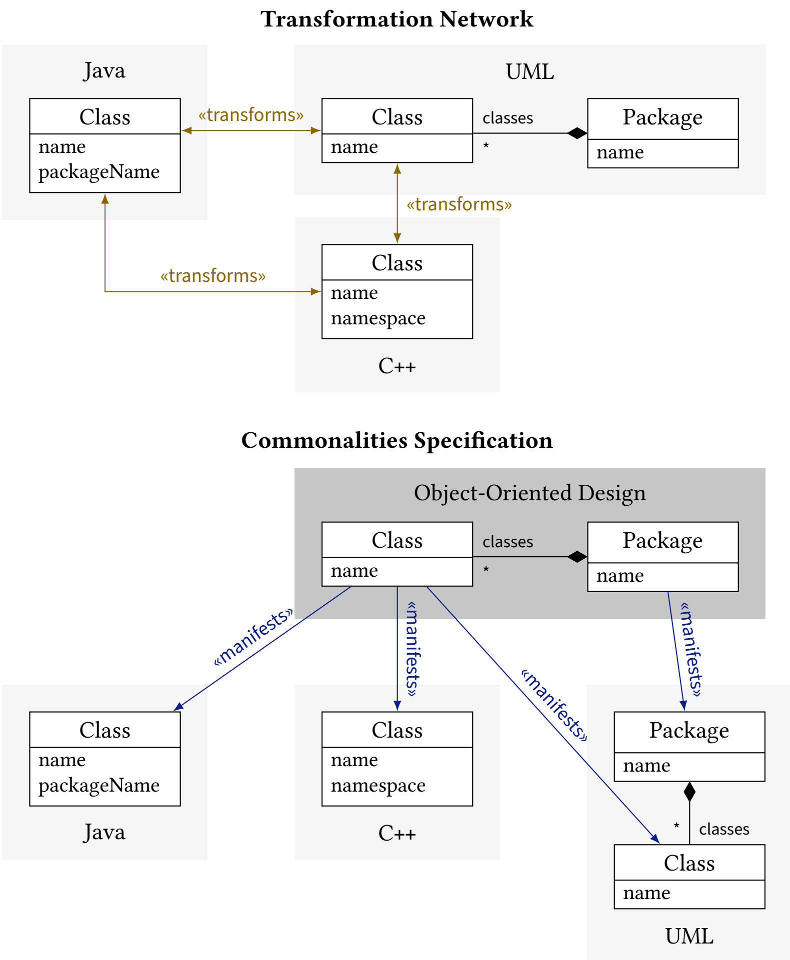


Figure 11.9.: Example for the number of defined relations with ordinary transformation networks and the usage of concept metamodels with the Commonalities approach.

The best case is, however, only achieved if the concept metamodel already contains all information shared between the concrete metamodel to be added and the ones for which the manifestation of the Commonalities in the concept metamodel is already defined. This is due to the already discussed fact that, informally speaking, the concept metamodel needs to represent the union of all pairwise intersections of the concrete metamodels. Thus, it will usually be necessary to also extend or adapt the concept metamodel and define or modify manifestations in the other concrete metamodels as well. For this scenario, a language that combines the specification of each Commonality with its manifestation relations, as we propose in Chapter 12, provides further benefits, as a modification or extension of a Commonality can be performed along with adaptations of the existing manifestation relations at one place.

In addition, applying the Commonalities approach may produce higher initial effort for the first consistency relations. For two metamodels to keep consistent, one concept metamodel and two manifestation relations have to be defined instead of only a single transformation in case of directly relating the two metamodels. This initial effort amortizes only if enough further concrete metamodels are kept consistent via the same concept metamodel.

The initial specification effort can, however, also be reduced by providing a specific language to define Commonalities that combines the definition of manifestation relations with the definition of its Commonality, such that the specification becomes nearly as concise as it would be if defined as a direct consistency relation between two metamodels. We propose such a language in Chapter 12 and discuss this benefit in Subsection 12.2.7.

11.4. Application Processes

The application of the Commonalities approach requires a process for defining them as well a concept for combining them with other specifications of transformations. In a specification using the Commonalities approach, the concept metamodels and manifestation relations are not as independent as they are supposed to be in the definition of an ordinary transformation network forming a dense or even a complete graph. Due to the necessity to relate all elements only via one transformation path, even if Commonalities are separated into concept metamodels by concerns and composed hierarchically, the developers must ensure that such a structure is achieved. We thus subsequently discuss different options how Commonalities can be defined.

We have identified in Subsection 11.1.3 that the Commonalities approach is well-suited for structural and “natural” consistency relations rather than arbitrarily complex and, in particular, behavioral dependencies. We thus discuss options for combining a Commonalities specification with other specifications, in particular ordinary transformations.

11.4.1. Defining Commonalities

We have discussed in Subsection 11.2.2 how Commonalities and the concept metamodels encapsulating them can be composed hierarchically. This allows to separate Commonalities by concerns, i.e., by the concepts they belong to, and fosters independent development and reuse of concept metamodels.

The Commonalities approach does, however, only provide an essential benefit regarding guaranteed correctness of the resulting transformation network if the manifestation relations specify consistency relations that form a consistency relation tree (see Subsection 11.2.3). Thus, Commonalities and their concept metamodels must be composed in a way that such a structure is achieved. This can, in the worst case, require all concrete metamodels to define consistency between and the according relations to be elicited a priori and thus conflict with our independent development assumption.

An intuitive process to define Commonalities is a bottom-up approach. Developers select concrete metamodels that share common concepts and are, by custom definition, most related among the concrete metamodels to define consistency between, and they define a concept metamodel of Commonalities between them. Then, they iteratively choose concept metamodels, and potentially also concrete metamodels, that share further higher-level commonalities and define an according concept metamodel for them. This ends up in a hierarchy of concept metamodels.

Since finally instances of the concrete metamodels shall be kept consistent, it is important to always consider the information represented in the concrete metamodels, even if consistency is defined between concept metamodels, i.e., at a higher level in the hierarchy of concept metamodels. Consider the running example of classes in the UML and Java as well as components in the PCM. We may define an object-oriented design concept metamodel with Commonalities between the UML and Java as well as a component-based design concept metamodel with Commonalities between object-oriented design and the PCM, as depicted in Figure 11.5. If these concept metamodels are defined in a bottom-up manner, i.e., first defining the object-oriented design concept metamodel and afterwards the component-based design concept metamodels, it is not sufficient to only consider the information represented in the object-oriented design concept metamodels for defining their Commonalities. This metamodel only contains the Commonalities that are relevant for object-oriented design, but for the relation to component-based design, further information that is only present in one of the concrete metamodels may be relevant. For example, Java contains a definition of behavior in terms of method bodies, which is not represented in the purely structural UML class models. Thus, the object-oriented design concept metamodel does not represent this behavioral information, as it does represent a Commonality. The PCM, however, also has an abstract representation of behavior used for predicting the system's performance, which needs to be kept consistent with the precise behavior specification in Java. Thus, the component-based design concept metamodel must either have an additional manifestation relation to Java for the behavioral information, or the object-oriented design concept metamodel must also contain behavioral information although not being a Commonality between the concrete metamodels it represents.

In general, this problem occurs because concept metamodels are supposed to represent the unions of all pairwise intersections of their concrete metamodels, as those represent the Commonalities that have to be kept consistent. Information that is unique to one of the concrete metamodels is not represented in the concept metamodel but may be relevant for further concepts and thus the relations to define to them. A first general solution would require a concept metamodel to contain the union of all information in the concrete metamodels rather than the union of their pairwise intersections. This does, however, not conform to the purpose of concept metamodels to only describe Commonalities. It leads to large and complex concept metamodels and thus also to high effort, because for each concrete metamodel a transformation, in terms of a manifestation relation, of all its information to a concept metamodel would have to be defined. In addition, the topmost concept metamodel of the hierarchy would inherently contain the union of information defined in all concrete metamodels, thus representing a SUM metamodel, i.e., a single metamodel that is capable of representing all information to define one system (see Section 2.3). In consequence, it would be sufficient to only manage an instance of that topmost concept metamodel, representing the SUM metamodel, and to consider the instances of all other concept and concrete metamodels as projections from the instance of that central metamodel, according to Atkinson et al. [ASB10].

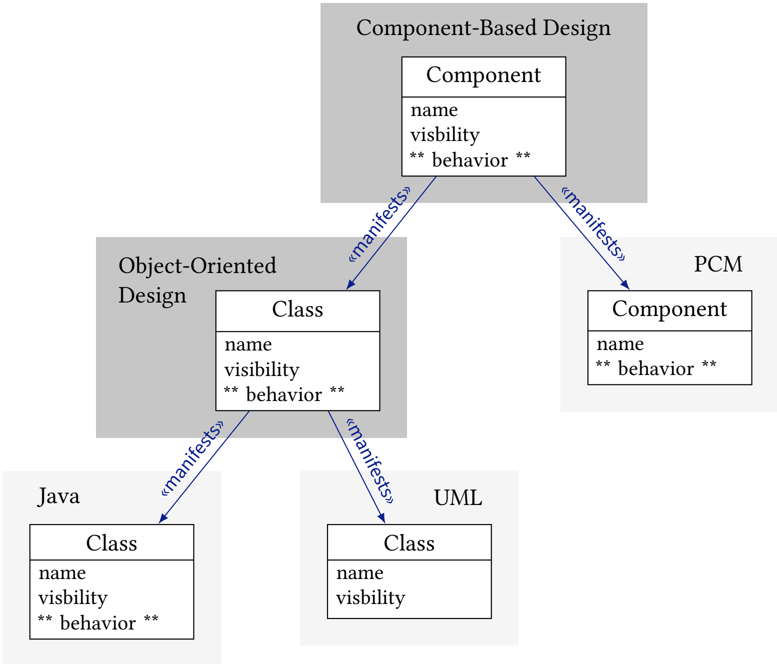


Figure 11.10.: Example for a hierarchy of concept metamodels and their Commonalities, in which concept metamodels represent the union of information in their manifestations. Behavior of classes and components is considered any, not further specified kind of behavioral information.

For the example in Figure 11.5 depicting hierarchic concept metamodels for classes and components, we derive an extension according to the discussed scheme in Figure 11.10. It additionally contains visibilities for classes and any kind of not further specified behavior description in Java classes and PCM components. Both concept metamodels contain the union of information in their manifestations, such that the component-based design concept metamodel contains all information represented in all metamodels. In consequence, the component-based design concept metamodel represents the visibility of classes in object-oriented design, although it is not relevant for components and is not kept consistent via that concept metamodel.

The previous considerations assume a kind of strict layered architecture (see [Bus+96]) in which the manifestation relations induce a tree between the metamodels. Thus, no manifestation relation bypasses a concept metamodel to whose Commonalities additional manifestation relations are defined. Referring to a non-strict layered architecture, another solution would be to allow manifestation relations to the manifestations of concept metamodels to which further manifestation relations are defined. For example, the component-based design Commonalities may have manifestation relations to elements in Java and the UML in addition to manifestation relations to the object-oriented design concept metamodels, which in turn has manifestation relations to those concrete metamodels. A drawback of this solution is that it can likely prevent achieving a tree structure. Considering a class in Java as a manifestation of a component in component-based design as well as a class in object-oriented design, which in turn is a manifestation of a component in component-based design, would violate the definition of a consistency relation tree, thus not giving guarantees regarding compatibility.

Figure 11.11 depicts this solution for the already discussed example. The concept metamodels contain only the information relevant for the Commonalities they represent. The additional manifestation relation between components of the component-based design concept metamodel and classes in Java induce a violation of a tree structure. Although behavior may actually be represented in terms of method bodies represented as separate metaclasses in Java, still consistency relations defined by the manifestation relations between Java and the object-oriented design concept metamodel would include both classes and methods, as methods do not share an isolated consistency relation but only in the context of the class they belong to.

A third option is to construct a concept metamodel not only driven by the Commonalities shared between its manifestations but also by its Commonalities with other metamodels. Thus, whenever a concept metamodel is used as a manifestation of another concept metamodel, it may be extended by the information from its manifestations required for the Commonalities in another concept with other metamodels. For example, as soon as the object-oriented design concept metamodel is considered as a manifestation of component-based design, its manifestations, namely Java and the UML, are checked for Commonalities with component-based design that are not yet considered Commonalities regarding object-oriented design. This could be a description of method bodies in Java to keep consistent with the behavior specification in the PCM. If consequently followed, such an approach would result in concept metamodels not only representing the union of the pairwise intersections of the manifestations, but the union of the pairwise

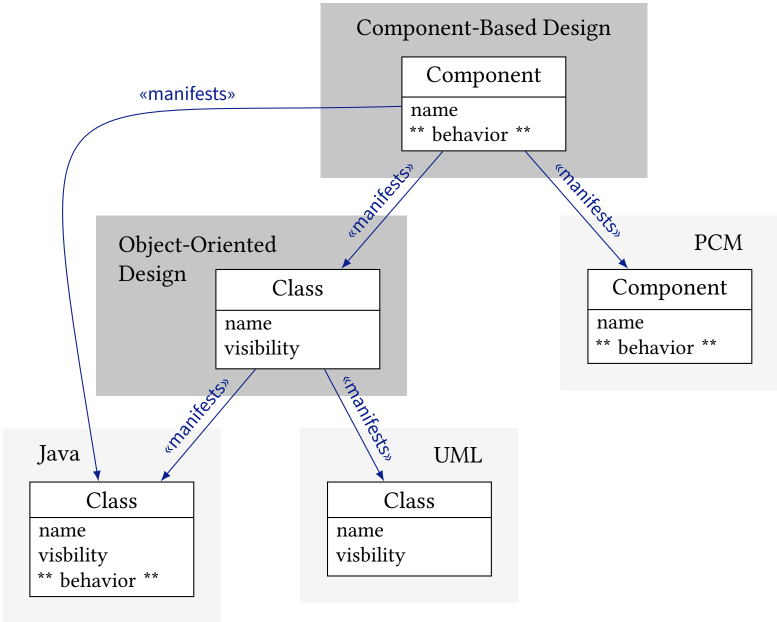


Figure 11.11.: Example for a hierarchy of concept metamodels and their Commonalities, in which Commonalities may have several manifestations inducing consistency relations that do not form a tree structure. Behavior of classes and components is considered any, not further specified kind of behavioral information.

intersections of their manifestations with all other concrete metamodels to be kept consistent. This promises to lead to concept metamodels that are significantly smaller and more precise than the union of all metamodels as in the first option, but it would still allow to achieve a tree structure, which is why we propose to use this option. This approach is comparable to the situation in which a further manifestation shall be added, like we exemplarily discussed for adding C++ as a manifestation of the object-oriented design concept metamodel in Subsection 11.3.2.

The application of this option to the already discussed example is depicted in Figure 11.12. In this solution, still a tree structure between the metaclasses and Commonalities is given and the concept metamodels are still restricted to the information in the manifestations and, in addition, the information of the manifestations necessary for the concept metamodels of which they are

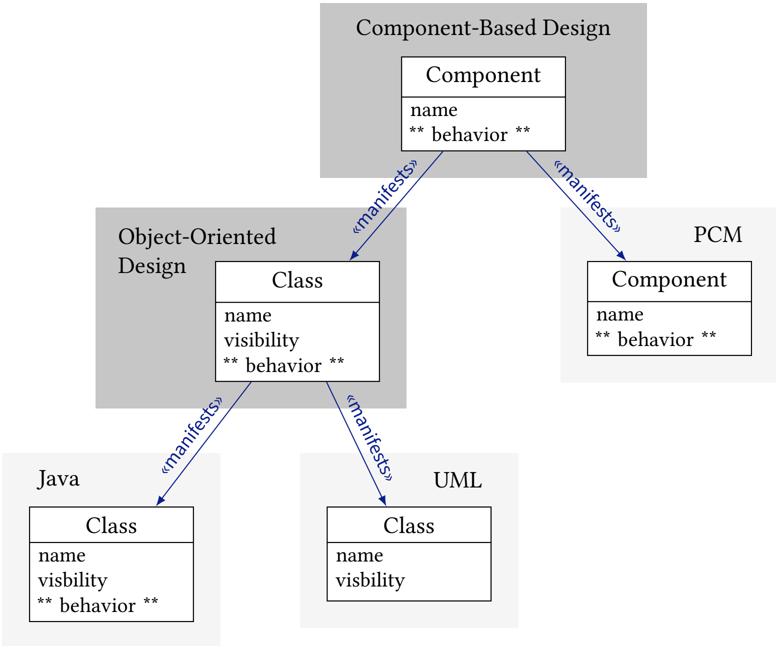


Figure 11.12.: Example for a hierarchy of concept metamodels and their Commonalities, in which Commonalities represent information necessary for the concepts they are manifestations of in addition to the information shared by their manifestations. Behavior of classes and components is considered any, not further specified kind of behavioral information.

manifestations. This is why the object-oriented design concept metamodel contains information about the behavior of classes and components although the UML and Java do not share behavioral concepts, but the component Commonality for component-based design does not contain the visibilities of classes as in the first option of representing the union of all information in the manifestations.

Finally, it is still an open question how problematic the actual dependencies in practical scenarios are. Potentially, only subsets of few metamodels are highly related and share large parts of one or more concepts, and the relation to other such subsets is only given across one metamodel or one concept. This could be seen as a graph of cliques, in which some metamodels are highly related whereas the relation to others is rather loose. In that case, it can be reasonable to define relations in these cliques by means of Commonalities and then define the loose relations to other cliques by means of an ordinary transformation, as we discuss in the subsequent section. We derive first insights on the achievability of the required tree structure for Commonalities in our evaluation in Chapter 13, but further evidence if one of the previously discussed strategies can be reasonably applied has to be gained in larger studies in practical scenarios with more metamodels of more tools.

11.4.2. Combining Commonalities

We have up to now discussed how to construct concept metamodels and manifestation relations in terms of the Commonalities approach such that the topology of the defined relations fulfills the definition of a consistency relation tree to achieve inherent guarantees regarding correctness of the transformation network. We have also derived how the Commonalities approach improves reusability in comparison to the construction of a transformation network with tree topology out of the concrete metamodels. Nevertheless, the approach has at least two limitations, which we have already identified. First, it lacks completeness, as it requires a specific topology of consistency relations to be achievable, which is likely to become more complex the more metamodels are involved. Second, it only fits well for structural relations in which commonalities can be described or prescribed.

In consequence, to improve applicability of the approach, it should be applied for subsets of metamodels that inherently share commonalities, comparable to the cliques mentioned before, which are suited to be described with the proposed approach. These specifications should then be combined with other consistency specifications, be they defined with the Commonalities approach or with ordinary transformations. Such a combination would restrict the size and complexity of a hierarchy of Commonalities and could foster reuse of consistency specifications for specific concepts in different context, as motivated by our assumptions of independent development and modular reuse as well as the process proposed in Section 3.2.

To preserve the benefits of a Commonalities specification, it can be combined with other specifications, be they ordinary transformations or another Commonalities specification, by considering any of the other metamodels as a manifestation or a concept metamodel of one of the concept metamodels of the Commonalities specifications. This preserves the tree structures of the

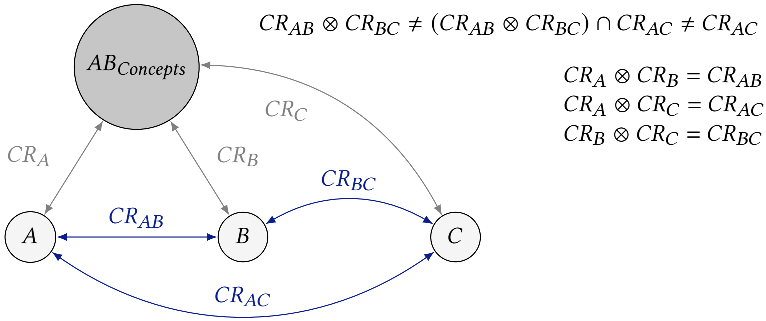


Figure 11.13.: Example for a concept metamodel $AB_{Concepts}$ to replace a consistency relation, and the replacement of ordinary consistency relations to the concrete metamodels with one to the concept metamodel. Adapted from [Kla18, Fig. 5].

Commonalities specification and its benefits. Consider the generic example in Figure 11.13 with three metamodels, a concept metamodel for two of them, and consistency relations between them, which are considered model-level consistency relations according to Definition 4.1 for reasons of simplicity. The consistency relation CR_{AB} between metamodels A and B is expressed by a concept metamodel $AB_{Concepts}$ and consistency relations for the according manifestation relations CR_A and CR_B. In addition, the metamodel C shares consistency relations with both other metamodels. To preserve reusability and the necessary tree structure, these consistency relations CR_{AC} and CR_{BC} should be described in terms of a consistency relation CR_C to the concept metamodel. This does, however, require the concept metamodel to contain all information that is necessary to preserve consistency between C and the two others, as described with the required relations in Figure 11.13. In contrast to the scenarios discussed in the previous section for how to define concept metamodels and which information to put into them, if C is a part of a different consistency specification to combine the Commonalities specification with or if the Commonalities specification covers more than two concrete and one concept metamodel, this can require an arbitrarily complex adaptation, which may not even be possible if modular reuse is desired.

To improve such a combination of specifications, virtualization concepts as known from OSM [ASB10] (see Subsection 2.3.1) and the Vitruvius approach [Kla+21] (see Subsection 2.3.2) can be applied. Their idea is to encapsulate metamodels and their instances behind a facet of views and to

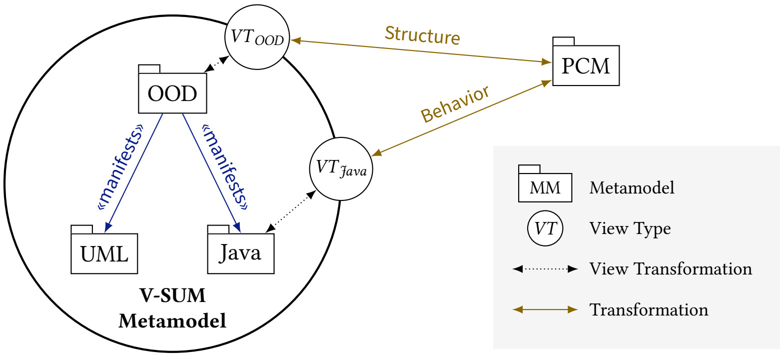


Figure 11.14.: Example for the combination of a Commonalities specification for object-oriented design (OOD) with PCM by encapsulation into a V-SUM metamodel.

enable access to the actual models only via these views. Views are projections of the encapsulated models, i.e., they derive all information from the models and potentially aggregate them or arrange them differently. The metamodels of these views are called *view types*. While those approaches were originally designed to provide a well-defined interface through views for developers and internally ensure consistency of the persisted artifacts by either avoiding or managing redundancy, they can also be used as an interface for consistency preservation. In the Vitruvius approach, a so called V-SUM is composed of models and rules for preserving their consistency, whose contents are exposed by views to be modified by developers.

Consider the example depicted in Figure 11.14. It comprises the Commonalities specification for Java and the UML using a single concept metamodel for object-oriented design. This consistency specification by means of Commonalities is encapsulated into a V-SUM, which exposes the Java code via a Java view and the object-oriented structure represented in instances of the concept metamodel as an object-oriented design view. These two views are then related to the PCM by means of ordinary consistency relations and transformations preserving them. The relations between metamodels and view types can, again, be considered ordinary transformations. Thus, the defined transformation network would actually contain cycles, such that it does not benefit from the Commonalities specification within the V-SUM in terms of correctness. If we only consider the V-SUM itself, it does, however, still have a tree structure, so if only one of the views is modified at the same time, it provides the benefits that we have discussed for a Commonalities specification in Section 11.3. In addition, views of a V-SUM by now assume that only one of them is changed at a time [Kla+21], as a developer is supposed to work on one view at a time. Thus, if the transformations outside the V-SUM ensure that only one of the views is changed at a time, the V-SUM provides the discussed benefits of the Commonalities approach.

This approach does, of course, not solve possible issues regarding synchronization and orchestration in the transformation network defined outside the V-SUM, but it only moves the problem of avoiding these issues away from the Commonalities specification by making according assumptions in terms of allowing only modifications of one view of a V-SUM. It does, however, clarify responsibilities, as there are precisely defined views across which other metamodels can be combined with those for which consistency is defined by means of Commonalities rather than defining consistency to the metamodels within the Commonalities specification directly and thus breaking the necessary assumption for the intended benefits of the approach. In the example, we have a clear separation into views for the structure of the object-oriented representation in Java, the UML, and potentially more metamodels and views for its behavior. It is up to the developer of the transformation network outside the V-SUM to ensure that no problems like execution loops occur by assigning clear non-conflicting responsibilities to the two transformations for structure and behavior of the V-SUM to the PCM.

Instead of only the PCM, there could be a more complex transformation network or another Commonalities specification, which may again be encapsulated in a V-SUM and provide its own views, across which both V-SUMs can be combined. Figure 11.15 depicts such an example, in which PCM and UML component models are related by a concept metamodel for component-based design, encapsulated in a second V-SUM. This V-SUM provides separate view types for the object-oriented structure, which is represented by both the PCM and the UML and is thus reflected in the concept metamodels, and for the behavior only represented in the PCM. These view types can be combined by means of ordinary transformations with those of the V-SUM for object-oriented design. Again, this approach does not prevent the occurrence of correctness issues due to the transformations outside the V-SUM as discussed in Part II, but at least it guarantees correctness within each V-SUM.

This approach can even be hierarchically composed, such that several kinds of specifications, including Commonalities encapsulated in V-SUMs, are again

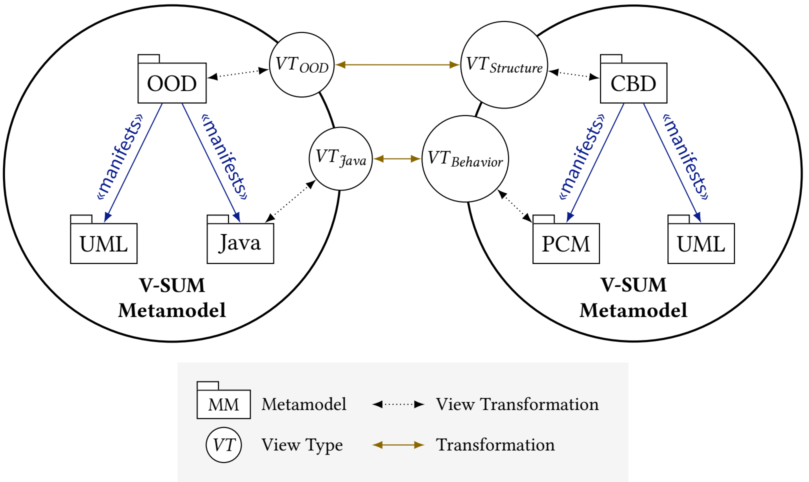


Figure 11.15.: Example for the combination of two Commonalities specifications for object-oriented (OOD) and component-based design (CBD) by encapsulation into V-SUM metamodels.

encapsulated into another V-SUM. For example, the V-SUMs in Figure 11.15 could be encapsulated in a V-SUM for object-oriented and component-based design to be reused together. If the transformation network between the inner V-SUMs is correct, which can also be achieved by defining Commonalities between the views of these V-SUMs again, the composed V-SUM again guarantees correctness and can provide well-defined views for different concerns of component-based and object-oriented design.

The sketched approaches for combining Commonalities specifications with other kinds of consistency specifications have to be considered as conceptual ideas which promise to provide the benefits of specifying modular, reusable specifications that ease the achievement of correctness. They have, however, not been applied yet and need to be practically evaluated in case studies.

11.5. Summary

In this section, we have discussed how the insights regarding effects of different network topologies on the quality properties of a transformation network can be used to mitigate trade-offs between them. We have motivated a different way of considering consistency in terms of making common concepts explicit as *Commonalities* instead of implicitly encoding them into consistency relations. We have used this way of specifying consistency to propose a construction approach for transformation networks that results in a tree topology providing inherent benefits regarding correctness but also providing high reusability due to the actual metamodels, whose instances are used to describe a system, being leaves of the tree induced by the transformation network. We conclude this chapter with the following central insight.

Insight III.2 (Trade-Off Mitigation)

Quality properties of transformation networks are influenced by the network's topology. Especially correctness and reusability are contrary properties, which induce a trade-off depending on whether the network topology is rather a dense or a sparse graph. The drawback regarding reusability in networks with tree topology arises from the fact that the metamodels represented by the inner nodes of the tree cannot be easily omitted, as consistency between several other metamodels is expressed across them. This can be mitigated by ensuring that the metamodels represented by the inner nodes are auxiliary artifacts and not the actual metamodels used by developers. This matches with a different way of thinking about consistency in terms of making the commonalities between metamodels to keep consistent explicit in additional metamodels rather than encoding them implicitly in consistency relations. Following such a specification approach leads to a network that improves both correctness and reusability, which are contradictory if only considering transformations between the metamodels whose instances are actually used by developers. Such an approach can even be used to define consistency partially for some of the metamodels and then combine it with other consistency specifications, such as ordinary transformations. To still have the same guarantees regarding correctness and reusability, such a specification can be encapsulated behind views, which provide projections of the information within the actual models and only allow one of them to be updated at a time.

12. Designing a Language for Expressing Commonalities

In the previous chapter, we have introduced the Commonalities approach, which defines a methodology for constructing transformation networks by means of auxiliary, so called concept metamodels. These concept metamodels contain the commonalities of the metamodels whose instances are to be kept consistent, denoted as concrete metamodels, as explicit entities rather than encoding them implicitly in transformations between the metamodels to be kept consistent. We have argued why this construction approach fosters achieving a specific tree topology of the transformation network. Such a topology improves correctness and reusability of the resulting transformation networks, which are contradictory properties when constructing networks only of transformations between the concrete metamodels, at least if a specific tree topology of the network is achieved.

Although the construction methodology of the Commonalities approach itself provides significant benefits and is thus a distinct and independently usable contribution on its own, the construction can be further supported with an appropriate language. While the approach requires the specification of concept metamodels as well as transformations realizing the manifestation relations between the metamodels, a language can integrate the specification of manifestations with those of the Commonalities. This improves conciseness and locality of the related information to be defined. While these improvements only foster usability but do not provide conceptual benefits, a language can also ensure the achievement of an appropriate tree topology. This can either be achieved by construction through restricting expressiveness or by defining analyzable constructs.

In this chapter, we discuss the design of such a language. We focus on design options and give an overview of the process and artifacts involved in such a language. We also depict a concrete language, for which we have developed the prototypical *Commonalities language*, with a focus on the relevant elements, their relations, and their operationalization. Although we also provide a prototypical realization of such a language, this chapter does not focus on the specifics of that language but rather the concepts behind it. It constitutes our contribution **C 2.3**, which consists of two subordinate contributions: a discussion of design options and the resulting process and artifacts for such a language; and a depiction of the structure of a concrete realization of such a language with a description of its semantics, its operationalization into transformations, and a summary of benefits that we expect from such a language. It answers the following research question:

RQ 2.3: How can a specialized language support the specification of a network topology that improves quality properties?

The insights in this chapter first give guidelines for developers of tools for constructing transformation networks. It especially clarifies the available design space for tools supporting the Commonalities approach. In addition, the chapter makes concrete proposals for how to develop such a language, which elements it has to contain, and how it can be operationalized. Finally, it even provides an actual realization of such a language, which can be readily used with the VITRUVIUS framework (see Subsection 2.3.2).

An overview of the prototypical realization of the Commonalities language and relevant design options along with a proof-of-concept has already been published [KG19]. An initial prototype of the language was developed in the Bachelor's thesis of Gleitze [Gle17] and extended for a case study evaluation in the Master's thesis of Hennig [Hen20], which have both been supervised by the author of this thesis. Since we focus on the concepts and design options for such a language in this thesis, we refer to those theses for details about the realization and capabilities of the Commonalities language.

12.1. Design Options

The development of a language for realizing the Commonalities approach offers several degrees of freedom. They range from conceptual degrees of freedom, e.g., regarding the operationalization alternatives discussed in Subsection 11.2.4, over notation types, such as textual or graphical representations, to the specific syntax to use or even reuse from existing languages.

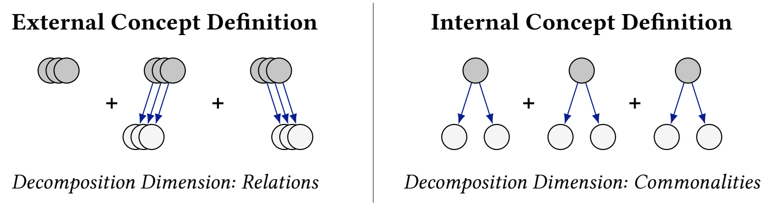


Figure 12.1.: Exemplification of alternatives to specify Commonalities by means of a separate, external specification of complete concept metamodels and manifestation relations or an integrated, internal definition of Commonalities with their manifestation relations. Circles denote Commonalities and manifestations, arrows denote manifestation relations.

We, in particular, consider the conceptual degrees of freedom and give an overview of how an according textual syntax can look like.

The conceptual degrees of freedom include options for operationalizing a specification in terms of using the concept metamodels as additional metamodels with the manifestation relations constituting ordinary transformations or in terms of generating direct transformations between the concrete metamodels from the Commonalities specification, as both discussed in Subsection 11.2.4. This option selection is transparent to the developer of a transformation network, as it only affects its operationalization.

In addition, we can distinguish *internal* and *external* specifications, depending on whether the specification is decomposed by the Commonalities or by the defined manifestation relations. This decision affects the developer of a transformation network, as he or she is directly concerned with the way in which Commonalities are specified. We discuss these two options in the following in more detail. Furthermore, we derive an overview of the resulting process for specifying and executing artifacts in such a language.

12.1.1. Internal and External Specification

We can distinguished two ways in which concept metamodels and manifestation relations can be specified according to the Commonalities approach. They depend on the dimension along which the specification is decomposed. More precisely, the specification can either be decomposed along the Commonalities, such that each Commonality together with all its manifestations is defined at one place, or it can be decomposed along the manifestation relations, such that all manifestation relations between a concept metamodel and its manifestation are defined at one place. We refer to these specifications as *internal* and *external* specifications, which we have already proposed in previous work [KG19] and which we illustrate in Figure 12.1.

External Concept Definition: Concept metamodels are defined as ordinary metamodels and each manifestation relation is defined as an individual transformation, i.e., manifestation relations are defined externally to concept metamodels and their Commonalities.

Internal Concept Definition: Each Commonality of each concept metamodel is defined together with its relations to manifestations, thus manifestation relations are defined internally with the Commonalities they belong to.

Without developing an additional language, the Commonalities approach can be realized by developing concept metamodels as if they are ordinary metamodels with appropriate modeling tools. The manifestation relations can then be defined with any existing transformation language that is able to generate incremental transformations. This conforms to an *external* specification, in which concept metamodels and manifestation relations are defined separately. It decomposes the specification along the relations, such that there are as many separate artifacts as there are concept metamodels and relations to be defined. For example, for Java and the UML an object-oriented design concept metamodel as well as two manifestation relations to each of the concrete metamodels would be defined separately.

Developing a specific language allows to integrate the definition of Commonalities with their manifestation relations. The relations to manifestations of a Commonality are then defined at one place with the declaration of the Commonality, improving locality of this related information. This conforms to an *internal* specification. It decomposes the specification along the Commonalities, thus as many separate specifications exist as Commonalities are defined. For example, for Java and the UML a class Commonality together with its manifestation as classes in both Java and the UML with the according relations of attribute values and references would be defined at one place.

Selecting one of these types of specification suffers from the “tyranny of the dominant decomposition” [Tar+99]. Thus, decomposition is only possible along one dimension of concerns, i.e., either the structural specification of Commonalities or the relational specification of manifestation relations, such that either one suffers from lacking separation of concerns in the other dimension. Thus, while one approach improves locality when adding Commonalities, the other improves locality when adding manifestation relations.

External specifications benefit from the separation of each manifestation relation into its own specification. This reduces dependencies between the manifestations and especially allows each developer who is responsible for a specific concrete metamodel to define the relation to each related concept metamodel as a whole instead of distributing this specification among all Commonalities specifications describing a concept represented in the concrete metamodel. In consequence, adding a new concrete metamodel only requires the addition and potentially adaptation of manifestation relations to concept metamodels. External specifications support this scenario well because of high locality of all information regarding a manifestation relation and because manifestation relations represent the largest part of the addition. Additionally, they can be realized without developing a new language.

Internal specifications require a dedicated language enabling the integrated specification of Commonalities and their manifestations. This improves locality regarding the information about each Commonality, as each Commonality is represented along with all its manifestations. In consequence, when initially developing Commonalities for a set of concrete metamodels, it is easier to add each single Commonality, because all information about the Commonality and its relations to the manifestations can be defined at one place. This can make it easier to understand the overall relation of that common concept among all concrete metamodels. In addition, it makes it less likely for a developer to miss the definition of one or more manifestations of a Commonality, as they are obviously missing in the specification of the Commonality, whereas in an external specification it is missing somewhere in the complete manifestation relation between the concept metamodel and its manifestation. Finally, the approach promises to be more concise, because the manifestation relations are defined within the Commonality they belong to instead of referencing the Commonality within a transformation again.

To benefit from locality regarding each Commonality and a more concise specification, we have decided to design a language that supports internal specifications. Depending on the usage context and usual change scenarios, an external specification may, however, be more appropriate. Then, modeling concrete metamodels with an existing modeling framework and the manifestation relations with existing transformation languages is sufficient.

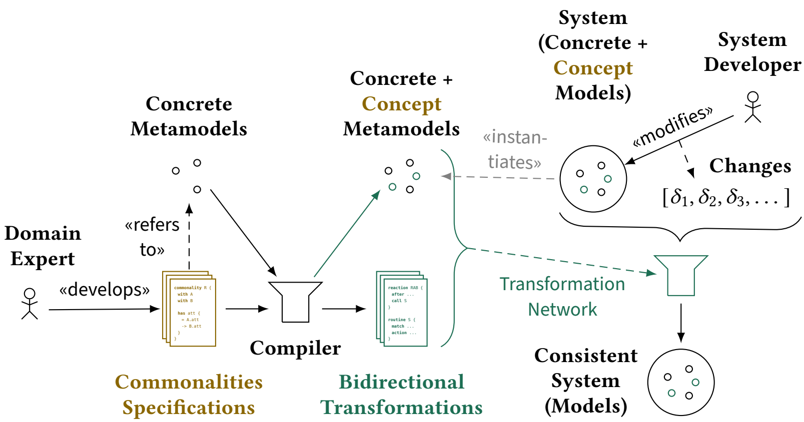


Figure 12.2.: The process for developing, compiling, and executing specifications in a language for Commonalities. From concrete metamodels and Commonalities specifications, additional concept metamodels and transformations are generated, which are executed at runtime for preserving consistency of models. Commonalities specifications by domain experts are marked orange, the generated artifacts (concept metamodels and transformations forming a network) are marked green. Concrete systems and changes depict runtime artifacts.

12.1.2. Artifacts and Process

Regarding the design options in Subsection 11.2.4 and Subsection 12.1.1, we have made the following, already argued decisions. First, we chose to operationalize a specification by treating concept metamodels as ordinary metamodels, such that instances of them are created and kept consistent. This option does especially not restrict expressiveness of the relations, and the generation of additional models can be hidden from the user by appropriate tooling. Second, we chose to provide a language that supports an internal specification of concepts to improve locality of the information regarding each Commonality. We expect this specification to be more concise and to better support the initial specification process for Commonalities.

The process of specifying, compiling, and executing artifacts in such a language is depicted in Figure 12.2. It is a specialization of the general process already depicted in Figure 1.2. A domain expert or transformation developer defines Commonalities specifications using the language, which refers to concrete metamodels that are to be kept consistent by the transformations derived from that specification. The compiler of the language takes the concrete metamodels together with the specifications to generate a set of concept metamodels in addition to the existing concrete metamodels, as well as a set of bidirectional transformations, which implement consistency preservation for the manifestation relations between the concept metamodels and concrete metamodels. These artifacts together form a transformation network as introduced in Definition 4.15.

A system developer specifies a system by models that instantiate the concrete metamodels of the Commonalities specification. The complete system description consists of instances of these concrete metamodels but also, in the best case hidden from developer, of instances of the concept metamodels for means of consistency preservation. Whenever the system developer produces changes to the instances of the concrete metamodels, the transformation network can be applied to the changes together with the models. It then returns a new set of instances of the concrete metamodel and concept metamodels that are consistent again, according to the proposed correctness notion of transformation networks in Definition 4.16.

12.2. The Commonalities Language

In this section, we present an overview of the Commonalities language. It constitutes one possible realization of a language for the Commonalities approach with the conceptual design choices that we have discussed in the previous section. This especially includes an internal specification of concepts. To give an impression of the language, we first introduce two examples for specifications in a prototypical realization of the language with a textual syntax, which we have already proposed in previous work [KG19] and which was originally developed in the Bachelor's thesis of Gleitze [Gle17] and extended in the Master's thesis of Hennig [Hen20]. We then give an overview of the language elements and introduce their general semantics before explaining the different categories of them at the given examples. Since we focus on the language concepts, we refer for details on its realization with a textual syntax to the theses of Gleitze [Gle17] and Hennig [Hen20].

```
concept ObjectOrientedDesign

commonality Class {
   with UML:(Class, single Model) {
      Class in Model.packagedElements
   }
   with Java:(Class, CompilationUnit) {
      Class in CompilationUnit.classifiers
   }

   has name {
      = UML:Class.name
      = Java:Class.name
      -> suffix(Java:CompilationUnit.name,
            Java:CompilationUnit.namespace + ".")
   }

   has methods referencing ObjectOrientedDesign:ClassMethod {
      = UML:Class.ownedOperations
      = Java:Class.members
   }
}
```

Listing 12.1: Exemplary specification for an extract of the `Class` Commonality between the UML and Java in the Commonalities language. Adapted from case study implementation at GitHub [GitApp].

12.2.1. Examples in Textual Syntax

We depict two examples for specifications in our prototype of the Commonalities language with a textual syntax in Listing 12.1 and Listing 12.2. The specifications depict extracts of a Commonality for classes in the UML and Java, as well as extracts of a Commonality for components in the PCM, the UML and as classes with their containing packages in the object-oriented design concept metamodel. The extracts are selected to reflect the different elements of the Commonalities language without introducing unnecessary complexity. We sketch the meaning of the examples in the following and clarify them along with the subsequent introduction of the language elements more precisely.

```
concept ComponentBasedDesign

commonality Component {
   with PCM:BasicComponent
   with UML:Component
   with ObjectOrientedDesign:(Class, Package) {
      Class in Package.classes
      <- Class.name hasSuffix "Impl"
   }

   has name {
      = PCM:BasicComponent.name
      = UML:Component.name
      = prefix(ObjectOrientedDesign:Class.name, "Impl")
      -> firstUpper(ObjectOrientedDesign:Package.name)
   }
}
```

Listing 12.2: Exemplary specification for an extract of the `Component` Commonality between the PCM, the UML, and the object-oriented design concept metamodel in the Commonalities language. Adapted from case study implementation at GitHub [GitApp].

The class Commonality, depicted in Listing 12.1, is restricted to their names and methods. In the UML, a class is represented by a class that is contained in a unique instance of a UML model. In Java, a class is also represented by a class that is contained in a compilation unit, which depicts one file consisting of imports and class specifications as a single unit of compilation [Hei+09b]. Names are represented equally in UML and Java classes. The name of the compilation unit is defined by the fully qualified name of the class, i.e., the concatenation of its namespace and the class name separated by a dot. The specification expresses this as the class name to be the suffix of the compilation unit name after the namespace followed by a dot. Methods are specified in a dedicated Commonality in the object-oriented design concept metamodel, such that they are only referenced in the class Commonality but without any specification of the relations of their contents.

The component Commonality, depicted in Listing 12.2, is restricted to their names. In the PCM and the UML, components are realized by explicit component or basic component metaclasses, respectively, which share the same name. In object-oriented design, components are defined to be represented by classes contained in a package. Classes are only considered to represent components when their name has an “Impl” suffix and their name is then defined to be the component name with an “Impl” suffix. The specification defines this as a prefix, analogous to the suffix for the name of a compilation unit, as it denotes that the component name is the prefix of the class name before “Impl”. Finally, the package name is defined to be the component name but starting with a lowercase letter whereas the component name is defined to start with an uppercase letter. Analogous to the prefix definition for the class name, the specification defines a `firstUpper` operation as the component name shall be the package name with the first letter in uppercase.

12.2.2. Elements Overview

The Commonalities language essentially consists of three categories of elements. First, at a top level, the structure of Commonalities needs to be defined in terms of specifying for each of them the concept metamodels they belong to as well as the features in terms of attributes and references it describes. Second, each Commonality needs to define its manifestations, i.e., the metaclasses of concrete metamodels or other concept metamodels being its manifestations, along with conditions defining when instances of metaclasses are to be considered a manifestation. This defines when a manifestation relation between a Commonality and metaclasses of another concept metamodel or concrete metamodel exist. Third, each Commonality needs to define the relations of its features to those of its manifestations. This defines the manifestation relations, i.e., the conditions that have to hold for considering a manifestation consistent to a Commonality.

Figure 12.3 depicts the essential elements of the Commonalities language. At the top, it depicts *metamodels*, *metaclasses*, *references*, and *attributes* as already existing in the notion of a general modeling formalism and as specified in concrete metamodels. The language introduces *concepts*, which represent the concept metamodels, and *Commonalities*, of which such a concept consists. In our realization, they can be considered specializations of metamodels and metaclasses but with the special semantics of being only auxiliary artifacts for the Commonalities approach. A Commonality consists of *Commonality references* and *attributes*, which, again, can be considered specializations of ordinary references and attributes. In the given examples, we have attributes for names and a reference to methods. Additionally, a Commonality contains

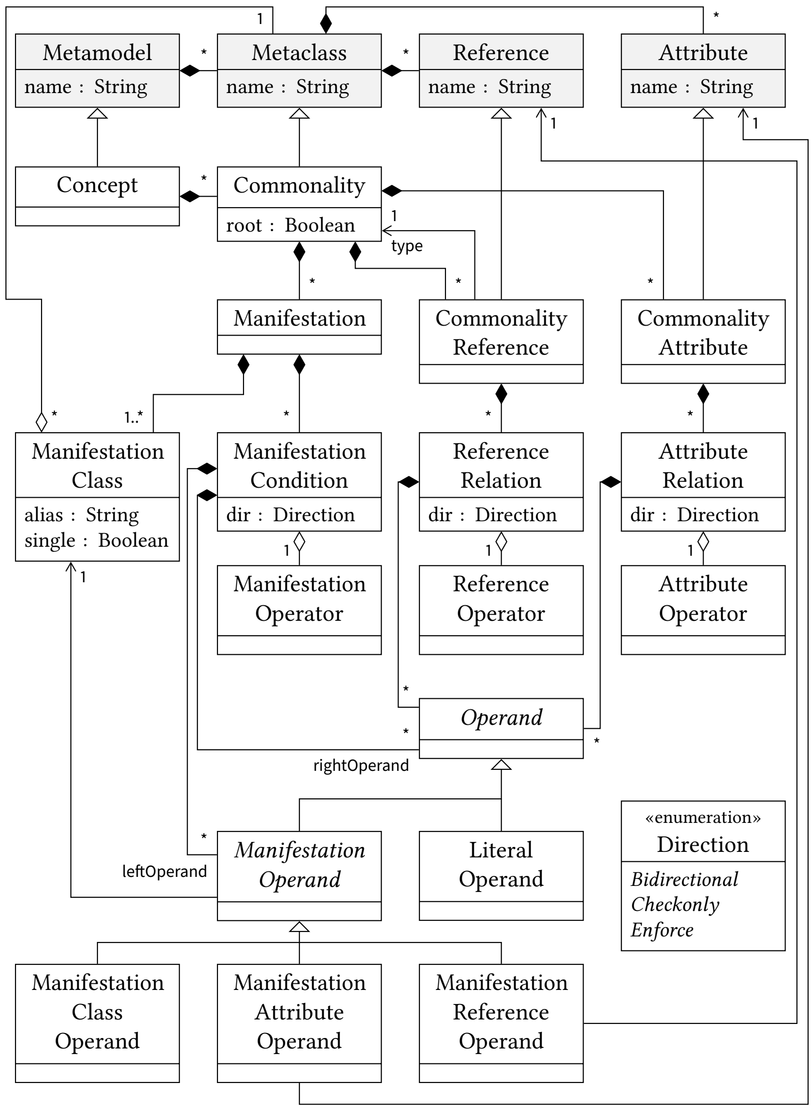


Figure 12.3.: Class diagram with the essential elements of the Commonalities language and their relations. Elements that exist independently from the language are depicted in the top row.

manifestations. Each manifestation represents the realization of the concept represented by the Commonality in another metamodel by one or more metaclasses and potentially further conditions for them. Such manifestation are, for example, a class and a compilation unit in Java for the class Commonality depicted in Listing 12.1. In preparatory work [Gle17; Hen20] as well as in the current state of prototypical implementation of the language [GitVit], such manifestations have also been called *participations*. Each Commonality reference and attribute is complemented by *reference* and *attribute relations* that define how these features are related to information in the manifestations.

In consequence, the manifestation conditions together with the attribute and reference relations define the consistency relations between the Commonality and its manifestations, which we have introduced as manifestation relations. All these relations consist of *operators*, which define how elements are related, and *operands*, which define the involved elements to be considered by the operator. The operators can be considered specifications of transformation rules, which take operands providing the information necessary to check or preserve consistency. In our language realization, operators can be specified by implementing specific interfaces of an API and thus dynamically extending the language with arbitrary operators. In consequence, these operators can be treated as reusable libraries containing operators at different levels of abstraction. They can, however, also be defined as a static part of the language and thus without the possibility to extend them. Operators have a *direction*, as they may enforce the defined relation either in both directions or only in one of them. For example, the name of a class in Listing 12.1 is related to the Java class name bidirectionally (denoted by "="). In consequence, a change of the Java class name leads to the change of the name of the class Commonality, which then changes the UML class name. But also a change of the class Commonality name, e.g., because of a change of the UML class name, leads to a change of the Java class name. The name of a compilation unit is only enforced, because it is derived from the Java class name, such that a change is propagated because of the changed Java class name anyway.

For reasons of simplicity, we omitted several elements of the language realization, which concerns generalizations as well as specializations of the depicted elements. For example, manifestation conditions, reference, and attribute relations represent relations between the Commonality and its manifestations, especially comprising a direction, which can be represented in a `Relation` supertype. Likewise, the three operator types for manifestations, references, and attributes can be derived from a common `Operator` supertype.

12.2.3. Language and Elements Semantics

A Commonality defines the elements that different manifestations have in common and how they are related. For example, the class Commonality given in Listing 12.1 denotes that classes have names and methods in common and that they are related by specific naming schemes of name attributes and specific references containing representations of methods. Thus, whenever there are elements in one model that match one of the specifications for a manifestation in terms of instantiating the defined classes and fulfilling the defined conditions, there must be elements in other models matching the other manifestation specifications and fulling the defined relations for attributes and references. In theory, from such a specification consistency relations, according to Definition 4.18, could be derived, which enumerate the tuples of instances of the metaclasses in the manifestations that fulfill the manifestation conditions as well as the attribute and reference relations.

We especially want to preserve consistency rather than only checking it and thus derive consistency preservation rules from such a specification. In Subsection 12.2.6, we discuss such an operationalization in more detail. In general, we distinguish the instantiation, update, and deletion of a Commonality, according to the scenarios already depicted for Mappings in the bidirectional Mappings language [Kla+21, Sec. 7.2.1]. The Commonalities language specifies a bidirectional transformation between a Commonality and each of its manifestations. Thus the behavior of each such transformation conforms to the behavior of the Mappings language, in which we could define the transformation between a Commonality and its manifestation.

Instantiation: A Commonality is *instantiated* whenever elements are added to a model such that they instantiate the metaclasses of a manifestation of that Commonality and fulfill the defined manifestation conditions. We say that these elements *match* the manifestation of the Commonality. In that case, an instance of the metaclass realizing the Commonality is created, and its attributes and references are initialized with values according to the relations defined in the Commonality. Then, for each other manifestation, instances of the metaclasses are generated and inserted into a model according to the specified manifestation conditions and defined relations of attributes and references. For example, according to Listing 12.2, whenever a Java class with the suffix “Impl” is created and inserted into a package, a component Commonality with the name of the class without that suffix is created, and a basic component in the PCM and a component in the UML model with that name are created.

Deletion: A Commonality is *deleted* whenever the elements for which a Commonality was instantiated do not match the manifestation anymore. Then, the instance of the metaclass realizing the Commonality is removed as well as the instantiations of all metaclasses of the other manifestations. For example, whenever a Java class representing a component is removed, or even if only the “Impl” suffix is removed, the Commonality and all other manifestations in the PCM and UML models are removed.

Update: A Commonality is *updated* whenever any of the attribute or reference values of the elements of a manifestation for which a Commonality was instantiated get changed. In that case, the values in the Commonality and all other manifestations are updated if the changed value is used in the according attribute or reference relations, i.e., if it is one of its operands. The relation also defines a direction to indicate whether the change is only checked in the manifestation, i.e., whether a change of any value in the manifestation leads to an update of the value in the Commonality and the other manifestations, whether the change is only enforced, i.e., whether a change of a value of the Commonality leads to a change of the values in the manifestation, or whether it is bidirectional, i.e., both checked and enforced. This ensures that consistency is preserved for the elements for which a Commonality is instantiated.

While for the instantiation and deletion of a Commonality only the manifestation classes and their conditions are relevant, for an update only the attribute and reference relations are relevant. To relate this to the Mappings language, manifestations and their conditions conform to *single-sided conditions*, whereas attribute and reference relations conform to *bidirectionalizable conditions* of Mappings [Kla+21, Sec. 7.2.1]. Since a Commonalities specification can be seen as a combination of defining multiple Mappings in the Mappings language, large parts of the semantics and possibilities for the realization are comparable. We thus focus on the structure that Commonalities define on top of bidirectional mappings and explicitly refer to work on Mappings for concepts that have already been researched and are reusable, such as operators and methods to define and execute them bidirectionally.

In Subsection 6.1.2, we have also discussed the addition, removal, and change of condition elements as the relevant change types to be distinguished when realizing consistency preservation. This conforms to the scenarios of instantiation, deletion, and update of a Commonality. The addition of a condition element of a consistency relations defined by a Commonality specification means that the according manifestation is matched and thus the Commonality is instantiated. The removal and update conform to the deletion and update of a Commonality analogously.

12.2.4. Commonalities and Manifestations

The top-level elements of the Commonalities language are Commonalities. Each of them depicts a common concept, such as a class or a component, and is associated with a concept metamodel, which groups common concepts that belong together. In the given examples, each specification contains one Commonality and starts with a specification of the concept metamodel it belongs to, comparable to a package specification of a class in Java. These concept metamodels are named *ObjectOrientedDesign* and *ComponentBasedDesign*, according to the ones we have proposed in the examples for composing Commonalities in Subsection 11.2.2.

The specification of each Commonality starts with its manifestations, which are metaclass tuples of the concrete metamodels or concept metamodels in which the Commonality manifests, together with further conditions on when instances of these metaclasses form a manifestation of a common concept. Such a manifestation denotes which elements have to exist in a model and which conditions they have to fulfill to consider these elements a manifestation of a common concept described by the Commonality. The metaclass tuples are represented by *manifestation classes*, which only reference an ordinary metaclass but may also have an alias for referencing it. The metaclasses they reference can be ordinary classes of a concrete metamodels, such as UML components in Listing 12.2, or they may be Commonalities of a concept metamodel, such as classes of the object-oriented design concept metamodel referenced in the component Commonality in Listing 12.2.

Additionally, manifestation classes can be declared *single* to denote that they only occur uniquely within one metamodel and do not share a Commonality with others, comparable to a singleton, but are still relevant for the Commonalities specification. For example, a UML model always has a root container of the metaclass *Model*, which does not share a Commonality with Java in the object-oriented design concept metamodel and exists uniquely, as there may only be one such UML model. An alternative representation of such unique elements are Commonalities with only one manifestation that are bootstrapped. This means that such a Commonality and its manifestation would always exist and thus be created at the start of system development rather than instantiating it when a manifestation is matched. For example, a UML model would be created as soon as a new software development project is started. Kramer uses such a bootstrap representation of elements in his Mappings language for bidirectional transformations [Kra17, Sec. 7.1].

Manifestations further define manifestation conditions, which specify when instances of the metaclasses referenced by the manifestation classes shall be considered a manifestation of the defined Commonality. Obviously, not every instance tuple shall be considered as such. This can further depend on properties of the single objects or on the relation between them. For example, for the manifestation of components in object-oriented design according to Listing 12.2, only classes matching a specific naming scheme shall be considered components, and only a pair of class and package in which the class is contained in that package shall be considered a component. Any other pair, in which the class is not even contained in the package at all, should not be considered a manifestation of a component. Such conditions can be seen as restrictions at the instance or model level, whereas the metaclass tuples define a restriction at the type or metamodel level.

A manifestation condition consists of a *manifestation operator*, a *left operand*, and a list of *right operands*. The left operand can be considered the reference element of the operator. It can be any metaclass of the manifestation or any of its attributes or references, for which a condition shall be defined. The operator can be any Boolean-valued condition that is evaluated for the left operand and potentially further right operands, which can, again, be metaclasses of the manifestation or any of their features, or a literal, such as a fixed number or string. Listing 12.2 contains the operator *in*, which validates whether the value of the left operand is contained within a reference given as the right operand. In addition, the operator *hasSuffix* checks whether the value of the left operand contains the right operand as a suffix.

12.2.5. Features and Relations

In addition to manifestations, a Commonality defines features, i.e., attributes and references, which represent the information shared by several manifestations, as well as their relations to information defined in the manifestations. Attributes only need to be identifiable by a name, whereas references, in addition, need to define the type they reference. This type has to be a Commonality again, such as the reference for methods of a class referencing the Commonality `ObjectOrientedDesign:ClassMethod` in Listing 12.1.

While these attributes and references only define the structure of the Commonality, the relations defined within them express how attributes and references are represented in the manifestations. Reference and attribute relations consist of an operator and operands. The operator defines how the Commonality attribute or reference is related to features of the manifestations or other literal values, which are passed to the operator as operands. For example, the name attribute of the component Commonality in Listing 12.2 is related to the name of a class in object-oriented design by a *prefix* operator, which takes the class name and an “Impl” string as operands. This operator expresses that the name of the component Commonality is the prefix of the given class name removing “Impl”.

In comparison to manifestation conditions, attribute and reference relations only have one set of operands, because the element for which the relation is defined is implicitly given by the Commonality attribute or reference, whereas a manifestation condition must explicitly define which metaclass or feature it belongs to. Analogous to manifestation conditions, they define a direction. For example, in Listing 12.2, the relation between the name of the component Commonality and the name of the class in object-oriented design is defined to be bidirectional (denoted with a “=”), which means that changes of both elements are propagated to the other. The component name is also related to the name of the package in which the class in object-oriented design is contained. This relation is, however, defined as an *enforce* relation, such that the package name is enforced whenever the name of the component changes, but a modification of the package name does not lead to a change of the component name.

Whenever a relation is defined as bidirectional, the operator needs to define how changes are propagated in both directions, i.e., how to update the Commonality attribute or reference among changes in any of the operands and how to update the operands whenever the Commonality attribute or reference is changed. Our prototypical implementation allows to define such operators in Java code. They need to be derived from a common interface to dynamically extend the language. Each operator needs to implement

```java
public class PrefixOperator
      extends AbstractAttributeOperator<String, String> {
   private final String suffix;

   public PrefixOperator(final String suffix) {
      this.suffix = suffix;
   }

   @Override
   public String applyTowardsCommonality(final String full) {
      String prefix = full;
      if (full.endsWith(suffix)) {
         prefix = full.substring(0, full.length() - suffix.length());
      }
      return prefix;
   }

   @Override
   public String applyTowardsManifestation(final String prefix) {
      return prefix + suffix;
   }
}
```

Listing 12.3: An implementation of the prefix operator for Commonalities as used in the prototypical implementation of the Commonalities language. The operator is derived from an abstract implementation for operators relating attributes to attributes. The generic type parameters denote the attribute types in the Commonality as well as in the manifestation. Adapted from the VITRUVIUS code repository [GitApp].

methods for being applied towards the Commonality as well as towards the manifestation. Listing 12.3 depicts the implementation of the mentioned prefix operator. It is initialized with the suffix to remove, such as the “Impl” suffix to remove from a class name to get the component name in our example. The operator application towards the manifestation simply concatenates the given prefix and suffix, such that in the example “Impl” is appended to the component name. Towards the Commonality, the operator checks whether the given name ends with the specified suffix and then returns the according prefix. The operator is implemented to return the given name whenever it does not have the defined suffix. This is sufficient in the example, because in that case the Commonality is deleted anyway because of the manifestation condition. In general, it may also be useful to define different behavior, such as throwing an error, asking the user for some decision about the name, or even mechanisms to reject the change.

Since both application directions of the operator need to be implemented individually, a developer can implement contradicting behavior in both directions. This can result in an incorrect transformation, because the consistency relation implied by a Commonality with an attribute or reference relation with such a faulty operator may be empty, as the relations encoded into the different operator directions can never be fulfilled at the same time. To avoid this, it can be beneficial to derive the implementation of both directions from one specification of the relation, like in declarative transformation languages such as QVT-R or the Mappings language. Especially for the latter one, Kramer has already proposed a methodology for defining unidirectional conditions and deriving the other direction whenever possible [Kra17, Sec. 7.4]. In addition, he has proposed a set of useful operators for defining consistency relations between elements [Kra17, Sec. 7.3].

Finally, operators should only employ information provided by their operands. They should especially not use further features of given elements or even traverse the model to retrieve further elements. If this is the case, the graph induced by the relations between features of Commonalities and their manifestations defined through the operands represents the graph of consistency relations, which we have employed in Chapter 5 to define and analyze compatibility of consistency relations. Thus, if this induced graph forms a tree, according to Definition 5.6, the consistency relations are inherently compatible according to Theorem 5.6, as we have aimed to achieve with the construction approach of Commonalities, as proposed in Subsection 11.2.3.

12.2.6. Operationalization to Transformations

In Subsection 12.1.2, we have depicted that a Commonalities specification must be compiled to concept metamodels and transformations between them and concrete metamodels to be used as an ordinary transformation network. Since the semantics of relations defined between a Commonality and its manifestations is analogous to the semantics of bidirectional relations defined in the Mappings language [Kra17, Chap. 7], we refer to that detailed discussion for operationalizing Commonalities specifications to transformations. We still discuss essential responsibilities of the compiler process.

The operationalization of Commonalities specifications requires the generation of transformations and, in particular, their consistency preservation rules according to Definition 4.5. Thus, we need to derive rules that instantiate, delete, or update Commonalities after changes to a manifestation such that they are again consistent to the consistency relations implied by the manifestation relations defined in the Commonalities specification and vice versa. The Reactions language (see Subsection 2.4.3) allows the definition of Reactions and routines that restore consistency after changes. Each Reaction defines the type of change it reacts to and executes routines, which identify whether the consistency relation to which they preserve consistency is violated by that change and then execute actions to restore it. Since that kind of specification fits to our formalization of consistency preservation rules in Chapter 4 and thus fits to the goals of the operationalization of Commonalities specifications, we describe the operationalization to Reactions and have also implemented it in our prototype. An analogous operationalization has been developed for the Mappings language by Kramer [Kra17, Sec. 7.7], which also compiles to Reactions.

The operationalization of Commonalities to Reactions requires that a Reaction is created for each change that may require the instantiation, deletion, or update of a Commonality. Thus, for each manifestation class and each of its features referenced in a Commonality as well as for the metaclass realizing a Commonality and each of its features, a Reaction for their change is created.

The creation of an instance of each of the metaclasses in a manifestation as well as each modification of a feature that is used within the manifestation conditions can lead to a set of model elements that match the manifestation. Thus, for each of these changes a Reaction needs to be derived that checks whether such a manifestation is actually instantiated and then instantiates a Commonality accordingly. In addition, for the creation of a Commonality, a Reaction that creates all its manifestations has to be created. Kramer proposes an analogous algorithm for the Mappings language [Kra17, Alg. 1].

Likewise, a deletion of an instance of any of the metaclasses in a manifestation as well as any modification of a feature that is used within the manifestation conditions can lead to the situation that elements that previously matched a manifestation do not match it anymore. Thus, for each of these changes a Reaction needs to be derived that deletes the Commonality, and for the deletion of a Commonality a Reaction that removes all its manifestations has to be created. For the Mappings language, this has been defined in an analogous algorithm [Kra17, Alg. 2].

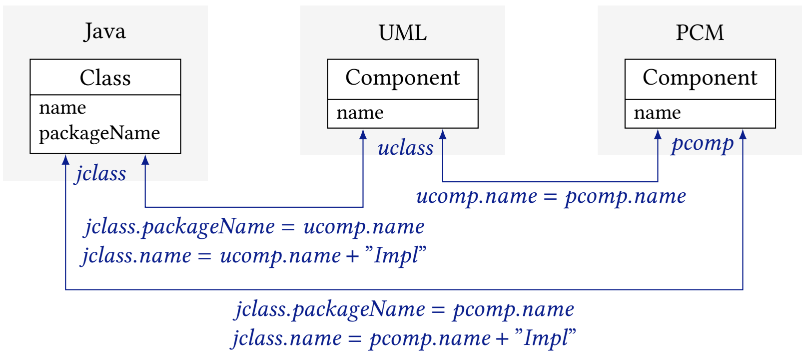
Finally, all changes to features used within the attribute and reference relations of a Commonality can require updates of the Commonality attributes and references, and, in consequence, of the features of the other manifestations. Thus, for each attribute and reference of both the Commonality and its manifestations, Reactions have to be created that update the related elements accordingly. The definition how to update the related elements is given by the implementation of the operators, such as the *prefix* operator depicted in Listing 12.3. An algorithm for updating features that are put into relation has also been proposed for the Mappings language [Kra17, Alg. 3].

A benefit of compiling to Reactions is that they have well-defined semantics [Kra17, Sec. 6.7] and that they are proven complete and correct [Kra17, Sec. 9.2.4 and 9.3]. This means that they are able to preserve consistency according to any possible consistency relation and that their execution actually preserves consistency to the consistency relations that are implied by the specified consistency preservation rules. Thus, the transformation language with which the manifestation relations of Commonalities are operationalized does especially not restrict expressiveness in any way.

12.2.7. Expected Benefits

The Commonalities approach proposed in Chapter 11 can provide several benefits compared to an ordinary network of transformations, especially in terms of mitigating the trade-off between correctness and reusability of the transformations. While this is a conceptual benefit that is given by construction of the approach and not only a claim that has to be validated, the expected benefits of a dedicated Commonalities language especially concern usability and applicability of the approach, which can be argued but also have to be empirically evaluated to provide further evidence.

In Figure 12.4, we depict simplified consistency relations between components in the PCM and the UML as well as classes in Java, together with a specification of these relations in QVT-R and the Commonalities language. In contrast to our previous examples in Listing 12.1 and Listing 12.2, the Commonalities specification does not define a hierarchy of Commonalities with two concept metamodels for component-based and object-oriented design



Commonalities Specification

```
concept ComponentBasedDesign

commonality Component {
  with uml:Component
  with pcm:Component
  with java:Class

  has name {
    = uml:Component.name
    = pcm:Component.name
    = prefix(java:Class.name,
        "Impl")
    -> firstUpper(
        java:Class.packageName)
  }
}
```

QVT-R Specification

```
relation UMLComponent2PCMComponent {
  componentName:String;
  domain uml ucomp:Component {
    name = componentName;
  }
  domain pcm pcomp:Component {
    name = componentName;
  }
}
```

```
relation Class2PCMComponent {
  componentName:String;
  domain java jclass:Class {
    name = componentName + 'Impl';
    packageName = componentName;
  }
  domain pcm pcomp:Component {
    name = componentName;
  }
}
```

```
relation Class2UMLComponent {
  componentName:String;
  domain java jclass:Class {
    name = componentName + 'Impl';
    packageName = componentName;
  }
  domain uml ucomp:Component {
    name = componentName;
  }
}
```

Figure 12.4.: Example for consistency relations between classes and components expressed with QVT-R and the Commonalities language.

but defines the Java manifestation within the component Commonality. The example gives an impression of the expected conciseness of Commonalities specifications in comparison to ordinary, bidirectional specifications, due to which we expect benefits in comprehensibility and specification effort.

As a first benefit, we expect the Commonalities language to improve comprehensibility. The language decomposes the specification of consistency along the Commonalities rather than along the transformations as with ordinary transformation languages. In consequence, the information how a single common concept is represented in different metamodels is necessarily spread across several transformations if each transformation only relates two metamodels. With Commonalities, this information is located at a single place, which is the specification of the according Commonality. We expect this to improve the overall comprehensibility of how different elements in different metamodels sharing a common concept are related. While a Commonalities specification improves compatibility anyway due to its likeliness of leading to a consistency relation tree, it can also make it easier for developers to get a global understanding of consistency, which would be necessary to avoid incompatibilities. This is due to the reason that incompatibilities occur when different transformations relate the same elements in different ways, which becomes less likely if these different transformations are defined at one place within the Commonality, such that developers responsible for other metamodels and thus further manifestations of that Commonality can easily understand the notion of consistency the other developers have. Figure 12.4 demonstrates how information about a component Commonality is represented at one place with the Commonalities language, whereas it is spread across three QVT-R transformations relating all pairs of metamodels. As discussed in Subsection 11.3.2, the number of transformations increases even quadratically with the number of manifestations to keep consistent. Finally, this is only a benefit of the Commonalities language, which realizes an internal specification of concepts (see Subsection 12.1.1), because only such a realization decomposes the specification along the Commonalities.

In Subsection 11.3.2, we have discussed the reduced specification effort of the Commonalities approach in general, when considering the scenario that a further metamodel shall be kept consistent. Especially if the information this metamodel shares with other concrete metamodels of an existing Commonalities specification is already represented by Commonalities, only the manifestation relations of the elements of the metamodel to be added to the existing Commonalities have to be defined. In an ordinary transformation network, all pairwise relations between the metamodel to be added and the existing metamodels, with whom it shares common concepts, have to be defined, potentially leading to duplications and thus higher effort.

We have, however, also discussed that the effort for keeping instances of two metamodels consistent or, analogously, the initial effort for defining Commonalities for multiple metamodels by specifying the Commonalities for the first two of them can be high and, in particular, higher than defining ordinary transformations. Two metamodels can be kept consistent by a single transformation, whereas a Commonalities specification requires an additional concept metamodel and two transformations, one between each concrete metamodel and that concept metamodel, to keep them consistent. The Commonalities language reduces the effort for specifying these three artifacts by the choice of an internal specification of concepts. A single transformation rule for a consistency relation of a common concept is expressed by a Commonality, its manifestations, and the specification of relevant features and their relation to the manifestations. But instead of three places to define this information at, it is defined at one place of the Commonalities specification. Although Figure 12.4 only represents a single, simple example, it indicates that a Commonalities specification, even concerning three rather than two manifestations to keep consistent, is not less concise than the expression of the according consistency relations in QVT-R. This comparison implicitly assumes an intuitive comparison of conciseness in terms of lines of code. It is, of course, an open question whether the specification effort actually correlates with such a metric and whether conciseness according to that metric is even given in further cases than the single one depicted here. Nevertheless, we have argued indicators for expecting the benefit of reducing specification effort by the proposed language, but we emphasize that its validation requires empirical studies in terms of controlled experiments with developers applying both approaches and measuring their effort in terms of the required time to achieve an error-free solution. We provide preliminary results of a case-study-based evaluation in Chapter 13.

12.3. Summary

In this chapter, we have introduced the Commonalities language. It supports the Commonalities approach for constructing transformation networks as proposed in Chapter 11 with a dedicated language. We have made the design choice of decomposing the specification in that language along Commonalities rather than transformations, which promises to improve comprehensibility of the specification and its conciseness, such that specification effort is even reduced when only few metamodels shall be related. While we have discussed the relevant elements of that language in more detail and explained them at examples with a concrete textual syntax that we have developed for our prototypical implementation in the Vitruvius framework [GitVit], we refer to the Mappings language of Kramer [Kra17] for further details on its operationalization. The proposed Commonalities language can be seen as an extension of that Mappings language for the purpose of relating metamodels to the concepts they share with each other rather than relating metamodels with each other. We close this chapter with the following central insight.

Insight III.3 (Language)

In addition to the design options given by the Commonalities approach as a whole, a language supporting it additionally needs to define how to decompose the specification. This can be done along the Commonalities, such that each Commonality is specified at one place with its manifestations, or along the transformations, such that each concept metamodel and each relation between a concept metamodel and one of its manifestations is defined at one place. While depending on the usage context either of them can be beneficial, a decomposition along the Commonalities can only be realized with a dedicated language that derives the concept metamodels and transformations between them from a specification in that language. This approach can especially improve conciseness and comprehensibility. Such a language consists of three categories of elements, one for the structure of concept metamodels, one for the manifestations, and one for the relations between both. The operators that define how information is propagated along the relations to keep models consistent across their Commonalities should make their operands, i.e., the features of Commonalities and manifestations, explicit and not internally acquire further information from the models. Then, the graph induced by these operands can be used to identify whether the specified consistency relations fulfill the definition of a consistency relation tree, which is likely to be achieved with a Commonalities specification and inherently guarantees compatibility.

13. Evaluation and Discussion

In the preceding chapters 10–12, we have discussed quality properties of transformation networks and how they can be improved systematically. We have discussed the effects of the network topology on properties, and we have derived the Commonalities approach for constructing transformation networks, which uses the effects of topologies to optimize specific quality properties and to mitigate trade-off decisions between them. Finally, we have proposed the Commonalities language, which supports the process of applying the Commonalities approach to define a transformation network.

The central benefit of the developed Commonalities approach and the supporting Commonalities language is given by construction. The way in which the transformation network is defined inherently improves correctness, especially in terms of compatibility, and reusability. These are contradicting quality properties in a network of transformations that are directly defined between the metamodels whose instances shall be kept consistent. We have argued this trade-off mitigation in Subsection 11.3.1. In addition to this central benefit, we have discussed further benefits that we expect from both the Commonalities approach as well as the Commonalities language in Section 11.3 and Subsection 12.2.7. We empirically evaluate these benefits with a case study presented in this chapter.

In the discussions of Chapter 11 and Chapter 12, two general issues affecting the Commonalities approach remained that may only be solved by empirical investigations. First, although consistency relations and their preservation are only described in a different way by means of auxiliary models, it may be possible that the approach restricts the possible consistency relations that can be described in any way, especially under the goal of achieving a consistency relation tree (see Subsection 11.2.3). Second, achieving a consistency relation tree with the approach is important to maximize the guarantee of compatibility while ensuring maximal reusability (see Subsection 11.2.3), but it is unclear how far or under which conditions such a tree can be achieved in practice.

In addition to the benefits of the Commonalities approach, the Commonalities language is expected to reduce the specification effort. The Commonalities approach itself can improve the specification effort in comparison to an ordinary transformation network when the auxiliary metamodels and transformations to them are defined with existing modeling tools (see Subsection 12.2.7). The Commonalities language is, however, supposed to reduce this additional effort. We have thus developed a prototype of that language, and we evaluate its correctness as well as the goal of reducing the specification effort in a case study that we present in this chapter.

13.1. Goals and Methodology

In this evaluation, we aim to validate relevant properties of the Commonalities approach and the Commonalities language that are not given by their construction but have to be analyzed empirically. This especially concerns the applicability of the approach and specific benefits provided by the language but also the general completeness of the approach, i.e., the ability to express every desired set of consistency relations. It is an extension of the preliminary case study that focused on validating feasibility that we have conducted and presented in previous work [KG19].

In the following, we present an empirical evaluation based on a case study, in which we apply a prototypical realization of the Commonalities language to consistency relations and their preservation in the domain of component-based software engineering, which we have introduced in Section 2.5 and already employed for the evaluation of our contributions regarding the construction of correct transformation networks in Section 9.2. We summarize the general goals of the evaluation along with according questions and metrics as quantitative measures for answering them in Table 13.1.

Regarding the Commonalities approach as such, we are interested in the possibility to be used by transformation developers to define consistency preservation. In a first place, this comprises the validation of completeness according to Section 10.1. We want to find out whether it is possible to define arbitrary consistency relations with the Commonalities approach. In fact, Stevens shows that every multiary relation can be expressed by an auxiliary metamodel with binary relations between this auxiliary metamodel and the metamodels to describe consistency between [Ste20b]. This means that also

Goal 6: (Approach)	Show that transformation developers can use the Commonalities approach to specify consistency and its preservation between multiple models.
Question 6.1: (Completeness)	How far are the Commonalities approach and the Commonalities language capable of defining arbitrary consistency relations?
Metric 6.1.1:	*Definition ratio: Ratio of consistency relations for which consistency can successfully be defined*
Question 6.2: (Practicality)	How far can a Commonalities specification achieve a consistency relation tree in practice?
Metric 6.2.1:	*Cross-tree ratio: Number of cross-tree relations compared to the number of relations*
Goal 7: (Language)	Show that transformation developers can define consistency in a concise way with the Commonalities language.
Question 7.1: (Correctness)	Do transformations generated by specifications in the Commonalities language preserve consistency according to the defined relations?
Metric 7.1.1:	*Preservation ratio: Ratio of scenarios in which consistency can successfully be preserved*
Question 7.2: (Benefit)	How much more concise is a specification in the Commonalities language compared to a specification in the Reactions language?
Metric 7.2.1:	*Code ratio: Ratio between the SLoC in a Commonalities specification compared to the SLoC in a Reactions specification*

Table 13.1.: Goals, questions, and metrics for Commonalities approach and language evaluation.

any set of binary relations, which induce a multiary relation as discussed in Subsection 4.1.2, can be expressed by an auxiliary metamodel and binary relations between it and the metamodels to define consistency between. This conforms to the general idea of the Commonalities approach and, if recursively applied, even to the hierarchic composition of Commonalities. Despite this theoretical insight, we investigate whether such a specification is actually achievable in practice, especially under the specific goal of achieving a consistency relation tree in a specification of Commonalities. Even if the Commonalities approach itself may not be restricted in expressiveness, the proposed Commonalities language may be because of the selected way in which Commonalities and their relations are defined. This leads to Question 6.1, which we aim to answer by measuring how many consistency relations of our case study we are able to define:

$$\textit{definition ratio} = \frac{\textit{\# of defined consistency relations}}{\textit{\# of total consistency relations}}$$

The more consistency relations we are able to define, the higher it is an indicator for the completeness of the approach and the language. It does, however, especially indicate completeness of the Commonalities language, such that we derive by argumentation whether restrictions in expressiveness exist only because of the language or already because of restrictions of the Commonalities approach. The language especially serves as a means to draw conclusions about completeness of the approach.

For the Commonalities approach to provide the benefit of inherently guaranteeing compatibility, it must be possible to define a consistency relation tree by means of the additional concept metamodels and their Commonalities. In this first place, we aim to identify whether such a tree can be defined at all. We do not aim to systematically find conditions under which this is possible or even how the Commonalities approach and the Commonalities language can systematically support this. Knowing whether the specification of such a tree is achievable at all is a prerequisite for these further investigations, which we refer to as future work. It identifies practicality of the approach, as considered in Question 6.2. To this end, we measure in our case study how many of the defined relations are cross-tree relations, i.e., violate the definition of a consistency relation tree:

$$\textit{cross-tree ratio} = \frac{\textit{\# of cross-tree consistency relations}}{\textit{\# of defined consistency relations}}$$

In the best case, this ratio is 0, such that the relations actually form a consistency relation tree. Referring to Definition 5.6 for consistency relation trees, we consider the graph induced by the relations defined by the manifestation relations of a Commonalities specification between metaclasses of the concrete metamodels and concept metamodels, in which they are called Commonalities. We only consider the actually defined consistency relations, as we cannot make statements about the relations that we do not express by Commonalities in the case study.

Regarding the Commonalities language, we are most interested in finding indicators for improving usability of the Commonalities approach by providing a concise way of specification. First of all, this requires that the language operates correctly, i.e., that it actually generates transformations that preserve consistency according to the defined consistency relations, as defined in Question 7.1. This actually evaluates two correctness notions. First, it identifies whether the language implementation is correct at a technical level. Second, it identifies whether the concepts for operationalizing Commonalities into transformations defined with the Reactions language, as proposed in Subsection 12.2.6, are correct. We measure this by executing change scenarios and identifying whether the results are consistent to the specified relations:

$$\textit{preservation ratio} = \frac{\textit{\# of successful scenarios}}{\textit{\# of total scenarios}}$$

In the best case, this metric evaluates to 1, such that in all scenarios consistency can successfully be preserved. In failure cases, we manually investigate the cause, especially distinguishing between conceptual issues in the operationalization of the Commonalities language and technical faults in the compiler implementation.

As an essential benefit of the Commonalities language, we have motivated the reduction of specification effort (see Subsection 12.2.7). This is of particular importance, because developing a Commonalities specification for consistency between two metamodels by means of existing tools for metamodel and transformation definition requires the definition of three artifacts compared to a single artifact when defining an ordinary transformation. The Commonalities language aims to resolve this issue. We consider the specification effort by means of conciseness, i.e., the size of a specification with Commonalities in comparison to a specification of ordinary transformations between the metamodels, as defined in Question 7.2. Since the Commonalities language compiles to Reactions and a comparable implementation of the case study already exists for them (see Subsection 9.2.3), we compare the size of a Commonalities specification with the size of a specification in the

Reactions language in terms of the Source Lines of Code (SLoC) and measure the following metric:

$$\textit{code ratio} = \frac{\textit{\# of SLoC with Commonalities}}{\textit{\# of SLoC with Reactions}}$$

The lower the value of this metric, the more concise a specification in the Commonalities language can be considered in comparison to a specification in the Reactions language. We expect this insight in conciseness to correlate with the required specification effort.

13.2. Prototypical Implementation

For conducting the case study, we have used a prototypical implementation of the Commonalities language and the realization of the case study with this language in the Vitruvius framework (see Subsection 2.3.2). We have also employed this framework for the implementation of the our case study for evaluating concerns and approaches regarding correctness in Section 9.2. In addition, the Reactions language (see Subsection 2.4.3), to which the Commonalities language compiles, is part of the Vitruvius framework.

The implementation of the Commonalities language conforms to the considerations discussed in Chapter 12. It implements an internal specification of concepts, i.e., it allows the specification of each Commonality in one file together with all its manifestations and relations to them, according to the elements we have introduced in Figure 12.3. The syntax conforms to the examples we have given in Listing 12.1 and Listing 12.2 but provides even more sophisticated specializations of the depicted language constructs. We have also defined a set of general as well as case-study-specific operators for manifestation conditions as well as attribute and reference relations.

The specifications in the Commonalities language are compiled to Ecore metamodels for the concept metamodels and specifications in the Reactions language for the manifestation relations, according to Subsection 12.2.6. Reactions, in turn, are compiled to ordinary Java code that implements a specific API of the Vitruvius framework. The framework orchestrates the transformations with a simple strategy that enqueues all transformations defined for the model that is modified by the current transformations and

Element Type	Total	Covered			
		Direct	Implicit	Overall	
Metaclass	13	7	3	10	77 %
Attribute	27	19	1	20	74 %
Containment reference	13	9	0	9	69 %
Non-containment reference	4	2	2	4	100 %
Enumeration	2	0	2	2	100 %
Total	**59**	**37**	**8**	**45**	**76 %**

Table 13.2.: Numbers of UML metamodel elements used in the case study. Adapted from [Hen20, Tab. 10.4].

executes them until no further changes are made. Since we aim to define Commonalities that represent a consistency relation tree, transformations should be inherently compatible and are thus likely to terminate with such an orchestration strategy (see Paragraph 9.2.5.2). The implementation of the framework with the Commonalities and Reactions language is available in a GitHub repository [GitVit].

13.3. Case Study

We have performed a case study based on the metamodels PCM, UML and Java, as introduced in Section 2.5. The specification of Commonalities is based on two sets of consistency relations, one for the PCM and object-oriented design, applying to both Java and the UML, and for the UML and Java, which we have both also introduced in Section 2.5. We have used the same consistency relations to implement a case study of transformations with the Reactions language in Chapter 9 for evaluating our contributions regarding correctness of transformation networks. Since the Commonalities language compiles to Reactions, this allows us to compare the two realizations.

The two sets of consistency relations are motivated by the two concepts of object-oriented design and component-based design, between which have already distinguished in the explanation of the Commonalities approach in Chapter 11 and for which we have especially considered a hierarchic representation in Subsection 11.2.2. We have thus implemented the case

Element Type	Total	Covered			
		Direct	Implicit	Overall	
Metaclass	30	9	11	20	67 %
Attribute	13	11	0	11	85 %
Containment reference	32	19	1	20	63 %
Non-containment reference	1	0	0	0	0 %
Enumeration	0	0	0	0	100 %
Total	**76**	**39**	**12**	**51**	**67 %**

Table 13.3.: Numbers of Java metamodel elements used in the case study. Adapted from [Hen20, Tab. 10.5].

Element Type	Total	Covered			
		Direct	Implicit	Overall	
Metaclass	16	7	2	9	56 %
Attribute	15	7	1	8	53 %
Containment reference	18	6	0	6	33 %
Non-containment reference	8	3	1	4	50 %
Enumeration	1	0	1	1	100 %
Total	**58**	**23**	**5**	**28**	**48 %**

Table 13.4.: Numbers of PCM metamodel elements used in the case study. Adapted from [Hen20, Tab. 10.3].

study with two according concept metamodels, of which the one for object-oriented design defines Commonalities between the UML and Java, and the one for component-based design defines Commonalities between the object-oriented design concept metamodel and the PCM. The case study has been implemented with the Commonalities language in the Master's thesis of Hennig [Hen20]. Details on the implemented consistency relations and Commonalities can also be found there [Hen20, Sec. 3, A.2]. In the following, we summarize the case study.

We have realized a subset of the consistency relations that we have introduced in Section 2.5 and that we have realized with the Reactions language in the case study presented in Chapter 9. Table 13.2, Table 13.3, and Table 13.4 give an impression of the size of the implemented case study. They depict the numbers of elements by type for the three metamodels that are relevant for the originally defined consistency relations, denoted as *total*, and those that were realized in the case study, denoted as *covered*. We distinguish between elements that are *directly* and *implicitly* covered, according to whether they were actually defined as manifestation classes or features of them and passed to the operators ensuring consistency explicitly, or whether they were only accessed within the operators. The total case study size is reflected by the absolute numbers of considered elements, and the coverage of the originally presented consistency relations is reflected by the relative numbers.

The implicitly covered elements concern, for example, primitive data types or enumeration literals, which have to be instantiated on demand but which are not explicitly represented within the Commonalities. Implicit elements also cover structures of elements that are only represented by one element in the other metamodels. For example, the UML represents the realization of an interface by a class through an indirect reference of a dedicated generalization element, i.e., the class references a generalization, which, in turn, references the implemented interface, whereas the Commonality and the Java representation have a direct reference to the implemented interface. In that case, the generalization element is not explicitly referenced in the Commonality specification but only implicitly used within the operators for the implementation relation. In Java, many metaclasses are only implicitly covered, because primitive types, type references, and modifiers are represented as metaclasses, whereas they are represented as instances in the other metamodels (see [Kla16, Sec. 5.7.4]) and are thus only used implicitly within operators for attributes that represent references or modifiers.

The implementation contains 15 Commonalities, of which eight belong to the object-oriented design and seven to the component-based design concept metamodel. These Commonalities put 124 elements of the concrete metamodels into relation, which represent around 64 % of the total 193 elements that are relevant for the complete set of introduced consistency relations. While the case study implementation covers most elements of the UML (76 %) and Java (67 %), it only covers 48 % of the PCM elements. Most of the missing consistency relations are due to intended restrictions of the case study size or restrictions in expressiveness of the Commonalities language. We further discuss the reasons in the subsequent results presentation.

The implementations of all Commonalities are available in a corresponding GitHub repository of the Vitruvius project [GitApp]. It also contains

Consistency Relation	Test Cases	Successful Test Cases	
Package	6	6	100 %
Class	26	26	100 %
Class method	40	40	100 %
Constructor	24	22	92 %
Field + association	20	20	100 %
Interface	10	10	100 %
Interface method	28	28	100 %
Total	**154**	**152**	**99 %**

Table 13.5.: Test case results for consistency relations in object-oriented design. Adapted from [Hen20, Tab. 10.2].

Consistency Relation	Test Cases	Successful Test Cases	
Repository	6	6	100 %
Interface	6	6	100 %
Signature + parameters	48	48	100 %
Composite data type	48	48	100 %
Repository component	6	6	100 %
Provided role	12	8	67 %
Total	**126**	**122**	**97 %**

Table 13.6.: Test case results for consistency relations in component-based design. Adapted from [Hen20, Tab. 10.1].

test cases, which we have reused from those that we have defined for the case study with the Reactions language, as presented in Subsection 9.2.3. Since we only want to evaluate whether results are correct regarding the consistency relations for which we have defined consistency preservation with Commonalities, we have reduced the tests to those for the according consistency relations in comparison to the tests summarized in Table 9.7. We have, however, also added further test cases such that in total more test cases for the case study implementation with the Commonalities language exist than for the implementation with the Reactions language. All these test cases perform changes that lead to the violation of a specific type of consistency relation and require the transformations to change the other models for restoring consistency, which are then validated by the test case.

Table 13.5 and Table 13.6 summarize the test cases together with their results when applied to the case study implementation with the Commonalities language, which we discuss in the subsequent section. The test cases are split into one set only concerning consistency relations for object-oriented design, i.e., those keeping only UML and Java models consistent, and another set for component-based design, in which also PCM models are kept consistent. For every change scenario, such as the addition or modification of a specific type of element involved in a consistency relation, we consider one test case per change direction per model pair. For object-oriented design, this results in two test cases for each scenario, since each change can be performed in the UML and checked in Java and vice versa. In component-based design, each change can be performed in any of the three models and propagated to any of the two other models, resulting in three test cases for each scenario. In consequence, test case numbers are a multiple of two for object-oriented design and a multiple of six in component-based design.

In total, we have executed 285 test cases. They include 154 test cases for keeping UML and Java models consistent with the object-oriented design Commonalities and 126 test cases for keeping PCM, UML, and Java models consistent with Commonalities for object-oriented design and component-based design. While these test cases use minimalist models that are sufficient for representing the consistency relation under test, we have also used the Media Store system model [SK16], which is a comprehensive case study system for the PCM and which we have already used in the evaluation of our approaches for constructing correct transformation networks in Chapter 9. For this PCM model, we simulate its construction by producing a change sequence that yields the models, which conforms to the *Reconstructive Integration Strategy* proposed by Langhammer [Kla+21; Lan17]. We have defined five additional test cases using this construction simulation, which validate that a UML model is created and that it is consistent to the defined consistency relations, including components, interfaces, operation signatures, data types, and provided roles.

13.4. Results and Interpretation

We use the implemented case study and the conducted tests to answer the evaluation questions summarized in Table 13.1 or at least to find indicators for how their general answers are expected to be based on the data from the case study. The questions are split into those especially concerning the Commonalities approach and those concerning the Commonalities language.

Commonalities Approach

We have explained that we did not implement all consistency relations with the Commonalities language that we have realized with the Reactions language in the evaluation for transformation network correctness in Section 9.2 but only a sufficiently complex subset. We selected consistency relations forming a coherent set that can be realized with reasonable effort and such that we do not expect further insights regarding applicability, practicality, and usability of the Commonalities approach and language from the implementation of the omitted relations. To avoid a bias by defining an arbitrary subset of the consistency relations, which, by accident, can completely or almost completely be realized with the Commonalities language, we consider the ratio of consistency relations realized with the Commonalities language in comparison to the complete consistency relations depicted in Section 2.5. It results in the following metric values, derived from the values in Table 13.2, Table 13.3, and Table 13.4:

$$\begin{aligned}
&\textit{definition ratio}_{sums} &&= 64\,\% \quad \left(= \tfrac{45+51+28}{59+76+58}\right)\\
&\textit{definition ratio}_{average} &&= 64\,\% \quad \left(= \tfrac{76\,\%+67\,\%+48\,\%}{3}\right)\\
&\textit{definition ratio}_{UML-Java} &&= 71\,\%
\end{aligned}$$

We counted the elements of the metamodels affected by the consistency relations. To avoid a bias by having different numbers of elements in the different metamodels, we have calculated the ratio both based on the sums of the elements across all metamodels ($\textit{definition ratio}_{sums}$), as well as the equally weighted average of the coverage of all metamodels ($\textit{definition ratio}_{average}$). They do, however, both sum up to the same value of 64 %. Since UML and Java represent those metamodels that are kept consistent by a single concept metamodel for object-oriented design and can thus be considered a minimal application of the Commonalities approach for only two metamodels, we explicitly calculated the ratio only for these two metamodels as well. Since both ways of calculation introduced above yield the same value, we have only depicted the single result of 71 %.

The coverage ratios especially give an impression of how comprehensive the realized consistency relations are. To evaluate completeness of the Commonalities approach and the language, it is of particular importance to identify how many of the consistency relations that we intended to implemented could not be realized. In summary, we found that most consistency relations that we aimed to realize could actually be achieved, except for multi-valued types in the PCM and Java, which is due to current limitations of the language. Multi-valued types are fields and parameters of a type with an upper bound in its multiplicity higher than one. This can be expressed with explicit multiplicities in the UML and with collection data types in the PCM, whereas they have to be rolled out as explicit implementations of collections in Java. The current implementation of the Commonalities language lacks an operator for that situation, which is, however, not a conceptual limitation but can be added with some additional effort. In addition, provided and required roles in the PCM as well as generalizations in the UML are currently not fully supported and in parts only covered implicitly, because the current implementation of the Commonalities language only supports explicit relations to containment references, but roles and generalizations contain ordinary references to the provided, required, or implemented interfaces, which can up to now only be accessed implicitly within operators of the Commonalities language. This is a technical limitation, which the current case study implementation avoids by implementing complex operators to support these situations, which is why the according test cases are actually successful, but the language lacks sufficient support for such relations.

The remaining consistency relations were omitted on purpose and are summarized in more detail in the Master's thesis of Hennig [Hen20, Sec. 3]. They comprise *composite components* in the PCM, which are comparable to basic components and only need to be distinguished by an according naming schema or the containment of assemblies of other components, of which at least the latter requires some implementation effort but is not expected to be a conceptual issue. In addition, *systems* and *subsystems* are not considered, because they are composite components with slightly different semantics.

In summary, the case study results indicate that in answer to Question 6.1 the Commonalities approach itself is complete, as we have already expected because of the theoretical considerations by Stevens [Ste20b]. The Commonalities language, however, currently has some limitations that prevent the realization of some consistency relations or at least made it more difficult than it should be. We found these to be only technical limitations that can be solved by extending the language, such that they do not hide actual limitations of the underlying Commonalities approach. The results emphasize the status of the Commonalities language implementation as a prototype but still indicate possible completeness of such a language according to the concepts for such a language proposed in Chapter 12.

The central question to evaluate for the Commonalities approach concerns its practicality in terms of achieving a consistency relation tree with a Commonalities specification to benefit from the discussed guarantees in quality improvement. We have discussed in Subsection 11.2.3 that the defined consistency relations have to form a consistency relation tree, and in Subsection 12.2.5 we found the graph induced by the operands of the operators putting Commonalities and their manifestations into relations to be the one to consider for identifying a consistency relation tree. Since in several Commonalities of our case study elements are accessed implicitly within the operators and not all of them are explicitly defined as operands, these elements have to be considered as well. For that reason, we conducted the investigation of the defined relations to identify the graph as a consistency relation tree manually. In this manual analysis, we found that none of the defined relations lead to the violation of the definition of a consistency relation tree according to Definition 5.6:

$$\textit{cross-tree ratio} = 0$$

Although restricted to a single case study, this at least serves as a first indicator for the practicality of the approach as asked in Question 6.2, i.e., that it actually supports or at least enables the specification of a consistency relation tree. To mitigate the risk of mistakes performed in the manual analysis of consistency relations, the test results also serve as a further indicator that the relations form a tree. Violations of such a tree structure can easily lead to incompatibilities, as discussed in Chapter 5, which can then lead to non-termination, as discussed in Chapter 8, especially with the simple orchestration strategy that we employed for the case study. We have, however, not observed any non-termination in the test cases. The failing tests were due to other reasons, which we discuss in the following. Although even without a tree structure the consistency relations can be compatible, or even if they are incompatible it may not lead to failures during transformation execution, it still serves as an indicator that the consistency relations form a tree. Even if this is not the case, the evaluation at least shows that the transformations behave correctly, thus no matter whether this is actually achieved by defining a consistency relation tree or any other reason that makes the operationalization of Commonalities specifications likely to be correct, it is only important that correctness is achieved.

Commonalities Language

As a prerequisite for any further insights on the Commonalities language, we first have to validate its correctness. This covers the correct implementation of the language and its compiler as well as correctness of concepts how to compile Commonalities into Reactions. While the former can be seen as simple bug testing, the latter gives us insights in whether the operationalization concept is correct, which especially means that the language can be seen as a derivation of the Mappings language, from which we have reused operationalization concepts (see Subsection 12.2.6). To validate correctness, we consider the test case results for those consistency relations of the case study that we have implemented with the Commonalities language. According to Table 13.5 and Table 13.6, more than 97 % of them are successful:

$$\textit{preservation ratio} \geq 97\,\%$$

In addition, the five test cases for the Media Store case study system also produce the expected results. Regarding Question 7.1, this is a high indicator for correctness of the operationalization concept of the Commonalities language as well as its implementation, especially because the failures of the remaining test cases are caused by the used Vitruvius framework and by incompleteness of the Commonalities language.

In total, six test cases fail. This concerns two tests cases for constructors in object-oriented design, which both implement the same scenario but once from Java to the UML and once vice versa. In this test scenario, multiple constructors with different parameter lists are created. The Vitruvius framework first executes transformations for the insertion of both constructors and afterwards for the addition of parameters. This leads to two indistinguishable constructors with empty parameter lists when first execution the transformation, such that when adding the parameters the two constructors cannot be distinguished anymore. Processing the constructor additions one after another in the framework would solve the problem. Anyway, the same problem would occur when using the Reactions language. Regarding provided roles

	Reactions *(omitted)*	Commonalities	Difference	
Specifications	2390 *(302)*	514	−1876	−78 %
Utilities	2250 *(445)*	2523	273	11 %
Total	**4640** ***(747)***	**3037**	**−1603**	**−53 %**

Table 13.7.: SLoCs in the Commonalities and Reactions specification for the consistency relations between the UML and Java. For Reactions, the numbers only cover consistency relations realized in the Commonalities specification, whereas those in parenthesis cover the relations not realized in the Commonalities specification. Adapted from [Hen20, Tab. 10.9].

in component-based design, four test cases fail, because the references to provided interfaces are only implicitly covered in operators. We have already discussed before that the Commonalities language currently only supports relations for containment references, such that other references have to be processed within operators. Provided roles are contained in components, which in turn reference the provided interface. When a provided role is added to a component, this is processed by a relation in the component Commonality. The operator for that relation also implicitly considers the reference to the interface within the role, but this reference may not yet be set. When the interface of the role is set or changed later, this change is not propagated, as no relation for it is defined in a Commonality and thus no Reaction is generated for it, such that the according test cases fail. In consequence, this is a result of technical incompleteness of the Commonalities language as discussed before, but it is not a matter of incorrectness of its operationalization.

The Commonalities language is supposed to support the construction of a transformation network according to the Commonalities approach. In comparison to applying the construction approach with ordinary modeling and transformation tools, it is supposed to reduce the specification effort, especially in the simple or initial case in which consistency between only two metamodels shall be specified. The case study implementation contains a specification for two metamodels in terms of the object-oriented design Commonalities between the UML and Java. We had already defined their consistency preservation with a direct transformation by means of the Reactions language in previous case studies for the Vitruvius framework [Kla+21], which we have also employed for the evaluation in Section 9.2. Table 13.7 compares the realization of consistency relations between the UML and Java by means of the Reactions language and the Commonalities language in terms of SLoCs. Since there is no unique measure for SLoCs for these languages, we have decided to format the code such that each statement for every grammar rule starts in a new line, according to the formatting used for Reactions [Kla+21]. Since not the complete specification is defined within the language artifacts itself but also within utilities written in Java or Java-like code, we also counted the SLoCs in that code. Since the Reactions language allows to define arbitrary code within the Reactions, only few utility code is necessary, whereas the Commonalities language requires utility code already for all use-case-specific operators. Considering all code together leads to a reduction in SLoCs between Reactions and Commonalities of more than 50 %:

$$\textit{code ratio} = 47\,\%$$

Drawing conclusions of this metric to the actual specification effort suffers from several biases. First, the counted SLoCs can only be considered an approximation, as, for example, the utilities are shared with other projects and thus they are not tailored to consistency between the UML and Java. Second, whether conciseness in terms of SLoCs actually leads to less specification effort is not evaluated but only assumed as a hypothesis. In response to Question 7.2, the case study provides an indicator for achieving conciseness in comparison to the Reactions language. Finally, it is only necessary to avoid an increase in specification effort, thus conciseness should at least not decrease. Whether or not the actual value of around 50 % in code size reduction is representative, it at least shows that we may not expect a drastic increase in code size, which would improve the specification effort.

13.5. Discussion and Validity

From the discussed case study and its results, we can derive several important insights. They are given even though a single case study only gives indicators for specific properties, as it suffers from potential limitations especially in external validity. We discuss threats to the validity of our results after a summary of important insights.

Insights

With the empirical evaluation, we especially aimed to validate two properties. First, we wanted to investigate practicality of the Commonalities approach in terms of being able to express arbitrary consistency relations and especially to achieve a tree structure that inherently guarantees certain quality properties. Second, we aimed to validate the reduction of specification effort with the Commonalities language to ensure that in the simple case of defining consistency between two metamodels, specification effort does not increase in comparison to a direct transformation between them.

In the case study, we found by manual analysis of the defined Commonalities that a tree structure inherently guaranteeing compatibility was achieved. There are several threats to the validity of generalizing this result to achievability of a tree structure in every case, which we discuss in the following subsection. Although a tree is what we want to achieve to have definite guarantees for compatibility, the actual goal is to achieve correctness, which can also be achieved without a specification inducing such a tree topology. Violations of a tree structure only introduce potential incompatibilities, which can potentially lead to execution cycles. This does, however, not need to be problematic, because a cycle in the relations does not have to lead to a cycle between corresponding elements in an instance and because even if there is such a cycle, it can be implemented properly so that execution terminates consistently, like we aimed to achieve for ordinary transformation networks in Part II. Thus, even if the results of our analysis of relations in the case study was erroneous, the execution of the transformations derived from the Commonalities specification still worked properly, i.e., it always terminated and led to consistent results in the executed test scenarios. Thus, independent from whether a consistency relations tree was actually achieved or not, the approach led to a correct specification, which also provides optimal reusability by construction of the Commonalities approach and thus mitigates the trade-off between these properties as intended.

Regarding the Commonalities language, we were able to show a reduction in code size of about 50 % for the consistency relations between the UML and Java in comparison to a specification with the Reactions language. Although this evaluation also suffers from several threats to validity, which we discuss in the following, it at least indicates that we do not have to expect a significant improvement in code size. For two metamodels, a specification according to the Commonalities approach by means of the Reactions language would require twice as many Reactions code plus the definition of the concept metamodel in comparison to a direct Reactions specification between the metamodels. Thus, a specification that requires at most two times the code lines required for a Reactions specification between the metamodels provides a benefit with respect to average realizations of the Commonalities approach. Thus, even if specification effort and code lines are not linearly correlated, the reduction of code lines by 50 % in comparison to an improvement by factor two will likely lead to less specification effort.

Threats to Validity

In the following, we discuss different potential threats to the validity of the discussed results. The restriction to one case study especially limits external validity, thus all results can only be seen as indicators for the statements that we make. We will, however, discuss for which reasons validity of the statements may actually be restricted, distinguished by construct, internal, conclusion, and external validity [Woh+12].

Construct Validity There are especially two threats regarding construct validity, which arise from the manual analysis of achieving a tree structure with Commonalities as well as from the selection of consistency relations to implement. The manual conduction of the analysis of the consistency relation graph induced by the Commonalities specification is prone to faults, as it lacks an explicit graph representation that can be analyzed automatically. Violations of the tree structure would, however, likely have led to failures during execution. The Reactions generated from Commonalities are not synchronizing, as discussed as a preliminary for transformations in networks (see Chapter 6), thus in case there are cycles of consistency relations and thus transformations across which changes are propagated, the execution would likely lead to failures as missing synchronization and also potential incompatibilities prevent the transformations from finding consistent results. In particular, in Section 9.2 we found that missing synchronization is the most severe issue that, in the case studies, led to a failure of every test case.

The selection of consistency relations that we have realized with the Commonalities language may not be representative. Other relations might have led to different results regarding all evaluation questions, including completeness of the approach and the language, the achievability of a tree structure, and correctness of the compiler. We have, however, argued why we performed that selection of consistency relations and why we do not expect other relations to yield other results. In addition, even if the actually realized relations may not be representative, at least the complete case study with all relations represents a sophisticated scenario. It especially requires the usage of all elements provided by the Commonalities language in contrast to preliminary studies in which only specific elements of the language were required and used to achieved feasibility results [KG19].

Internal Validity Internal validity may especially be affected regarding the results for properties of the Commonalities language. First, the language was only a proof-of-concept before implementing the case study and was improved along with the case study realization. Thus, there is the risk of optimizing the language for the case study. This especially affects the operators, as several of them are specific for the case study, whereas the overall structure of the language is generic. Even if this reduces validity, it does not affect the results for the evaluation of the Commonalities approach and the conciseness of the language but only its completeness and correctness.

In addition, the case study was implemented by a single person, such that the results may be affected by the performance of this person. This may affect completeness and conciseness of the approach. Regarding completeness, another person may have been able to implement more relations, thus evaluation results may only become better when performed with a different person. Conciseness of an implementation can vary in both directions when performed by different persons, which induces a bias in the results regarding conciseness. It can be the case that the average developer produces less concise results than in our evaluation, which can affect generalizability of the results. The goal of the language is, however, that specifications are not less concise than an implementation with ordinary transformations, which we have shown is at least possible and which is even given if the results are biased due to the measured amount of improvement in conciseness.

Another threat is given by the comparison of Commonalities with Reactions for evaluating conciseness of the language. The Reactions language allows imperative, unidirectional specifications of transformations and provides high expressiveness by being rather verbose and providing only few abstractions in comparison to a general-purpose programming language. A language at a higher abstraction language, such as the Mappings language, from which we have reused large parts of the compiler, or QVT-R, may provide a better baseline for comparison. We have used the Reactions language as a baseline because the case study was already implemented and, in particular, evaluated with that language. The case study implementation has already been compared with an implementation in ordinary Java code [Kla+21], which has shown a reduction in code size. This shows that specifications in the Reactions language are not arbitrarily verbose, and for other languages such an evaluation does even not exist. Since the goal of the evaluation was especially to show that Commonalities specifications do not considerably increase the specification effort and since the results indicate that we do not have to expect an increase in code size by several times, such an increase in specification effort cannot be expected.

Conclusion Validity The correlations between evaluated metrics and performed statements are straightforward for completeness, correctness, and the achievement of a tree topology. The assumed correlation between conciseness of the code and specification effort is, however, a threat to conclusion validity. Code that is more concise may even be harder to specify, because it can require more knowledge about language constructs and experience with using them. In particular, much of the logic of a Commonalities specification is defined in operators. We especially observed a significant improvement in conciseness in the specification code but not in the utilities code, to which the operators belong. Thus, if the operators are the part that is hard to specify, the effort may even increase. As discussed before, we do, however, not require a significant reduction in specification effort to gain a benefit from the Commonalities language, as the central benefit is already given by its guarantees regarding correctness and reusability. In consequence, the language is only supposed to mitigate the increase in effort induced by the Commonalities approach as such, which is at least twice the effort, measured in terms of SLoC, for a direct transformation between two metamodels, whereas in the case study the language reduced it by the same factor. So even if there is a large bias in the relation of conciseness and specification effort, the results still indicate that specification effort does not increase with the Commonalities language.

External Validity The central threat regarding external validity is the limitation to a single case study. This may affect generalizability if other case studies produce different results regarding completeness, correctness, and conciseness. We did, however, mitigate this threat by not using a toy example but a sophisticated case study, including multiple realistic consistency relations and a hierarchic definition of them with Commonalities. In addition, we do not expect practicality in terms of achieving a tree topology to depend on the actual case study but only on the kinds of relations, which we expect to be representative in the study as discussed for construct validity. Finally, the effort for setting up a case study and to set up the baseline for a comparison with ordinary transformation is rather high. At least, we were able to use an independently developed baseline, which we have used for our evaluations in Section 9.2, to evaluate conciseness of the Commonalities language.

13.6. Limitations and Future Work

The Commonalities approach as well as the Commonalities language have been developed particularly for the specification of descriptive specifications of consistency (see Subsection 11.1.3) and with specific goals of quality improvement that require a tree topology of the specified relations. These assumptions as well as the discussed evaluation results yield limitations of the proposed approach and language. We discuss them in the following and derive opportunities for future work.

Tree Achievement The essential benefits of the Commonalities approach regarding correctness guarantees arise from the likeliness of defining a tree of consistency relations. Although the evaluation indicates that such a tree is achievable, or even if it is not achieved, it may ease the achievement of a correct transformation network, it is finally still up to the developer to ensure correctness. The language supports him or her in achieving it, but it would be beneficial to finally make the language ensure correctness. We have sketched in Subsection 12.2.5 how the graph of consistency relations can be derived from the operands of relation operators in the Commonalities language. It requires that operators only use model elements that were explicitly passed to them as operands. The approach for proving compatibility, which we have presented in Chapter 5, can then be applied to these relations. Since the relations are not expressed as OCL constraints but as arbitrarily complex operators written in Java, redundancy of relations cannot be determined easily. Thus, the approach may only be used to validate if the relations represent a consistency relation tree, but this is sufficient as achieving a tree is the goal anyway. Performing such an integration in future work would further improve the benefits of the language by giving the developer explicit feedback whether he or she defined a topology that actually guarantees the benefits provided by the Commonalities approach.

Declarative Specification We have motivated the Commonalities approach as a reasonable way of thinking about and specifying common concepts of different metamodels. In Subsection 11.1.3, we have discussed that this especially fits for descriptive specifications of consistency, i.e., those where metamodels actually share common concepts, often in terms of redundancies. Other consistency specifications, which may prescriptively define more complex dependencies, may not fit into such a notion of common concepts, which can make it difficult to apply the Commonalities approach to them, or which may lead to specifications that do not inherently induce a tree topology. We have already considered in Section 11.4 how Commonalities specifications can be combined with other transformation networks, be they defined with Commonalities or ordinary transformations, which allows to combine different kinds of specifications for different purposes and to apply Commonalities only where they fit properly. In future work, it would thus be of particular interest to apply these ideas of combining specifications and evaluate the feasibility of these ideas. In addition, the further application of the approach to different case studies can reveal whether the restriction to declarative specifications is actually relevant or whether the approach can also be applied well to other specifications, despite our motivation.

Language Extensions The limitations we have found in the evaluation were mostly caused by limitations of the current implementation of the Commonalities language. Even in the Master's thesis, in which the case study was conducted, several language extensions had to be made to support the parts of the case study that we have presented before [Hen20, Sec. 9]. The current limitations concern the availability and complexity of operators, such as missing operators for relating attributes to references, missing complex pattern matching for indirect references that led to the test failures, and missing reusability, e.g., in terms of inheritance, which currently leads to repetitions when specifying similar concepts. While these limitations should be addressed in future work by conceptual and technical extensions of the Commonalities language, it also induces a research question regarding the operators. Currently, the operators are suited for the implemented case study, but it is an open question how a reusable set of operators at an appropriate level of abstraction can or should be defined such that relevant, recurring cases can be realized with a predefined operator set. This is currently left open in the language design, as we did not make any restrictions regarding the operators (see Figure 12.3) but should be considered as a general future research question.

Evidence Improvement A central drawback of the presented evaluation is the limited evidence due to the restriction to a single case study, which affects generalizability of the results. Although we have intensively argued why the results are still valuable indicators for the properties that we have evaluated, it still remains a threat in external validity. Thus, it is important to provide further evidence on the results by applying the approach and the language to further case studies. This can also be used to evaluate the assumptions we have made for the language, as we have discussed before. Finally, since the evaluation presented in Section 9.2 lacks similar drawbacks in external validity, case studies in future work can be combined for both, such that consistency relations are elicited and validated by test cases only once. This also allows to compare the results, for example, to further validate the specification effort of the Commonalities language.

13.7. Summary

In the preceding chapters, we have presented the Commonalities approach and the Commonalities language for mitigating trade-off decisions between quality properties of transformation networks induced by the topology of that network. We have discussed how this mitigation can be achieved by an appropriate construction approach for transformation networks and how it can be supported by a proper language under the assumption of achieving a specific kind of tree topology.

To evaluate whether this assumption is achievable and thus how complete and practicable the approach is, and to evaluate how far the language actually supports the specification, we have conducted an empirical evaluation at a case study. The evaluation indicates that the approach is actually applicable in scenarios in which metamodels share common concepts and that the language provides a concise way of specifying consistency. Since the approach is only supposed to be applied in specific situations, it is, however, necessary to combine such a specification with other ordinary specifications of transformation networks. In consequence, the Commonalities approach depicts a solution for specific consistency relations, for which it provides more guarantees regarding certain quality properties than ordinary transformation networks. In general, it must be combined with other transformations, such that correctness of the combination must again be ensured by means discussed in Part II.

Part IV.

Epilogue

14. Related Work

Collaboration is a key factor in software engineering processes and especially MDSD, but it also represents a key challenge, in particular due to the necessity of consistency preservation [Fra+18]. This thesis contributes to the goal of preserving consistency of different engineering artifacts or models, and it especially uses and extends the methodology of model transformations to achieve that goal. We relate our research to existing work separated into two categories given by the general goal of consistency preservation, which primarily comprises different approaches to solve the underlying problem but with potentially different methodologies, and the methodology of using transformations for consistency preservation and specific topics regarding transformations, their properties, and their composition.

Figure 14.1 depicts an overview of the topics and research areas that we relate our work to and sketches how they are related to each other and to our contributions, indicated by overlaps of the ellipses representing them. The figure is neither complete nor do the sizes of the areas and overlaps have a specific meaning. We do also not depict the relation of each of our contributions to related topics in that figure, but we do so in the subsequent sections. Several research topics are cross-cutting, such that some work fits into multiple categories. We discuss these works in the areas to which they are mostly related. Parts of the discussions in this chapter have already been published in previous work [Kla18; Kla+19b; KG19; Kla+20; Kla+21].

14.1. Consistency and its Preservation

Checking and preserving consistency of software artifacts, i.e., models, has been researched in several contexts. It covers a broad topic and is often traced back to the *view-update problem*, which considers the backpropagation of changes within a view to the original source and is especially known from

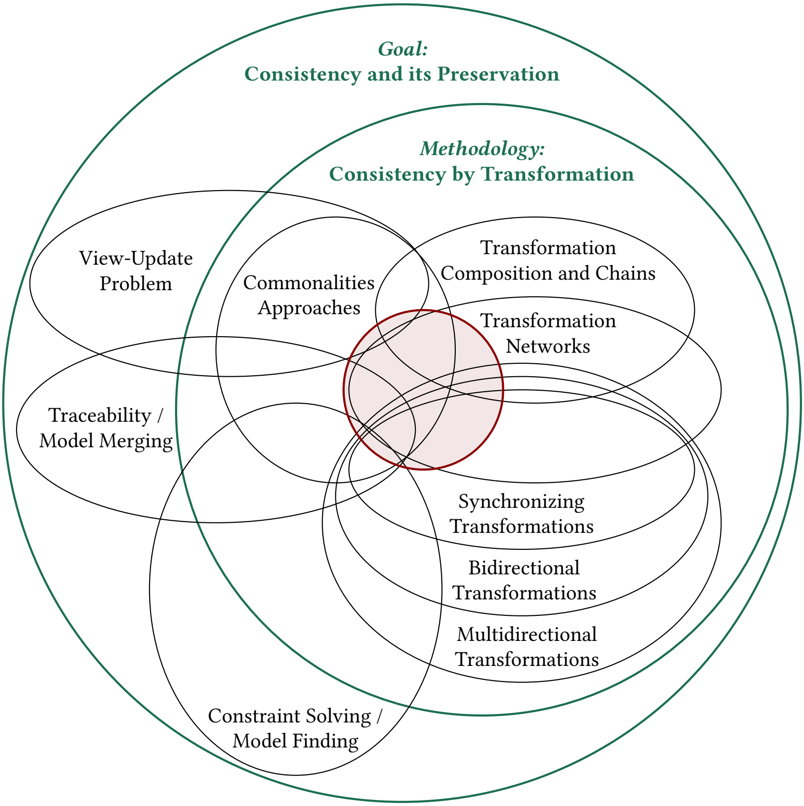


Figure 14.1.: Sketch of different research areas (circles) related to the work of this thesis, their overlaps, and the relation to contributions of this thesis (shaded red in the center).

database engineering [BS81]. Consistency has been considered for different development artifacts, including the common scenario of *roundtrip engineering* between UML models and code [DMW05], and especially rose with the definition of a general methodology defined by the MDA process [MDA]. Depending on the scenario, the kinds of dependencies and inconsistencies between multiple models can vary and have been discussed by Kolovos et al. [KPP08]. Several approaches provide domain-specific solutions for consistency problems, such as for consistency between SysML [SysML] and AUTOSAR [Sch15] in the automotive domain [GHN10].

The development of modeling frameworks, such as the EMF [Ste+09], have enabled the definition of tools, such as transformation languages, that are independent from the actual metamodels to consider consistency between. General methods and approaches regarding model consistency have been based on such modeling frameworks and can be separated into approaches that are only able to check consistency of models [RE12b] and those that are also able to preserve or enforce it. Consistency-preserving approaches range from providing recommendations for repair [Ohr+18] over generation and classifications of repair options [KKE18] to approaches that actually perform *model repair*, which have been subject to intensive research and surveyed by Macedo et al. [MJC17]. The survey also classifies approaches regarding their support for the scenario of keeping multiple, i.e., more than two, models consistent, which is the focus of this thesis. It found that only one of the considered approaches is able to handle multiple models, which is done by considering consistency pairwise, like we do in our work.

We focus the discussion of related work on consistency preservation rather than checking, as the contributions of this thesis aim to support it. We first depict an overview of relevant consistency preservation approaches, including the foundation of the view-update problem, model merging, constraint solving, and the methodology of multi-view modeling.

14.1.1. The View-Update Problem

The *view-update problem* is common in software engineering. It occurs whenever a view, i.e., a model, is supposed to represent information from some underlying source, which in our case is also a model, such that modifications to this view can be propagated back to the underlying source without changing information that is not contained in the view. The problem was and is a central topic in database research [BS81; DB82], where views are derived from database tables. Updating database tables with changes in views to them, denoted as *update translation* [BS81], can be achieved by considering a *complement* view that contains all information of the database that is not contained in the modified view. This means that the Cartesian product of the functions for generating a view and its complement must be injective. Calculating the update of the database after changes to the view can be achieved by inverting this function. There are many possible complements to a view, but for considering a view *updatable*, it must be possible to translate its updates to the database tables with a constant complement [BS81]. Update translation must, however, ensure that it leaves invariant the information in the complement. It is thus inevitable to design views and complements properly to enable automated translation of updates.

An application of the view-update problem to software artifacts and, in particular, to transformations, is given by the *lenses* framework [Fos+05; Fos+07]. It defines two essential operations, which are *get* for deriving a view from a model and *putback* for propagating changes in the view back to the model. This defines a transformation between a view and an underlying model. Specific laws ensure that lenses are *well-behaved* [Fos+07, Def. 3.2], i.e., that they are complete such that all information changed in a view is propagated back to the model, and that they do not perform unintended changes. The proper design of the *putback* function influences expressiveness and robustness of the view and the changes that can be propagated back to the underlying source [Fos+07]. Lenses also depict a well-researched formal foundation to express and study incremental transformations [Ste08b].

While lenses originally consider states of models, *delta lenses* [Dis+11] consider the application to deltas, which conforms to our notion of changes and consistency preservation according to Definition 4.5. They particularly consider the so called *symmetric* case [Dis+11], in which the view is not a projection from the underlying source, but the view and the underlying source are both models with information that is unique to each of them, and thus transformations are defined in both directions.

Lenses have also been extended to the multiary case, in which more than two models need to be kept consistent [DKL18]. It especially reflects that transformations may need to change the originally modified models as well, which is denoted as *reflective updates* in that work and which we have also motivated and introduced with the notion of *synchronizing transformations* in Chapter 6. Despite these multiary lenses, work on lenses is especially focused on or related to bidirectional transformations, which build the basis for our work of constructing networks of them.

14.1.2. Traceability and Model Merging

Traceability is an important concept for different concerns, ranging from comprehension over change impact analysis to the identification and resolution of inconsistencies. For example, architectures based on correspondence models to identify that elements belong together and affect each other among changes have been developed [SC13]. While traces are also used as auxiliary or witness structures for consistency preservation, much work on traceability is focused on consistency checking, such as UML/Analyzer [Egy06] for checking consistency of UML models incrementally, and its generalization Model/Analyzer [Egy11] for checking consistency of arbitrary models. These tools were also extended to repair inconsistencies with repair actions derived from the incremental consistency checks to determine the scope of consistency repair [RE12a]. Our approaches go beyond consistency checking and use traceability especially as a means to trace consistency relations in the practical approach realization to be able to update them among changes.

Model merging goes beyond traceability by not only providing correspondences for related information but by merging elements that share and redundantly represent information. This process is also known as *amalgamation* [KD17]. Model merging consists of matching elements that represent the same information and merging them [KD17]. Such an approach has also been applied in a framework based on category theory [DXC10]. Model merging is comparable to the Commonalities idea (see Chapter 11), as it is also concerned with finding elements that represent the same information. Model merging is, however, usually used for merging models into a single, redundancy-free, and thus inherently consistent representation or for checking consistency during the merge task but not to preserve consistency like we do with the Commonalities approach. In addition, Commonalities relate redundant elements by construction, i.e., as soon as an element is created that requires a corresponding one in another model, it is created, whereas model merging identifies redundant elements after their creation.

14.1.3. Multi-View Modeling

Multi-view modeling, as introduced in Section 2.3, concerns the description of a system by means of multiple views, reflecting different interests. A recent survey of such approaches [CCP19] has identified lacking consistency management as a central challenge of them, which is also emphasized by Reineke et al. [RST19]. In addition to identifying this challenge, Persson et al. [Per+13] classify different types of relations between views to be distinguished. Our contributions can be applied in the context of multi-view modeling, as model transformations are a possible means to solve the consistency challenge in multi-view modeling. We give an overview of different approaches to multi-view modeling, even beyond transformations, to sketch the research field and embed and highlight the relevance of our contributions.

A systematic approach to multi-view modeling is OSM (see Subsection 2.3.1). It considers the description of a system in a single repository, a SUM, from which views are projected that allow modifications that can be propagated back to the SUM. The approach defines how views can be organized in orthographic dimensions representing the different concerns that shape a view. The idea is comparable to a hybrid approach using an underlying metamodel from which multiple views can derived [CCL12], which are consistent through the underlying model by construction. A SUM can be achieved by construction or by applying data integration approaches [Áng+18].

Different ways to construct a SUM, i.e., an underlying repository of consistent information, have been discussed and classified [Mei+19; Mei+20]. Vitruvius, which we have introduced in Subsection 2.3.2, composes a SUM from different models, which conform to metamodels of existing tools and are kept consistent by transformations, and calls this a V-SUM. Role-oriented single underlying models (R-SUMs) let model elements take different roles by separating their properties into different *compartments*, such that depending on the view someone takes on the system only specific compartments are relevant [Wer+18; WA18]. They provide *relation compartments* that can be used to relate information of multiple elements to preserve their consistency. MoConseMi constructs a SUM by metamodel integration [MW18]. It can be considered a model merging approach, which does not only merge the models but also the metamodels by means of operators that check and preserve consistency. While all these approach rely on the idea of multi-view modeling and project views from a single repository, they all ensure consistency of information in the underlying repository in different ways by means of some explicit consistency preservation mechanisms, be they called transformations, operators or something else, such that they all have to deal with the challenges that we have addressed in this thesis. *Action-driven consistency* [AMK20] is a comparable approach, which uses language-specific actions rather than generic change operations, but it is, in fact, only a framework for defining transformations with actions of language-specific semantics.

In general, multi-view modeling considers that one or multiple users work on a single system with different interests reflected by different views. A realization of multi-view modeling with a specific focus on collaborative engineering is the *DesignSpace* approach [Dem+15; Egy+18], which integrates the previously discussed Model/Analyzer approach for checking consistency. It is comparable to a V-SUM approach, but it performs an ad-hoc integration of data and definition of consistency repair rather than applying predefined relations and preservation rules as a V-SUM in the Vitruvius approach does. The DesignSpace approach even integrates consistency preservation capabilities [TME19; KKE19] and especially considers that artifacts may be temporarily inconsistent as well as that inconsistencies have to be resolved in potentially complex processes [Kre+20].

Multi-paradigm modeling [VL03] covers an idea that is comparable to multi-view modeling. It aims at combining multiple modeling formalisms with transformations to avoid redundant specification effort and inconsistencies. It has a particular focus on engineering domains beyond software engineering. In consequence, it also focuses on the runtime state of continuous and hybrid systems rather than the static structure of discrete systems, and it is especially concerned with simulations of a system. Current research especially applies it in the context of cyber-physical systems [CAV20]. Multi-paradigm modeling covers the broad topic of model consistency, especially for cyber-physical systems design, and relies on foundations such as transformations and the construction of networks of them, such that is serves as an application area of our contributions, like multi-view modeling does.

Macromodeling denotes a methodology [SWS12] for defining relations between multiple models for different purposes, ranging from only improving comprehension to consistency management [SME08; SME09]. It is comparable to the notion of *megamodels*, which reflect systems of models and relations, properties, and operations over them [DKM13]. To express relations between models, the application of collection-based operators known from functional programming have been investigated [Sal+15; Sal+20]. Stevens applies megamodel terminology to transformation networks [Ste20a], which we discuss in more detail regarding transformations and networks of them.

Most multi-view modeling approaches, if considering consistency between multiple models and its preservation at all, assume that there is a common knowledge about how all involved models shall be related. When knowledge about relations between views is distributed, like we assume for the construction of transformation networks, and thus the relations between views are defined independently, the problems such as incompatibilities discussed in this thesis can occur. In consequence, regardless of the multi-view modeling approach, the findings of our work are relevant for most of these approaches. Multi-view modeling, including multi-paradigm modeling and SUM approach, are thus an important application area of our contributions.

14.1.4. Constraint Solving and Model Finding

Some approaches consider consistency preservation as a constraint solving problem rather than a transformation problem. They use constraints to represent consistency relations, like we do for the relations of transformations, and then try to find valid solutions after modifications that introduce inconsistencies by model finding. Answer Set Programming (ASP) [CDE06; Era+08] is an approach based on logical programming techniques. Logic programs define the rules and constraints for models, such that consistent models are those fulfilling all of them, which are known as *ground instantiations*. After changes, the ASP engine can deduce consistent sets of models reflecting the given changes and the original states of the models.

Echo [MGC13] is a model repair tool that checks and resolves inconsistencies by model finding. It employs *Alloy*, which is a formal specification language supporting model finding via constraint solving. It can transform Ecore models, as well as OCL expressions, QVT-R transformations, and ATL transformations into Alloy descriptions [MC13; MC16], which applies constraint solving to validate consistency and finds options to restore it.

Constraint solving is a different approach to consistency preservation than transformations, as it relies on declarative specifications of consistency and employs generic solvers to find solutions for inconsistent models. A benefit of using transformations is that they provide more means to influence how consistency is actually achieved. Constraint solving, however, can inherently deal with an arbitrary number of models, as constraints are not restricted to two models, whereas the imperative specification in transformations how consistency between models is restored becomes difficult for more than two models. Since we focus on transformation-based techniques, we depict constraint solving as an alternative technique for consistency preservation, but we do not discuss that research area in mode detail. In addition, it serves as a foundation of our approach for identifying compatibility, in which we use constraint validation techniques.

14.2. Consistency by Model Transformation

We have focused on model transformations as a means to preserve consistency between multiple models, as transformations provide a high degree of freedom for specifying how consistency is preserved. Most existing transformation approaches are restricted to the bidirectional case [Cle+19; WS20], in which two models are kept consistent. Two central approaches for relating multiple metamodels by transformations are transformation networks and multidirectional transformations. They have been discussed in a dedicated Dagstuhl seminar [Cle+19] with a particular focus on the usage of networks of bidirectional transformations and the interaction of several such transformations.

We have identified multidirectional transformations to be complex to specify, whereas networks of bidirectional transformations have limited expressiveness [Ste20b], which, however, may not be practically relevant [Cle+19]. Adding auxiliary models circumvents the limitations of binary relation expressiveness in transformation networks [Ste20b], like we do with the Commonalities approach. Research on transformations is especially driven by theoretic investigations of bidirectional transformations and tools that support their specification. Since reasonable consistency preservation requires incrementality, the area of incremental, bidirectional transformations is most relevant for that purpose. Different scenarios regarding the transformation direction and the scope of changes that need to be propagated between two models have been classified and based on a taxonomy [Dis+16b]. Since our approaches do not make any restrictions regarding the transformation directions or the scope of changes to be considered, our contributions fit into any of the needs for consistency preservation covered by this classification.

14.2.1. Bidirectional Transformations

Stevens emphasizes the importance of bidirectionality for model transformations and for software engineering in general [Ste18]. Although bidirectional transformations themselves are not sufficient for achieving consistency between more than two models, they are still relevant for and related to our work. First, we compose networks of transformations out of bidirectional transformations, thus they serve as a foundation for our work. Second, some approaches already implement necessities for building transformations networks, for example, by matching existing elements to achieve synchronization, like provided by QVT-R. Bidirectionality can be achieved by an explicit specification of consistency preservation in both directions, for example, with imperative languages such as QVT-O, by the specification of one direction and inference of the opposite one [Xio+07; HLR08; Sem+16], or by declaratively specifying constraints that have to hold and inferring the way to preserve it in both directions, like with QVT-R.

Bidirectional transformations are a well-researched option for keeping two models consistent. They have been formally founded on the lenses framework [Ste08b], whose laws have been related to requirements of bidirectional transformations, such as correctness, hippocraticness, or undoability [Ste10]. Correctness and hippocraticness have been identified as essential properties for bidirectional transformations, whereas undoability is beneficial but usually not achievable [Ste10]. We have reflected correctness and hippocraticness in our formalization (see Definition 4.6 and Definition 4.9). Another interesting property is the one of *least change*, which we have discussed in Chapter 7 as an improvement for finding orchestrations. This property has been considered as a basic principle [Che+17] especially by transformation tools [MC16]. Stevens [Ste12] also discusses equivalence relations given by the consistency relations of bidirectional transformations, denoting those instances of one metamodel that are consistent to the same instances of another. They can be considered as an explicit description of different options for a transformation to select from, as discussed in Chapter 7.

Several tools and languages have been developed to support the specification of bidirectional transformations, which have been summarized and classified over the time in several surveys regarding different criteria [Ste08a; DEP12; Kus+13; Jak+14; SZK15; SZK16; Hid+16; Kah+19]. It is a current and open discussion whether specific transformation languages actually provide benefits over using general-purpose languages for specifying model transformations, especially because of lacking evidence and adoption [BCG19]. For our work, it is not important whether a transformation language or a general-purpose language is used to define a transformation, since we only define and consider the properties a transformation has to fulfill, no matter how it is defined. Thus, our contributions are not tied to specific languages or the usage of transformation languages at all.

Popular approaches for specifying bidirectional transformations include imperative and declarative languages, such as the QVT language family [QVT], the ATL [Jou+06; Xio+07] and especially its incremental realization [MTD17], the Epsilon languages [Kol+14] and approaches using them [SZK18], as well as VIATRA [Ber+15; Var+16]. VIATRA is a consistency framework based on an event-driven mechanism, which conforms to our notion of delta-based consistency preservation (see Definition 4.5) and which the authors refer to as *change-driven* transformations [Ber+12]. A different kind of specification is followed by graph-based approaches, such as TGGs, which were originally developed by Schürr [Sch95] and which are well-suited for model transformations [Anj+14]. Several tools for specifying TGGs have been developed [Leb+14], in particular based on the EMF, such as eMoflon [Anj14]. Expressiveness [AVS12] and applicability of TGGs are continuously extended, e.g., in terms of applying integer linear programming to consider consistency as an optimization problem [Wei+19; WA20]. Kramer has proposed an approach combining a language for declarative mappings between metamodels with a fallback language for imperative consistency repair [Kla16; Kra17], which have been developed for the Vitruvius framework [Kla+21]. We have used these languages for the realization of the Commonalities languages and for evaluation purposes throughout this thesis. While all these languages are external DSLs, i.e., they use an own syntax, some languages [Buc18; HB19] are internal DSLs, i.e., they reuse existing languages by providing an internal API and are thus more lightweight.

Extensions to support consistency preservation between more than two models have been proposed for only few tools, which we discuss subsequently. In general, our approaches to build transformation networks can be applied to any existing approach or language for bidirectional transformations. Depending on which assumptions a language makes and which abstraction it provides, different requirements to fulfill our notion of synchronizing transformations have to be considered. First, most languages operate in a state-based manner and thus applying a change to a modified state can be more complex than in a delta-based approach, in which changes can be reapplied to another state of the models. In such a case, approaches for change reconstruction have to be applied, which are especially difficult to develop for textual languages such as code [Fal+14]. Second, most languages do not allow the definition of synchronizing transformations (see Definition 4.7), such that our approach for making transformations synchronizing proposed in Chapter 6 has to be applied, whereas some languages, such as QVT-R, already provide a level of abstraction that achieves synchronization.

14.2.2. Synchronizing Transformations

Transformation networks of arbitrary topology require synchronizing transformations (see Definition 4.7) as a special case of bidirectional transformations. In our definition, this covers transformations that consider changes to both models and are able to update both of them. While in literature the term *concurrent synchronization* always covers this scenario, the term *model synchronization* is used ambiguously for incremental updates [GW09] as well as for concurrent synchronization [SZK15]. Thus, much work on model synchronization is not related to the concurrent modification scenario that we consider. The case of interest is also denoted as *bidirectional synchronization with reconciliation* [Ant08]. Work in this area is especially related to our work on synchronization, as presented in Chapter 6.

EVL+trace [SZK15] considers concurrent modifications of both models related by a transformation. The authors make a case distinction of several scenarios of concurrent changes to support the developer of transformations in considering these different situations of concurrent modifications. They do, however, leave it up to the developer to implement the scenarios. In addition, they consider the case of conflicting user changes, which we have excluded in this thesis as it is not relevant during the execution of a transformation network if transformations are not conflicting, thus making the necessary solution that we have proposed in Chapter 6 simpler.

Approaches for handling concurrent modifications to both models are often concerned with the case of conflicts, i.e., that changes concurrently performed in both models are conflicting. This has, for example, been researched for TGGs [Her+12; OPN20; WFA20]. Orejas et al. [OPN20] proposed an approach that provides different solutions to synchronize concurrent modifications and leaves it up to the developer to decide how conflicts shall be resolved. While this behavior may be desired and beneficial for resolving conflicts of user changes, having multiple transformation results is not applicable in transformation network as the execution has to proceed with a single one. Weidmann et al. [WFA20] propose an approach based on integer linear programming to find consistent solutions after concurrent updates. This approach also handles conflicting changes and could thus be applied in transformation networks to resolve conflicting user inputs. It should, however, not replace the approach we have presented for the synchronization case in transformation networks, as performing the matching of existing elements by construction through encoding it into transformations ensures that matching is performed deterministically and successfully rather than potentially getting unexpected results when considering the scenario as an optimization problem solved by integer linear programming.

One highly related approach to synchronize concurrent changes with bidirectional transformations is given by Xiong et al. [Xio+09; Xio+13]. They propose a certain process of executing a transformation in both directions and merging the generated changes in between with a special three-way merger. While the idea of executing the consistency preservation rules on specific states of the two modified models to reflect concurrent changes is equal to our synchronization approach (see Chapter 6), there are two essential differences. First, their approach merges the changes rather sequentially applying them. Second, their approach does not iteratively apply the preservation rules in both directions to improve partial consistency but assumes to achieve consistency after executing each of them once. Thus, they do not consider that changes to one model may require both models to be changed. Merging the changes rather than sequentially applying them has the benefit that a transformation developer does not have to ensure that elements are not duplicated. The merger must, however, correctly consider that case, which, in general, can only be implemented as a heuristic. The differences between our and the discussed approach especially arise from their different goals. While our approach aims to synchronize concurrent changes performed by transformations, which will not produce conflicts if the transformations are not contradictory, their approach merges user changes, because these changes can, of course, be conflicting and these conflicts need to be resolved.

Design patterns are an established way of defining a common notion for solutions to recurring problems, such as the design patterns for object-oriented software by Gamma et al. [Gam+95]. We have also defined patterns to achieve synchronization of transformations, and several further patterns have been researched for the specification of transformations. This especially comprises patterns for specific kinds of consistency relations [ISH08] and the improvement of modularization [Lan+14]. Patterns for transformations have been surveyed by Lano et al. [Lan+18b], and even ways to semi-formally describe them have been proposed [ESG16]. These patterns focus on improving the development of single transformations and mainly unify how specific kinds of consistency relations can be expressed in transformation languages, but they do not aim to achieve interoperability with other transformations like the patterns that we have proposed do. However, the catalog of Lano et al.

[Lan+14] also comprises patterns for the single instantiation of elements, like we have discussed for achieving synchronizing transformations, but covers a more general use case than the specific scenario of ensuring synchronization of a transformation in a transformation network.

14.2.3. Transformation Networks

Combining multiple transformations, in particular bidirectional transformations, to a network is one approach to preserve consistency between several models. Lämmel has already emphasized the necessity to couple transformation early in the research of model transformations in MDSD [Läm04]. Combining transformations to networks is a task that is external to the individual transformations and languages to define them, which is why existing transformation languages do not consider the combination of transformations developed with them. Stevens [Ste20b] states that it is reasonable to target consistency between multiple models by combining binary transformations, even though multiple binary relations cannot express all relations between multiple models. She also derives the relaxed notion of *binary-implemented* relations, which requires that models consistent to the binary relations need to be consistent to the multiary one but not vice versa.

Favoring transformation networks over multidirectional transformations is motivated by multiple reasons. Networks are easier to develop when domain knowledge is distributed [Kla18], and they are easier to comprehend by a single developer [Cle+19; Ste20b] in comparison to multidirectional transformations. Additionally, binary transformations are researched well and a variety of tools supporting different kinds of specifying them exist, as discussed in the previous subsection. Finally, there is also the problem of technical debt in transformations [Lan+18a], which can be mitigated by modularizing the specification rather than developing a monolithic multidirectional transformation. Research regarding transformation networks especially concerns the orchestration and execution of them and is thus related to our work on orchestration presented in Chapter 7.

Several research papers consider theoretical properties of transformation networks, especially including their resolvability, i.e., the possibility to find a consistent orchestration. While we aim at finding a *universal* approach for orchestrating and executing transformations of arbitrary transformation network topologies, most existing approaches restrict the number of allowed executions. A general approach for a platform managing multiple models [Den+08] considers change propagation based on a dependency graph between the models and performs a depth-first search for determining an execution order. In networks of arbitrary topology, however, no such explicit dependencies exist, and the approach is restricted to executing each transformation only once. Likewise, Di Rocco et al. [Di +17] describe a simple strategy for orchestrating transformation networks, but they also make strong assumptions in terms of the necessity to apply each transformation only once. Stevens [Ste20b] proposes a strategy that also executes each transformation only once in one direction. This includes a notion of authoritative models, which are not allowed to be changed, and does not consider synchronizing transformations. She also discusses non-termination and resolvability issues, i.e., reasons for not finding a consistent orchestration, which can arise from incompatibilities of the relations, as we have discussed in Chapter 5, or further problems such as the selection of different options, as we have discussed in Chapter 7. But that work is restricted to the single execution of each transformation and does not distinguish and discuss the reasons for missing resolvability like we do. In the same way, Stevens [Ste20a] proposes to find an *orientation model* that defines in which direction transformations are executed after a change to restore consistency, also considering authoritative models. However, if there are several transformations that modify the same model, this work leaves it up to the developer to ensure that the transformations are executed in an appropriate order such that all consistency relations hold afterwards. We have presented use cases in which this is too limiting to be used as a universal approaches for orchestration, which is why our approach for orchestration presented in Chapter 7 explicitly considers that an arbitrary number of transformation executions may be necessary.

Provenance is a topic of growing attention and importance in research for bidirectional transformations [Cle+19; AC19]. While Anjorin et al. [AC19] especially consider provenance information about changes performed by a single transformation, we provide such information for the cause of a failure of a transformation network. This affects and supports the network developers rather than the users of a transformation.

One motivation for building transformation networks and our assumptions is the modular reuse of individual transformations. There has been research regarding the reuse of and variability in transformations [Bru+20], supporting the derivation of different transformations from a single specification for different purposes, comparable to product lines. An approach for transformation product lines reuses concepts from software product lines [Lar+18] to derive several transformations with variable parts from one specification. Another approach supporting reuse considers that it is not necessary to define a transformation for two metamodels but only for some requirements that two metamodels have to meet [Lar+19], thus allowing reuse of a transformation for all metamodels fulfilling these requirements. Such approaches for reusability support development processes that cover the assumptions that we have made in our work. Although these works consider quality properties of transformations, such as reuse, which we have discussed in Chapter 10, they are not concerned with quality properties of a transformation network and especially the reuse of transformations in other network.

14.2.4. Transformation Composition and Chains

Transformation composition has especially been researched in terms of creating chains of transformations, composing larger transformations from smaller ones, and finding and extracting common parts in different transformations, known as *factorization*. These approaches deal with specific problems of the execution of and compatibility in transformation networks and are thus related to our work on compatibility and orchestration, which we have presented in Chapter 5 and Chapter 7.

A transformation chain defines a sequence of transformations to represent an MDSD process. It especially covers the case that an abstract model at a high level of abstraction shall be transformed into a model at a low level of abstraction across one or more other models at different abstraction levels, comparable to the idea of the MDA (see Subsection 2.1.3). Transformation chains thus deal with specific kinds of transformation networks. While approaches for transformation chains have in common that they support the specification of such chains, often with dedicated languages, they aim to achieve different additional goals. Tools like UniTI [Van+06; Van+07; Pil+08] enable the explicit specification of chains while treating models as black-boxes, FTG+PM [Lúc+13] provides a complete framework that also aims to model and support processes of applying transformation chains, and CITRIC [Bas+18] especially aims to optimize the automatic selection of transformation chains between two defined metamodels. Transformation chain approaches are currently also applied to low-code development platforms [SDP20]. However, tools like UniTI derive compatibility from additional, external specifications of the transformations, for which conformance to the actual transformations is not guaranteed. Additionally, transformation chains are only a special case of transformation networks, as each transformation network is also aware of the individual transformation chains between all pairs of metamodels. They are, by construction, not that prone to correctness issues, because there are no multiple paths of transformations that can lead to cycles and conflicts in the network, like was our motivation for the Commonalities approach in Chapter 11.

To improve maintainability, approaches for separating transformation chains into smaller concern-specific ones [Yie+12] and to support evolution [Yie+09] have been developed. Other approaches support the incremental development by automated testing [KGZ09]. Etien et al. consider specific properties of transformation chains. They investigate how two transformations with incompatible input and output metamodels can be chained [Eti+10] and how conflicts in terms of results that depend on the execution order can be detected [Eti+12]. A comparable approach validates whether chained transformations fit together in terms of matching contracts and types during both construction and execution [HKA10]. Although these approaches are related to finding interoperability issues and to finding an orchestration for transformations, they particularly aim at checking syntactic compatibility rather than semantic interoperability leading to termination with consistent results, and they do not aim to relieve developers from the task of finding an execution order manually, like we do in our work.

A variety of transformation composition approaches is focused on composing transformations between the same two metamodels. They can be separated into internal and external techniques [Wag08]. Internal techniques are white-box approaches integrated into a language [Wag+11], such as inheritance or superimposition techniques [WVD10]. External approaches consider the transformations as black-boxes and thus work independently from the language. Our approaches can be considered as a combination of white-box and black-box approaches. Achieving synchronization is an intrusive concept that needs to be applied to the implementation of a transformation, thus it is a white-box approach to the transformation. Analyzing compatibility requires knowledge about the relations encoded in the transformations, thus it is not a white-box approach, as it does not consider the actual consistency preservation rules. Considering the consistency relations as a kind of interface, we may denote the compatibility analysis as a gray-box approach.

Finally, the orchestration of a transformation network works under the assumption of having synchronizing, compatible transformations, which are then orchestrated without considering their contents, thus using a black-box approach. The proposed Commonalities approach specifies how to define the internals of transformations and thus represents a white-box approach.

Factorization approaches identify common parts of transformations and extract them into a base transformation from which the individual parts are extended [SG08]. Such approaches use intrusive operators that adapt the transformations for composition, whereas we only provide construction approaches and non-intrusive analyses but do not perform intrusive modifications of the transformations. A recent approach applies higher-order transformations to modularize transformations [Fle+17]. Some approaches also deal with processes for specifying composition, which simply assume interoperability of the individual transformations [Old05].

Existing composition approaches especially have the goal of enhancing modularization of transformations to improve maintainability and reusability, and thus they support composition of transformations between the same metamodels. We, in contrast, combine transformations between different metamodels and with the goal of achieving interoperability rather than maintainability. However, our findings on compatibility can also be applied to composition of transformations between the same metamodels, as compatibility is also a reasonable and relevant notion for a single transformation, as we have identified in our evaluation in Chapter 9.

14.2.5. Multidirectional Transformations

Multidirectional transformations are an alternative to networks of bidirectional transformations. Although they benefit from being less prone to interoperability issues, they do not allow for modular definitions of consistency specifications. Early ideas include the Multi Document Integration (MDI) approach [KS06]. The approach proposes Multi Graph Grammars (MGGs) as an extension of TGGs for defining transformation rules between multiple models. Another extension of TGGs to relate multiple models via one multidirectional transformation rather than defining relations between pairs models are Graph Diagram Grammars [TA15; TA16]. The QVT-R standard [QVT] provides the opportunity to define multidirectional transformations by design, but Macedo et al. [MCP14] reveal ambiguities in the standard that lead to several limitations of its applicability, and they propose strategies to circumvent them.

In contrast to our work, these approaches support the specification of multiary relations between multiple, i.e., more than two, metamodels. Although this allows to preserve consistency between multiple models and although a single multidirectional transformation is, by design, especially less prone to the correctness and compatibility issues discussed in this thesis, it does not support the specification and preservation of consistency under the assumption of distributed knowledge, requiring independent development and modular reuse, which we have made in this thesis to support the motivational process. Multidirectional transformations require a transformation developer to have or acquire knowledge about and be able to express all relations between the involved metamodels. Such approaches may, however, be used to define multidirectional transformations between some of the involved metamodels to be later combined with others to a network. We have depicted the extension of our approaches to construct transformation networks of multidirectional rather than bidirectional transformation as future work.

14.2.6. Commonalities Approaches

Commonalities approaches consider additional auxiliary models in transformation networks, which can be beneficial for different reasons. These reasons range from expressiveness of multiary relations [Ste20b; Stü+18] to engineering methodologies for improving quality properties, like in this work. The classification of Kolovos et al. [KPP08] covers commonalities models as “weave models”, which were originally focused on trace models but also apply to the idea of commonalities models. Work in this area is especially related to our work on the Commonalities approach for improving quality properties of transformation networks, as presented in Chapter 11.

The idea of defining commonalities to express consistency of multiple models was especially researched from a theoretical viewpoint. Not every multiary relation can be expressed by sets of binary relations [Ste20b]. An n-ary consistency relation describing consistency between n metamodels can, however, be decomposed into binary relations to an additional $n{+}1$-th metamodel [Ste20b]. Formal foundations for the construction of commonalities have been based on category theory [Stü+18]. These considerations especially assume one commonalities metamodel, but they may be extended to a hierarchy of them, like we have proposed in this thesis. These foundations have been used to propose a construction approach of commonalities for comprehensive systems [Stü+20]. A formalization of the preservation of multiary consistency relations has been given with the lenses framework [DKL18], which was originally proposed by Foster et al. [Fos+07] and which we have discussed before. All this work has a particular focus on expressiveness of consistency relations rather than engineering considerations such as the improvement of quality properties that we focus on. In addition, if not guaranteeing specific tree structures of commonalities specification, like we have discussed in Chapter 11, a commonalities specification is still a transformation network for which correctness has to be achieved, for example, by applying the approaches proposed in this thesis.

Some existing approaches to practically use commonalities for keeping multiple models consistent are domain-specific. The DUALLy approach [Mal+10; Era+12] uses a domain-specific metamodel of commonalities for architecture description languages to which relations of arbitrary architecture description languages can be defined. DUALLy is based on a generic model consistency approach, which uses ASP based on logical programming techniques. In contrast to such domain-specific solutions, our Commonalities approach can be applied to arbitrary domains and scenarios.

14.2.7. Validation and Verification

Validation and verification is important for transformations to ensure that they do what they are supposed to do. It is cross-cutting to the topics discussed before, since it is relevant for every kind of transformation or composition of them, which is why it is not explicitly depicted in Figure 14.1. Most existing approaches concern correctness of a single transformation rather than correctness of a network of them, as we have considered. They either validate single constraints defined in a transformation or validate a transformation as a whole. Our approach for proving compatibility can be seen as a validation approach for transformation network correctness. In addition, some approaches consider termination criteria for transformations, which is related to our work on orchestration but also concerns a single transformation rather than a network of them.

Several approaches for the validation of OCL constraints used to define conditions on valid models or to define model transformations exist. Kuhlmann et al. [KHG11] and González et al. [Gon+12] have proposed an approach using SAT solvers to validate the existence of models that fulfill specific OCL constraints. Different approaches for the validation of model transformations have been proposed and surveyed [CS13; AW15]. Cabot et al. [Cab+10] derive invariants from transformations for verification purposes, such as to find whether a model that fulfills a transformation rule exists. Comparably, Cuadrado et al. [CGL17] have proposed an approach to analyze ATL transformations for errors in them and to find out whether a source model exists that may trigger a transformation. Other approaches support testing by model comparison [KPP06], regression testing by deriving test cases that ensure that changes to the transformations or their incremental execution are correct [TSR18], or mutation testing [Tro+15]. Rather than using constraint logic for verifying a transformation, an approach by Azizi et al. [AZK17] verifies correctness of transformations written with the Epsilon Transformation Language (ETL) [Kol+14] using the symbolic execution of the transformation. Instead of checking a transformation on its own, Vallecillo et al. [Val+12] have proposed to define a formal specification of transformations against which they can be validated. This is comparable to a validation approach for contracts of transformations, representing contracts as models to be able to apply model validation techniques [BSS14]. Finally, Büttner et al. [BEC12] have proposed an approach for proving correctness of ATL transformations against pre- and postconditions using SMT solvers. Most approaches use some kind of constraint logic or theorem proving for validating correctness of transformations, which is comparable to our approach of proving compatibility of transformation.

Existing works on termination of transformations has especially considered the termination of single graph transformations. They prove termination of transformations [Ehr+05] and use Petri Nets [Var+06] based on criteria for the termination of graph transformation systems. We have considered the termination of transformation networks in terms of the orchestration problem to which we have reduced the halting problem of Turing machines to prove undecidability. The problem could also be considered as a term rewriting problem, in which models states and changes to them may be encoded as terms, which are modified by transformations encoded as a reduction relation. Since termination of rewriting systems is equivalent to termination of Turing machines and thus undecidable [End+11], the results would be the same. Rewriting systems are specifically interesting, because confluence is well-researched in terms of the Church-Rosser theorem. We have, however, argued in Subsection 7.2.5 why confluence is not a desired property of transformation networks.

Our defined notion of compatibility is comparable to correctness notions in the approaches of Cuadrado et al. [CGL17] and Cabot et al. [Cab+10], as they try to figure out whether a rule can be triggered by any model. Nevertheless, all these approaches consider correctness of a single transformation, whereas we consider a correctness notion for complete transformation networks. Only few works, especially on transformation chains, consider validation of transformation networks by means of tests [BK11] but not by means of constructive or analytic approaches that we have proposed in this thesis.

15. Conclusions

We conclude this thesis with a summary of the developed contributions for the problems of achieving correctness of transformation networks and improving their quality properties as well as a summary of central topics for future work. In addition to summarizing the central insights, we focus on bringing them into relation and deriving the overall benefits of these contributions. Limitations and future work have already been discussed in detail within the two evaluations in Chapter 9 and Chapter 13. We thus emphasize general topics of future work that we derive from the overall assumptions made for this thesis and the limitations these assumptions induced to the presented approaches.

15.1. Summary

With our work, we aim to support the construction of transformation networks to enable the evolution of multiple models describing a software-intensive system while ensuring their consistency. We have motivated the necessity to develop such transformations independently and to enable their modular reuse, because knowledge about consistency to be defined in transformations is distributed across several roles and because subsets of the transformations may be reused across multiple projects. In consequence, it was our goal to find assumptions for transformations such that they can be combined with arbitrary other transformations to a network and to find an approach to decide how and in which order to execute them, i.e., to find an orchestration, such that all models are consistent afterwards. We have restricted ourselves to the combination of bidirectional transformations and refer to future work for the combination of multidirectional transformations.

For this context, we have identified two important topics. First and most essentially, transformation networks need to be correct. Thus, we have identified the necessity to define a notion of *correctness* for them and approaches how to achieve it in Part II. Second, as building transformation networks is a software engineering task, not only correctness but also further *quality properties*, such as maintainability, are important. Thus, we have discussed the relevance of certain quality properties, and we have identified how they can be influenced by the way in which a transformation network is specified in Part III.

15.1.1. Correctness of Transformation Networks

We have defined transformation networks as a combination of transformations and functions for determining an execution order of the transformations after a change was performed to a set of models, as well as for applying the transformations in that order. Such a network can be considered correct if for every set of models and changes the application of the transformations in the determined order yields consistent models, provided such an execution order restoring consistency for the inputs exists. This correctness notion consists of three requirements. First, each transformation must be correct on its own. Second, the combination of transformations must preserve consistency according to a non-contradicting notion of consistency. Third, the determined execution order of transformations must ensure that the resulting models are consistent to the consistency notions of all transformations.

Correctness of the individual transformations is a well-defined requirement for bidirectional transformations [Ste10]. Transformations to be used in a transformation network must, however, be *synchronizing*, i.e., they must be able to process changes to both models and update both models they keep consistent. We have thus discussed how transformations can be defined to be synchronizing with existing transformation languages, which only support processing changes to one model and which only update the other model to restore consistency. To this end, we have derived a formal property, for which we have proven to achieve synchronization of bidirectional transformations, and a pattern for practical application, of which we have successfully evaluated completeness and correctness to achieve synchronization in case studies. This approach enables the specification of synchronization transformations with existing transformation languages without the necessity to know about other transformations to later combine the developed ones with.

When knowledge about consistency between models is distributed across multiple roles, these roles can have a contradicting notion of consistency, which can prevent the transformations from finding models that are consistent to all these notions. This is especially the case if the different pairwise notions of consistency induce a global notion among all models that cannot be fulfilled by any set of models. We have defined *compatibility* as a property to reflect when consistency notions are contradicting and proposed a formal approach to validate compatibility, which is proven correct. In addition, we have derived a practical approach for validating compatibility for QVT-R transformations, which operates conservatively, i.e., which is able to prove compatibility for many sets of transformations that are actually compatible but not for all possible transformations because of undecidability of OCL used in QVT-R. In an empirical applicability evaluation of the practical approach, the approach was able to validate compatibility of transformations in 80 % of the cases. Compatibility is a property of a set of transformation and thus its validation requires knowledge about all transformations to be combined. The contributions give systematic knowledge about when transformations cannot be combined properly, and the validation approach even enables transformation network developers to automatically validate their transformations to that effect.

Finally, transformations must be executed in an order such that the resulting models are consistent to the notions of consistency of all transformations. We have identified and defined the *orchestration problem*, which considers finding an orchestration, i.e., an execution order, of the transformations such that the resulting models are consistent whenever such an order exists. We have proven that this can require each transformation to be executed multiple and an even arbitrary high number of times, and that this problem is, in general, undecidable. In addition, we did not find restrictions of the transformations or networks to make the problem decidable and expect it to be unlikely to find such restrictions, as the considered ones were even too restrictive to be practically applicable. In consequence, we have proposed an algorithm that conservatively approaches the problem by only applying transformations when the resulting models are consistent, and in cases in which it fails to find such an orchestration, it supports identifying the reasons for that. These contributions provide the knowledge that a combination of transformations cannot preserve consistency between multiple models in every case, but they also give an algorithm at hand to support transformation network developers and users in identifying the reasons for not finding an orchestration of transformations that preserves consistency.

In conclusion, we have provided an approach to achieve correctness for the individual transformations by construction, an approach to statically validate compatibility of transformations, and an approach to dynamically deal with undecidability of the orchestration problem. In case studies, we have identified missing synchronization to be the most relevant type of mistake, i.e., most occurring failures during transformation network execution were caused by missing synchronization. Since synchronization can be achieved by construction of the individual transformations, most failures can be avoided without knowing about the other transformations to combine the developed one with. In addition, the case studies indicate that the orchestration problem may not be that relevant in practice, as no failures due to it occurred.

Our contributions thus provide systematic knowledge about correctness of transformation networks and the different necessities to achieve it. They enable transformation developers to achieve synchronization, as one of the most important properties in transformation networks, already by construction of the individual transformations, to analyze compatibility of transformations, and to be aware of undecidability of the orchestration problem but also to have an algorithm at hand that eases the identification of the cause whenever transformations are not able to preserve consistency.

15.1.2. Quality Properties of Transformation Networks

Beyond correctness, we have discussed how further quality properties of software systems according to the ISO 25010 standard [I25010] apply to transformation networks. We have identified how they are influenced by the network topology and which of them are contradictory in the sense that determining a specific topology of the transformation network induces a trade-off decision between them. This especially applies to the two essential properties of correctness and reusability of the individual transformations within other transformation networks. We especially found that correctness can be optimized in specific kinds of tree topologies of transformation networks, whereas reusability of the individual transformations is optimized if the network forms a complete graph.

From the insights regarding effects of topologies on properties, we have derived the *Commonalities approach*, which is a construction approach for transformation networks that mitigates these trade-offs by introducing additional auxiliary models. On the one hand, the approach introduces a different way of thinking about consistency in terms of explicitly defining common concepts represented redundantly to be kept consistency rather than implicitly encoding them into rules of transformations. On the other hand, the approach mitigates the trade-off between correctness and reusability.

To support the construction of transformation networks according to the Commonalities approach, we have discussed how a specialized language can support that process and proposed a realization in terms of the *Commonalities language*. It provides a problem-specific, concise syntax for specifying consistency by means of common concepts, from which a compiler then derives an ordinary transformation network.

While the trade-off mitigation, as the essential benefit of the approach, is given by construction if a specific kind of tree topology of the network is achieved, whether such a topology can be achieved in practice was subject to an empirical evaluation by means of a case study. In this evaluation, we have also evaluated the benefits provided by the Commonalities language in terms of reducing the specification effort. The evaluation revealed initial indicators for the practical applicability of the approach and the benefits of the language, but additional studies still need to provide further evidence.

In general, our contributions provide systematic knowledge about the effects of network topologies on quality properties and about their systematic improvement. The Commonalities approach is supposed to be applied only in specific situations, in which consistency actually concerns redundant representations of common concepts, whereas it may not be well applicable when consistency describes more complicated dependencies. In situations for which the approach fits, it gives more guarantees regarding specific quality properties than ordinary transformation networks and thus relieves the transformation developer from ensuring them. Especially in comparison to defining ordinary transformations, the transformation developers must take less care of ensuring correctness of the defined transformation network.

Due to the restriction to those specific situations, it is necessary to enable the combination of a specification using the Commonalities approach with other transformation networks defining consistency, be it in terms of another specification with the Commonalities approach or with ordinary transformations. In consequence, the approaches for building a correct transformation network derived in Part II of the thesis must still be applied when using the Commonalities approach proposed in Part III of the thesis to ensure correctness when combining it with other, ordinary transformations.

15.2. Future Work

The contributions of this thesis provide several detailed opportunities for future work, given by the limitations and specific options for improvement as discussed in the evaluations in Chapter 9 and Chapter 13. In the following, we discuss relevant directions of future work, which need to be followed to make transformation networks applicable for preserving consistency in realistic, complex development scenarios. They especially require the relaxation of some of the assumptions that we have made for this thesis.

Concurrent Editing: We have restricted ourselves to modifications of a single model (see Subsection 3.1.3). In general, multiple developers may modify several models concurrently or even a single developer may modify multiple models at a time. The former scenario could be resolved by reapplying changes of other developers whenever one of them has published his or her modifications, comparable to rebasing commits with Git. For example, if one developer changes an architecture model, which leads to changes to the code through transformations, and he or she publishes his or her changes, another developer, who may have also adapted the code, reapplies his or her changes to the new system state. If these changes to the code are conflicting with the ones performed by transformations to stay consistent with the architecture model, these conflicts need to be resolved manually. It is important that two independent changes together with the changes performed by a transformation network for each of them cannot simply be merged, as there is no guarantee that a merge yields consistent models (see Subsection 6.2.1). In the latter scenario, in which even a single developer may modify multiple models, applying the changes sequentially can, however, also lead to conflicts that need to be resolved by the same developer. Research for considering concurrent modifications within the two models kept consistent by a single transformation already exists, for example, for TGGs [Her+12; OPN20] and in terms of specific algorithms conforming to our notion of synchronization [Xio+13; Xio+09]. Supporting this for transformation networks is, however, subject to future research.

User Decisions: We have introduced transformations to be composed of consistency relations and consistency preservation rules (see Definition 4.7), of which the latter are functions accepting models and changes to them and delivering new changes. In Subsection 1.3.2, we have restricted the considerations of this thesis to the case in which transformations can restore consistency in a fully automated way, i.e., we have assumed the consistency preservation rules to be computable. It may, however, be necessary to require decisions or inputs from users to properly restore consistency. For example, whether a class added to the code is supposed to represent an architectural component or not may not be decidable based on information given within the code but may be a decision of the software architect. Relaxing consistency preservation rules to not necessarily be computable but to involve user decisions has two essential issues to be researched. First, different transformations may require the same decisions, but they would then need to ensure that the user cannot make contradictory decisions, as already discussed in Subsection 9.2.6. This does, however, require transformation developers to align the transformations with each other, which conflicts our assumption of independent development. Second, decisions cannot necessarily be made by the same role who performed the original change. For example, when a software developer adds a class, whether or not it represents a component may be the decision of a software architect. In consequence, the execution of transformation networks can become a long-running process while waiting for necessary decisions of other roles. This requires the definition of a reasonable notion of transactions and considerations of workflows to avoid that a network has to pause somewhere in its execution while waiting for an input. It can even be extended with explorations of the decision space to avoid that if cyclic decisions between several roles are necessary, they have to be asked repeatedly but can instead make speculative decisions based on different options for the decision of another role to be performed later.

Inconsistency Toleration: We have introduced consistency as a total notion (see Definition 4.19), except for the partial notion for the process of repeated execution of transformations to emulate synchronizing behavior (see Subsection 6.3.1). This manifests in our induction assumption in Subsection 4.3.2, in which we assume models to be consistent before applying changes that need to be kept consistent by transformations. In current development processes, the system description, especially for large scale systems, will, however, not always be consistent. This may not always be by accident but can also be intended to share temporarily inconsistent states with other stakeholders. Inconsistencies can be resolved later and potentially by other roles and not necessarily instantly by transformations. It is an open question whether this can or should be covered with relaxed or potentially different levels of consistency notions, or whether tolerating such temporary inconsistencies may not be necessary with future processes enabled by consistency preservation approaches anymore. The former case could even enable further workflows to integrate user decisions by annotating inconsistencies to temporarily inconsistent states, which can be resolved in that state rather than in a workflow that requires an explicit decision of a user. This could enable the definition of different levels of consistency on which development can be performed, in addition to the completely consistent representation of the system and the user-local representation with inconsistencies performed by the user before restoring consistency with transformations. Tolerating inconsistencies and managing uncertainty have already been discussed for bidirectional transformations [EPR15; Ste14; Dis+16a], but transferring this to complete system descriptions and their consistency preservation by networks of transformations has to be considered in future research.

Evidence: Several of our evaluation results lack evidence regarding external validity due to the restriction to few case studies. Although we have argued why and where we expect the results to generalize despite the low number of case studies, further evidence should be provided especially for central insights, such as the relevance of the orchestration problem. Since a realization of such case studies requires significant effort, evidence could especially be provided by community benchmarks, as recently initiated [Anj+20], or by practical applications of transformation networks in industrial cooperation. Then, benefits would not only arise from evidence for the scientific results presented in this thesis but also from the practical usability of the case study.

Appendix

A. Compatibility Proofs

In Section 5.3, we have given Theorem 5.6 for inherent compatibility of consistency relation trees as defined in Definition 5.6. Due to the complexity of the according proof, we have separated it into this appendix.

To prove the statement of Theorem 5.6, we first present a lemma that shows that in a consistency relation tree one can always find an order of the relations such that the classes at the right side of a relation do not overlap with the classes at the left side of a relation that preceded in the order, i.e., there is no cycle between classes in the relations.

Lemma A.1 (Consistency Relation Tree Unique Paths)
Let $\mathbb{CR} = \{CR_1, CR_1^T, \ldots, CR_k, CR_k\}^T$ be a symmetric, connected set of consistency relations. $\mathbb{CR}$ is a consistency relation tree if, and only if, for each $CR \in \mathbb{CR}$ exists a sequence $\mathbb{CR}'[] = [CR'_1, \ldots, CR'_k]$ with $CR'_1 = CR$ that contains for each i either CR_i or CR_i^T, i.e.,

$$\forall\, i \in \{1, \ldots, k\} : \big[\big(CR_i \in \mathbb{CR}'[] \land CR_i^T \notin \mathbb{CR}'[]\big) \\ \lor \big(CR_i^T \in \mathbb{CR}'[] \land CR_i \notin \mathbb{CR}'[]\big)\big]$$

such that:

$$\forall\, s \in \{1, \ldots, k-1\} : \forall\, t \in \{s+1, \ldots, k\} : \\ \big(\mathfrak{C}_{r,CR'_s} \cap \mathfrak{C}_{r,CR'_t} = \emptyset \land \mathfrak{C}_{l,CR'_s} \cap \mathfrak{C}_{r,CR'_t} = \emptyset\big)$$

Proof. We start with the forward direction. Given a consistency relation tree $\mathbb{CR}$, we show that a sequence according to the requirements in Lemma A.1 exists by constructing such a sequence $\mathbb{CR}'[] = [CR'_1, \ldots, CR'_k]$ for any $CR \in \mathbb{CR}$. We begin with any $CR'_1 = CR \in \mathbb{CR}$ and inductively add further relations to that sequence. We take any consistency relation $CR_s = CR_{s,1} \otimes$

$\ldots \otimes CR_{s,m} \in \mathbb{CR}^+$ with $\mathfrak{C}_{l,CR_{s,1}} \subseteq \mathfrak{C}_{r,CR}$. Such a sequence exists because of $\mathbb{CR}$ being connected. Then we add all $CR_{s,1}, \ldots, CR_{s,m}$ to the sequence, such that we have $[CR, CR_{s,1}, \ldots, CR_{s,m}]$, which fulfills both requirements to that sequence in Lemma A.1 by definition. The addition of further consistency relations can be applied inductively. We take any other consistency relation $CR_t = CR_{t,1} \otimes \ldots \otimes CR_{t,n} \in \mathbb{CR}^+$ such that:

$$\exists\, CR' \in \{CR, CR_{s,1}, \ldots, CR_{s,m}\} : \mathfrak{C}_{l,CR_{t,1}} \subseteq \mathfrak{C}_{r,CR'}$$
$$\wedge\, CR_{t,1}, CR_{t,1}^T \notin \{CR, CR_{s,1}, \ldots, CR_{s,m}\}$$

In other words, we take any concatenation in the transitive closure of $\mathbb{CR}$ that starts with a relation with a left class tuple that is contained in a right class tuple of a relation already added to the sequence. Again, such a sequence must exist because of $\mathbb{CR}$ being connected and, again, we add all $CR_{t,1}, \ldots, CR_{t,n}$ to the sequence. Per construction, for each CR' in the sequence, a non-empty concatenation of relations within the sequence $CR \otimes \ldots \otimes CR'$ exists, because relations were added in a way so that such a concatenation always exists. Since all relations in the sequence are contained in $\mathbb{CR}$, such a concatenation was also contained in $\mathbb{CR}^+$. First (1.), we show that the sequence still contains no duplicate elements, i.e., that none of the $CR_{t,i}$ or $CR_{t,i}^T$ is already contained in the sequence $[CR, CR_{s,1}, \ldots, CR_{s,m}]$. Second (2. ,3.), we show that both further conditions for the sequence defined in Lemma A.1 are still fulfilled for the sequence $[CR, CR_{s,1}, \ldots, CR_{s,m}, CR_{t,1}, \ldots, CR_{t,n}]$.

1. Let us assume that $[CR, CR_{s,1}, \ldots, CR_{s,m}]$ already contained one of the $CR_{t,i}$ or $CR_{t,i}^T$. If the sequence contained $CR_{t,i}$, there is a non-empty concatenation $CR \otimes \ldots \otimes CR_{t,i}$ of relations in $[CR, CR_{s,1}, \ldots, CR_{s,m}]$. In addition, the concatenation $CR \otimes \ldots \otimes CR_{t,1} \otimes \ldots \otimes CR_{t,i}$ is non-empty by selection in our construction approach. Since $CR_{t,1} \notin \{CR, CR_{s,1}, \ldots, CR_{s,m}\}$ by construction, these two concatenations are not identical but relate the same class tuples, i.e., they contradict the definition of a consistency relation tree. If $CR_{t,i}^T$ was contained in the sequence $[CR, CR_{s,1} \otimes \ldots \otimes CR_{s,m}]$, there is a non-empty concatenation $CR \otimes \ldots \otimes CR_w \otimes CR_{t,i}^T$ of relations in $[CR, CR_{s,1}, \ldots, CR_{s,m}]$, and, like before, the concatenation $CR \otimes \ldots \otimes CR_{t,1}, \ldots, CR_{t,i}$ is non-empty by construction. Due to $\mathfrak{C}_{r,CR_w} \cap \mathfrak{C}_{l,CR_{t,i}^T} \neq \emptyset$ (with $\mathfrak{C}_{l,CR_{t,i}^T} = \mathfrak{C}_{r,CR_{t,i}}$) and $CR_{t,1}^T \notin \{CR, CR_{s,1}, \ldots, CR_{s,m}\}$ by construction, the two concatenations $CR \otimes \ldots \otimes CR_w$ and $CR \otimes \ldots \otimes CR_{t,1} \otimes \ldots \otimes CR_{t,i}$ have an overlap in both their left and right class tuples, i.e., they con-

tradict the definition of a consistency relation tree. In consequence, $[CR, CR_{s,1}, \ldots, CR_{s,m}]$ cannot have contained any $CR_{t,i}$ or $CR^T_{t,i}$ before.

2. Let us assume that $[CR, CR_{s,1}, \ldots, CR_{s,m}, CR_{t,1}, \ldots, CR_{t,n}]$ contains any CR'_x and CR'_y such that $\mathfrak{C}_{r,CR'_x} \cap \mathfrak{C}_{r,CR'_y} \neq \emptyset$. As discussed before, for each of these relations exists a non-empty concatenation of relations $CR \otimes \ldots \otimes CR'_x$ and $CR \otimes \ldots \otimes CR'_y$ in the sequence $[CR, CR_{s,1}, \ldots, CR_{s,m}, CR_{t,1}, \ldots, CR_{t,n}]$ that is contained in $\mathbb{CR}^+$. This contradicts the definition of a consistency relation tree, so there cannot be two such relations with overlapping right class tuple.

3. Let us assume that $[CR, CR_{s,1}, \ldots, CR_{s,m}, CR_{t,1}, \ldots, CR_{t,n}]$ contains any CR'_x and CR'_y with $x < y$ such that $\mathfrak{C}_{l,CR'_x} \cap \mathfrak{C}_{r,CR'_y} \neq \emptyset$. Again per construction, there must be a non-empty concatenation $CR \otimes \ldots \otimes CR'_w \otimes CR'_x$ with $w < x$. Since $\mathfrak{C}_{l,CR'_x} \subseteq \mathfrak{C}_{r,CR'_w}$ per definition, it holds that $\mathfrak{C}_{r,CR'_w} \cap \mathfrak{C}_{r,CR'_y} \neq \emptyset$. We have already shown in (2.) that this contradicts the definition of a consistency relation tree.

The previous strategy for adding relations to the sequence can be continued inductively by adding relations of the transitive closure of $\mathbb{CR}$ if their relations were not yet added to the sequence. This process can be continued until finally all relations in $\mathbb{CR}$ are added to the sequence. Inductively applying the same arguments as before, the final sequence still fulfills all requirements for the sequence in Lemma A.1.

We proceed with the reverse direction, i.e., given a sequence according to the requirements in Lemma A.1 for all $CR \in \mathbb{CR}$, we show that the set of consistency relations fulfills the definition of a consistency relation tree. Let us assume that the tree definition was not fulfilled, i.e., that there were two consistency relations $CR_s = CR_{s,1} \otimes \ldots \otimes CR_{s,m} \in \mathbb{CR}^+$ and $CR_t = CR_{t,1} \otimes \ldots \otimes CR_{t,n} \in \mathbb{CR}^+$ such that $\mathfrak{C}_{l,CR_s} \cap \mathfrak{C}_{l,CR_t} \neq \emptyset$ and $\mathfrak{C}_{r,CR_s} \cap \mathfrak{C}_{r,CR_t} \neq \emptyset$. Without loss of generality, we assume that $CR_{s,m} \neq CR_{t,n}$, because if these last relations are the same, the previous relations $CR_{s,m-1}$ and $CR_{t,n-1}$ must have an overlap in the classes at the right side and thus we could instead consider the sequences without those last relations and still fulfill the defined requirements. Any sequence according to Lemma A.1 containing both $CR_{s,m}$ and $CR_{t,n}$ would contradict the assumption, because $\mathfrak{C}_{r,CR_{s,m}} \cap \mathfrak{C}_{r,CR_{t,n}} \neq \emptyset$ in contradiction to the assumptions regarding the sequence. Thus, the sequence has to contain either $CR^T_{s,m}$ or $CR^T_{t,n}$. Let us assume that the sequence contains $CR^T_{s,m}$. Then the sequence cannot contain

$CR_{s,m-1}$, because $\mathfrak{C}_{r,CR^T_{s,m}} \cap \mathfrak{C}_{r,CR_{s,m-1}} \neq \emptyset$, which, again, would contradict the assumptions regarding the sequence. This argument can be inductively applied to all $CR_{s,i}$, such that the sequence has to contain all $CR^T_{s,i}$. Since the sequence contains $CR^T_{s,1}$, it must contain $CR_{t,1}$, because $\mathfrak{C}_{r,CR^T_{s,1}} \cap \mathfrak{C}_{r,CR^T_{t,1}} \neq \emptyset$. In consequence of $CR_{t,1}$ being contained in the sequence, all $CR_{t,i}$ have to be contained as well for the same reasons as before. So we have these conditions, which introduce a cycle in the overlaps of the class tuples of the relations within the sequence:

$$\begin{aligned}
&\mathfrak{C}_{l,CR^T_{s,i-1}} \cap \mathfrak{C}_{r,CR^T_{s,i}} \neq \emptyset \wedge \mathfrak{C}_{l,CR_{t,1}} \cap \mathfrak{C}_{r,CR^T_{s,1}} \neq \emptyset \\
&\wedge \mathfrak{C}_{l,CR_{t,i}} \cap \mathfrak{C}_{r,CR_{t,i-1}} \neq \emptyset \wedge \mathfrak{C}_{l,CR^T_{s,m}} \cap \mathfrak{C}_{r,CR_{t,n}} \neq \emptyset
\end{aligned}$$

Because of that cycle in the overlap of class tuples, there is no order of these relations $[CR''_1, \ldots, CR''_{m+n}]$ such that for all of them it holds that $\mathfrak{C}_{l,CR''_u} \cap \mathfrak{C}_{r,CR''_w} \neq \emptyset$ $(u < w)$, which contradicts the assumptions regarding the sequence in Lemma A.1. The analogous argument holds when we assume that the sequence contains $CR^T_{t,n}$ instead of $CR^T_{s,m}$. In consequence, there cannot be two such concatenations CR_s and CR_t without breaking the assumptions for the sequence in Lemma A.1. □

The previous lemma shows that the definition of consistency relation trees in Definition 5.6 is equivalent to the possibility to find sequences of the relations that do not contain cycles in the related class tuples. We can now show that a consistency relation tree is always compatible by a constructive proof that requires the equivalent definition from Lemma A.1. We have defined this statement in Theorem 5.6 and now provide the according proof.

Proof. We prove the statement by constructing a tuple of models for each condition element in the left condition of each consistency relation. This model tuple contains the condition element and is consistent, i.e., it fulfills the compatibility definition. The basic idea is that because $\mathbb{CR}$ is a consistency relation tree, we can simply add necessary elements to get a model tuple that is consistent to all consistency relations by following an order of relations according to Lemma A.1. Thus, we explain an induction for constructing such a model tuple, which is also exemplified for a simple scenario in Figure 5.12, which is based on the relations in the consistency relation tree in Figure 5.11.

Base Case: Take any $CR \in \mathbb{CR}$ and any condition element of the left-side condition $\mathfrak{c}_l = \langle o_{l,1}, \ldots, o_{l,m} \rangle \in \mathbb{c}_{l,CR}$. Select any $\mathfrak{c}_r = \langle o_{r,1}, \ldots, o_{r,n} \rangle \in \mathbb{c}_{r,CR}$, such that $\mathfrak{c}_l$ and $\mathfrak{c}_r$ constitute a consistency relation pair $\langle \mathfrak{c}_l, \mathfrak{c}_{r,} \rangle \in CR$. We now construct the model tuple $\mathfrak{m} = \{o_{l,1}, \ldots, o_{l,m}, o_{r,1}, \ldots, o_{r,n}\}$. In consequence, we have a minimal model tuple $\mathfrak{m}$, such that $\mathfrak{m}$ *contains* $\mathfrak{c}_l$ and $\mathfrak{m}$ *consistent to* CR. Additionally, $\mathfrak{m}$ is consistent to CR^T due to symmetry of CR and CR^T: It is $\mathfrak{c}_r \in \mathbb{c}_{l,CR^T}$ and $\langle \mathfrak{c}_r, \mathfrak{c}_l \rangle \in CR^T$ and no other condition element of $\mathbb{c}_{l,CR^T}$ is contained in $\mathfrak{m}$ by construction, thus $\mathfrak{m}$ is consistent to CR^T. In consequence, we know for all $CR \in \mathbb{CR}$ that $\{CR, CR^T\}$ is compatible. Considering the example in Figure 5.12, for the selection of any person as a condition element in $\mathbb{c}_{l,CR_1}$ (1), we select a resident in $\mathbb{c}_{r,CR_1}$ with the same *name* (2), such that the elements are consistent to CR_1.

Induction Assumption: We know from Lemma A.1 that for the relations in $\mathbb{CR}$ there is a sequence $[CR_1, \ldots, CR_k]$ with $CR_1 = CR$ such that:

$$\forall s \in \{1, \ldots, k-1\} : \forall t \in \{s+1, \ldots, k\} :$$
$$\left(\mathfrak{C}_{r,CR'_s} \cap \mathfrak{C}_{r,CR'_t} = \emptyset \land \mathfrak{C}_{l,CR'_s} \cap \mathfrak{C}_{r,CR'_t} = \emptyset\right)$$

Considering the example in Figure 5.12, such a sequence would be $[CR_1, CR_2]$, because the elements in the right condition of CR_2 are not represented in the left condition of CR_1. We assume that for some $i < k$ we know that $\{CR_1, CR_1^T, \ldots, CR_i, CR_i^T\}$ is compatible. Then for every $\mathfrak{c}_l \in \mathbb{c}_{l,CR}$ we can find a model tuple $\mathfrak{m}$ that contains $\mathfrak{c}_l$ and that is consistent to $\{CR_1, CR_1^T, \ldots, CR_i, CR_i^T\}$. We can especially create a minimal model by our construction for the base case and the following induction step.

Induction Step: We consider CR_{i+1}. There is at most one condition element $\mathfrak{c}_l \in \mathbb{c}_{l,CR_{i+1}}$ with $\mathfrak{m}$ *contains* $\mathfrak{c}_l$. If there were at least two condition elements $\mathfrak{c}_l, \mathfrak{c}'_l \in \mathbb{c}_{l,CR_{i+1}}$ that are both contained in $\mathfrak{m}$, then by construction there is a consistency relation CR_s $(s < i+1)$ with $\mathfrak{c}_l, \mathfrak{c}'_l \in \mathbb{c}_{r,CR_j}$. Let us assume there were two consistency relations CR_s, CR_t, each containing one of the condition elements in the right condition, then there would be non-empty concatenations $CR \otimes \ldots \otimes CR_s$ and $CR' \otimes \ldots \otimes CR_t$ with $\mathfrak{C}_{l,CR} \cap \mathfrak{C}_{l,CR'} \neq \emptyset$, because we started the construction with elements from the left condition of CR and so every element is contained in the models because of a relation to those elements, and with $\mathfrak{C}_{r,CR_s} \cap \mathfrak{C}_{r,CR_t} \neq \emptyset$, because both condition elements $\mathfrak{c}_l$ and $\mathfrak{c}'_l$ instantiate the same classes, as they are both contained in $\mathbb{c}_{l,CR_{i+1}}$. This would violate Definition 5.6 for a consistency relation tree, thus there is only one such consistency relation CR_s. Consequently, there must be two condition elements $\mathfrak{c}_{ll}, \mathfrak{c}'_{ll} \in \mathbb{C}_{l,CR_s}$ with $\langle \mathfrak{c}_{ll}, \mathfrak{c}_l \rangle, \langle \mathfrak{c}'_{ll}, \mathfrak{c}'_l \rangle \in CR_s$, because, per construction, $\mathfrak{m}$ was consistent to CR_s and so there must be a witness structure with a unique mapping between condition elements contained in $\mathfrak{m}$. The above argument can be applied inductively until we find that there must be two condition elements $\mathfrak{c}_{lll}, \mathfrak{c}'_{lll} \in \mathbb{C}_{l,CR}$ that are contained in $\mathfrak{m}$. This is excluded by construction, as we started with only one element from $\mathbb{C}_{l,CR}$, so there is only one such condition element $\mathfrak{c}_l \in \mathbb{C}_{l,CR_{i+1}}$ with $\mathfrak{m}$ *contains* $\mathfrak{c}_l$.

For this $\mathfrak{c}_l \in \mathbb{C}_{l,CR_{i+1}}$, we select an arbitrary $\mathfrak{c}_r = \langle o_1, \ldots, o_s \rangle \in \mathbb{C}_{r,CR_{i+1}}$ such that $\langle \mathfrak{c}_l, \mathfrak{c}_r \rangle \in CR_{i+1}$. Now we create a model tuple $\mathfrak{m}' = \mathfrak{m} \cup \{o_1, \ldots, o_s\}$. Since $\mathfrak{c}_l$ is the only left condition element of CR_{i+1} that $\mathfrak{m}$ contains, $\mathfrak{m}'$ is consistent to CR_{i+1} per construction. $\mathfrak{m}'$ is also consistent to CR^T_{i+1}, since the symmetry of CR_{i+1} and CR^T_{i+1} implies $\mathfrak{c}_r \in \mathbb{C}_{l,CR^T_{i+1}}$, and due to $\langle \mathfrak{c}_r, \mathfrak{c}_l \rangle \in CR^T_{i+1}$ a consistent corresponding element exists in $\mathfrak{m}'$. Furthermore, there cannot be any other $\mathfrak{c}' \in \mathbb{C}_{l,CR^T_{i+1}}$ with $\mathfrak{m}'$ *contains* $\mathfrak{c}'$, because otherwise there would have been another consistency relation CR' that required the creation of $\mathfrak{c}'$, which means that there are two concatenations of consistency relations $CR \otimes \ldots \otimes CR'$ and $CR \otimes \ldots \otimes CR_{i+1}$ that both relate instances of the same classes, which contradicts Definition 5.6 for a consistency relation tree.

Additionally, we know the following for all CR_s $(s < i+1)$ due to Lemma A.1: First, it is $\mathfrak{C}_{l,CR_s} \cap \mathfrak{C}_{r,CR_{i+1}} = \emptyset$. Since the newly added elements $\mathfrak{c}_r$ are part of $\mathbb{C}_{r,CR_{i+1}}$, these elements cannot match the left condition of CR_s. So $\mathfrak{m}'$ is still consistent to all CR_s $(s < i+1)$. Second, it is $\mathfrak{C}_{r,CR_s} \cap \mathfrak{C}_{r,CR_{i+1}} = \emptyset$. Again, since the newly added elements $\mathfrak{c}_r$ are part of $\mathbb{C}_{r,CR_{i+1}}$, these elements cannot match the left condition of CR^T_s. So $\mathfrak{m}'$ is still consistent to all CR^T_s $(s < i+1)$. In consequence, we know that $\mathfrak{m}'$ *consistent to* $\{CR_1, CR^T_1, \ldots, CR_{i+1}, CR^T_{i+1}\}$.

Considering the example in Figure 5.12, we would select CR_2 and add for the resident, which is in the left condition elements of CR_2, an appropriate employee to make the model tuple consistent to CR_2 (3).

Conclusion: Taking the base case for CR and the induction step for CR_{i+1}, we have inductively shown that

$$\mathfrak{m}' \textit{ consistent to } \{CR_1, CR^T_1, \ldots, CR_k, CR^T_k\} = \mathbb{CR}$$

Since the construction is valid for each condition element in every relation in $\mathbb{CR}$, we know that a consistency relation tree $\mathbb{CR}$ is compatible. $\square$

B. Verifiability

Along with the depiction of our approaches and their evaluation, we have referred to their realization artifacts, which are available at GitHub. In particular, most of the evaluation results can be reproduced with the case studies provided in those repositories. Since these repositories evolve, we have annotated the date of the repository state that we have last used for the evaluation in the bibliography. In addition, we provide a reproduction package [Kla21] that contains all artifacts in that state together with an according environment that eases the reproduction of the results to improve long-term reproducibility.

We have developed four kinds of artifacts. They comprise a realization of the approach for validating compatibility of transformations, a simulator for transformation networks, an evaluation of the categorization of errors in transformation networks and approaches to resolve them, and finally the Commonalities language and a comprehensive case study for its evaluation.

We have realized a decomposition approach for the validation of compatibility (see Subsection 5.1.2) [GitDec]. The implementation validates compatibility of given QVT-R transformations defined with the Eclipse implementation QVTd [EcQVT]. The case study presented in Section 9.1 is implemented in terms of test cases in the according repository.

We have implemented a simulator for transformation networks, in which different scenarios of transformations and models to which they are applied can be executed step by step (see Section 7.3) [GitSim]. The implementation provides a predefined set of transformation networks and model states to apply them to, which can be extended by further scenarios. It is realized as a web-based visualization of the network execution process.

For the case studies on error categorization and synchronization of transformations (see Section 9.2), the prototypical implementation of the Commonalities language (see Chapter 12), and its evaluation (see Chapter 13), we have employed and extended the Vitruvius framework [Kla+21]. The Commonalities language has been realized as an additional language in the Vitruvius framework repository [GitVit], next to the existing Reactions and Mappings languages. The case studies have been realized based on transformations and test scenarios implemented in the case study repository for component-based system development [GitApp].

The contributions and case studies for the Vitruvius framework have last been validated with release version 2.0.1 of the framework and release version 0.2.0 of the case studies repository. Results may also be reproducible with later versions, but the framework behavior may change and the case studies may be developed further, such that the absolute result values will differ although the same conclusions should be derivable from them. To support the reproduction of the results with the Vitruvius framework presented in this thesis, the reproduction package [Kla21] contains the depicted artifact versions and a Docker-based execution environment to ease their setup.

Several evaluation results, especially regarding the categorization and resolution of errors in Section 9.2, depend on the execution of a process. This process starts with independently developed transformations, combines them to a network, and fixes faults revealed by occurring execution failures. The states of this process during the development have been tagged in the case studies repository [GitApp], but they may be difficult to reproduce in detail, as they depended on the framework at that time. However, the mentioned versions of the artifacts and especially the reproduction package contain the final state after performing the depicted process, in which all faults have been fixed.

Bibliography

The titles of most entries are hyperlinks resolving the DOIs or pointing to other online sources for the entries.

[AC19] A. Anjorin and J. Cheney. "Provenance Meets Bidirectional Transformations". In: *11th International Workshop on Theory and Practice of Provenance (TaPP 2019)*. USENIX Association, 2019, pp. 1–5.

[AGK14] C. Atkinson, R. Gerbig, and T. Kühne. "Comparing multi-level modeling approaches". In: *1st Workshop on Multi-Level Modelling (MLM) co-located with 17th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*. Vol. 1286. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 53–61.

[AK03] C. Atkinson and T. Kühne. "Model-driven development: a metamodeling foundation". In: *IEEE Software* 20.5 (2003), pp. 36–41.

[AK08] C. Atkinson and T. Kühne. "Reducing accidental complexity in domain models". In: *Software and Systems Modeling* 7.3 (2008), pp. 345–359.

[AMK20] H. Ali, G. Mussbacher, and J. Kienzle. "Action-Driven Consistency for Modular Multi-Language Systems with Perspectives". In: *12th System Analysis and Modelling Conference (SAM 2020)*. ACM, 2020, pp. 95–104.

[Áng+18] M. S. Ángel, J. de Lara, P. Neubauer, and M. Wimmer. "Automated modelling assistance by integrating heterogeneous information sources". In: *Computer Languages, Systems & Structures* 53 (2018), pp. 90–120.

[Anj+14] A. Anjorin, S. Rose, F. Deckwerth, and A. Schürr. "Efficient Model Synchronization with View Triple Graph Grammars". In: *10th European Conference on Modelling Foundations and Applications (ECMFA 2014)*. Vol. 8569. LNCS. Springer International Publishing, 2014, pp. 1–17.

[Anj+20] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H.-S. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, and A. Zündorf. "Benchmarking bidirectional transformations: theory, implementation, application, and assessment". In: *Software and Systems Modeling* 19.3 (2020), pp. 647–691.

[Anj14] A. Anjorin. "Synchronization of Models on Different Abstraction Levels using Triple Graph Grammars". PhD thesis. Technische Universität Darmstadt, 2014.

[Ant08] K. Antkiewicz Michałand Czarnecki. "Design Space of Heterogeneous Synchronization". In: *International Summer School Generative and Transformational Techniques in Software Engineering II (GTTSE 2007)*. Vol. 5235. LNCS. Springer Berlin Heidelberg, 2008, pp. 3–46.

[Arm+11] E. Armengaud, M. Zoier, A. Baumgart, M. Biehl, D. Chen, G. Griessnig, C. Hein, T. Ritter, and R. Tavakoli Kolagari. "Model-based Toolchain for the Efficient Development of Safety-Relevant Automotive Embedded Systems". In: *SAE 2011 World Congress & Exhibition*. SAE International, 2011.

[ASB10] C. Atkinson, D. Stoll, and P. Bostan. "Orthographic Software Modeling: A Practical Approach to View-Based Development". In: *3rd and 4th International Conferences on Evaluation of Novel Approaches to Software Engineering (ENASE 2008/2009)*. Vol. 69. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2010, pp. 206–219.

[ASCET] ETAS Group. *ASCET-DEVELOPER*. URL: `https://www.etas.com/ascet` (visited on 01/15/2021).

[ATM15] C. Atkinson, C. Tunjic, and T. Möller. "Fundamental Realization Strategies for Multi-view Specification Environments". In: *19th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2015)*. IEEE, 2015, pp. 40–49.

[AVS12] A. Anjorin, G. Varró, and A. Schürr. "Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques". In: *Electronic Communications of the EASST* 49 (2012), pp. 1–16.

[AW15] L. Ab. Rahim and J. Whittle. "A survey of approaches for verifying model transformations". In: *Software and Systems Modeling* 14.2 (2015), pp. 1003–1028.

[AZK17] B. Azizi, B. Zamani, and S. Kolahdouz-Rahimi. "Contract verification of ETL transformations". In: *7th International Conference on Computer and Knowledge Engineering (ICCKE 2017)*. IEEE, 2017, pp. 154–160.

[Bas+18] F. Basciani, D. Di Ruscio, M. D'Emidio, D. Frigioni, A. Pierantonio, and L. Iovino. "A Tool for Automatically Selecting Optimal Model Transformation Chains". In: *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS 2018)*. ACM, 2018, pp. 2–6.

[BCD05] D. Berardi, D. Calvanese, and G. De Giacomo. "Reasoning on UML class diagrams". In: *Artificial Intelligence* 168.1 (2005), pp. 70–118.

[BCG19] L. Burgueño, J. Cabot, and S. Gérard. "The Future of Model Transformation Languages: An Open Community Discussion". In: *Journal of Object Technology* 18.3 (2019), 7:1–11.

[BEC12] F. Büttner, M. Egea, and J. Cabot. "On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers". In: *15th International Conference Model Driven Engineering Languages and Systems (MODELS 2012)*. Vol. 7590. LNCS. Springer Berlin Heidelberg, 2012, pp. 432–448.

[Ber+12] G. Bergmann, I. Ráth, G. Varró, and D. Varró. "Change-driven model transformations". In: *Software and Systems Modeling* 11.3 (2012), pp. 431–461.

[Ber+15] G. Bergmann, I. Dávid, Á. Hegedüs, Á. Horváth, I. Ráth, Z. Ujhelyi, and D. Varró. "Viatra 3: A Reactive Model Transformation Platform". In: *8th International Conference on Theory and Practice of Model Transformations (ICMT 2015)*. Vol. 9152. LNCS. Springer International Publishing, 2015, pp. 101–110.

[Bet16] L. Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition.* 2nd. Packt Publishing, 2016.

[Béz05] J. Bézivin. "On the unification power of models". In: *Software and Systems Modeling* 4.2 (2005), pp. 171–188.

[BFT17] C. Barrett, P. Fontaine, and C. Tinelli. *The SMT-LIB Standard: Version 2.6.* Tech. rep. Department of Computer Science, The University of Iowa, 2017.

[BK11] E. Bauer and J. M. Küster. "Combining Specification-Based and Code-Based Coverage for Model Transformation Chains". In: *4th International Conference on Theory and Practice of Model Transformations (ICMT 2011).* Vol. 6707. LNCS. Springer Berlin Heidelberg, 2011, pp. 78–92.

[BKS02] B. Beckert, U. Keller, and P. H. Schmitt. "Translating the Object Constraint Language into First-order Predicate Logic". In: *2nd Verification Workshop (VERIFY) co-located with Federated Logic Conferences (FLoC 2002).* 2002, pp. 113–123.

[Bro87] F. P. Brooks. "No Silver Bullet Essence and Accidents of Software Engineering". In: *IEEE Computer* 20.4 (1987), pp. 10–19.

[Bru+20] J.-M. Bruel, B. Combemale, E. Guerra, J.-M. Jézéquel, J. Kienzle, J. de Lara, G. Mussbacher, E. Syriani, and H. Vangheluwe. "Comparing and classifying model transformation reuse approaches across metamodels". In: *Software and Systems Modeling* 19.2 (2020), pp. 441–465.

[BS81] F. Bancilhon and N. Spyratos. "Update Semantics of Relational Views". In: *ACM Transactions on Database Systems* 6.4 (1981), pp. 557–575.

[BSS14] C. Braga, C. Santos, and V. T. da Silva. "Consistency of model transformation contracts". In: *Science of Computer Programming* 92 (2014), pp. 86–104.

[Buc18] T. Buchmann. "BXtend - A Framework for (Bidirectional) Incremental Model Transformations". In: *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018).* SciTePress, 2018, pp. 336–345.

[Bus+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.* Wiley Publishing, 1996.

[BW84] V. R. Basili and D. M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *IEEE Transactions on Software Engineering* SE-10.6 (1984), pp. 728–738.

[Cab+10] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. “Verification and validation of declarative model-to-model transformations through invariants”. In: *Journal of Systems and Software* 83.2 (2010), pp. 283–302.

[CAV20] P. Carreira, V. Amaral, and H. Vangheluwe. “Multi-Paradigm Modelling for Cyber-Physical Systems: Foundations”. In: *Foundations of Multi-Paradigm Modelling for Cyber-Physical Systems.* Springer International Publishing, 2020, pp. 1–14.

[CCL12] A. Cicchetti, F. Ciccozzi, and T. Leveque. “A hybrid approach for multi-view modeling”. In: *Electronic Communications of the EASST* 50 (2012), pp. 1–12.

[CCP19] A. Cicchetti, F. Ciccozzi, and A. Pierantonio. “Multi-view approaches for software and system modelling: a systematic literature review”. In: *Software and Systems Modeling* 18.6 (2019), pp. 3207–3233.

[CDE06] A. Cicchetti, D. Di Ruscio, and R. Eramo. “Towards Propagation of Changes by Model Approximations”. In: *10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW’06).* IEEE, 2006, p. 24.

[CGL17] J. S. Cuadrado, S. Guerra, and J. de Lara. “Static Analysis of Model Transformations”. In: *IEEE Transactions on Software Engineering* 43.9 (2017), pp. 868–897.

[CH06] K. Czarnecki and S. Helsen. “Feature-based survey of model transformation approaches”. In: *IBM Systems Journal* 45.3 (2006), pp. 621–645.

[Che+17] J. Cheney, J. Gibbons, J. McKinna, and P. Stevens. “On principles of Least Change and Least Surprise for bidirectional transformations”. In: *Journal of Object Technology* 16.1 (2017), 3:1–31.

[Che17] F. Chen. “Änderungsgetriebene Konsistenzhaltung zwischen UML-Klassenmodellen und Java-Code”. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2017.

[Cle+19] A. Cleve, E. Kindler, P. Stevens, and V. Zaytsev. "Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491)". In: *Dagstuhl Reports* 8.12 (2019), pp. 1–48.

[CS13] D. Calegari and N. Szasz. "Verification of Model Transformations: A Survey of the State-of-the-Art". In: *Electronic Notes in Theoretical Computer Science* 292 (2013), pp. 5–25.

[Cua19] J. S. Cuadrado. "A verified catalogue of OCL optimisations". In: *Software and Systems Modeling* 19.5 (2019), pp. 1139–1161.

[dB08] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008).* Vol. 4963. LNCS. Springer Berlin Heidelberg, 2008, pp. 337–340.

[DB82] U. Dayal and P. A. Bernstein. "On the Correct Translation of Update Operations on Relational Views". In: *ACM Transactions on Database Systems* 7.3 (1982), pp. 381–416.

[Dem+15] A. Demuth, M. Riedl-Ehrenleitner, A. Nöhrer, P. Hehenberger, K. Zeman, and A. Egyed. "DesignSpace: An Infrastructure for Multi-user/Multi-tool Engineering". In: *30th Annual ACM Symposium on Applied Computing (SAC 2015).* ACM, 2015, pp. 1486–1491.

[Den+08] T. Denton, E. Jones, S. Srinivasan, K. Owens, and R. W. Buskens. "NAOMI – An Experimental Platform for Multi–modeling". In: *11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008).* Vol. 5301. LNCS. Springer Berlin Heidelberg, 2008, pp. 143–157.

[DEP12] D. Di Ruscio, R. Eramo, and A. Pierantonio. "Model Transformations". In: *12th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Model-Driven Engineering (SFM 2012).* Vol. 7320. LNCS. Springer Berlin Heidelberg, 2012, pp. 91–136.

[DGC17] Z. Diskin, A. Gómez, and J. Cabot. "Traceability Mappings as a Fundamental Instrument in Model Transformations". In: *20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017).* Vol. 10202. LNCS. Springer Berlin Heidelberg, 2017, pp. 247–263.

[Di +17] J. Di Rocco, D. Di Ruscio, M. Heinz, L. Iovino, R. Lämmel, and A. Pierantonio. “Consistency Recovery in Interactive Modeling”. In: *3rd International Workshop on Executable Modeling (EXE) co-Located with 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2017).* Vol. 2019. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 116–122.

[Dis+11] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. “From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case”. In: *14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011).* Vol. 6981. LNCS. Springer Berlin Heidelberg, 2011, pp. 304–318.

[Dis+16a] Z. Diskin, R. Eramo, A. Pierantonio, and K. Czarnecki. “Incorporating Uncertainty into Bidirectional Model Transformations and their Delta-Lens Formalization”. In: *5th International Workshop on Bidirectional Transformations (BX 2016).* Vol. 1571. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 15–31.

[Dis+16b] Z. Diskin, H. Gholizadeh, A. Wider, and K. Czarnecki. “A three-dimensional taxonomy for bidirectional model synchronization”. In: *Journal of Systems and Software* 111 (2016), pp. 298–322.

[DKL18] Z. Diskin, H. König, and M. Lawford. “Multiple Model Synchronization with Multiary Delta Lenses”. In: *21st International Conference on Fundamental Approaches to Software Engineering (FASE 2018).* Vol. 10802. LNCS. Springer International Publishing, 2018, pp. 21–37.

[DKM13] Z. Diskin, S. Kokaly, and T. Maibaum. “Mapping-Aware Megamodeling: Design Patterns and Laws”. In: *6th International Conference on Software Language Engineering (SLE 2013).* Vol. 8225. LNCS. Springer International Publishing, 2013, pp. 322–343.

[DMW05] C. R. Dantas, L. G. P. Murta, and C. M. L. Werner. “Consistent evolution of UML models by automatic detection of change traces”. In: *8th International Workshop on Principles of Software Evolution (IWPSE 2005).* IEEE, 2005, pp. 144–147.

[DXC10] Z. Diskin, Y. Xiong, and K. Czarnecki. “Specifying Overlaps of Heterogeneous Models for Global Consistency Checking”. In: *1st International Workshop on Model-Driven Interoperability (MDI 2010).* ACM, 2010, pp. 42–51.

[EcMMT] Eclipse Foundation. *Model to Model Transformation (MMT).* URL: `https://www.eclipse.org/mmt/` (visited on 01/15/2021).

[EcOCL] Eclipse Foundation. *Eclipse OCL (Object Constraint Language).* URL: `https://projects.eclipse.org/projects/modeling.mdt.ocl` (visited on 01/15/2021).

[EcQVT] Eclipse Foundation. *Eclipse QVTd (QVT Declarative).* URL: `https://projects.eclipse.org/projects/modeling.mmt.qvtd` (visited on 01/15/2021).

[Egy+18] A. Egyed, K. Zeman, P. Hehenberger, and A. Demuth. “Maintaining Consistency across Engineering Artifacts”. In: *IEEE Computer* 51.2 (2018), pp. 28–35.

[Egy06] A. Egyed. “Instant Consistency Checking for the UML”. In: *28th International Conference on Software Engineering (ICSE 2006).* ACM, 2006, pp. 381–390.

[Egy11] A. Egyed. “Automatically Detecting and Tracking Inconsistencies in Software Design Models”. In: *IEEE Transactions on Software Engineering* 37.2 (2011), pp. 188–204.

[Ehr+05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. “Termination Criteria for Model Transformation”. In: *8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005).* Vol. 3442. LNCS. Springer Berlin Heidelberg, 2005, pp. 49–63.

[End+11] J. Endrullis, H. Geuvers, J. G. Simonsen, and H. Zantema. “Levels of undecidability in rewriting”. In: *Information and Computation* 209.2 (2011), pp. 227–245.

[EPR15] R. Eramo, A. Pierantonio, and G. Rosa. “Managing Uncertainty in Bidirectional Model Transformations”. In: *2015 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2015).* ACM, 2015, pp. 49–58.

[Era+08] R. Eramo, A. Pierantonio, J. R. Romero, and A. Vallecillo. "Change Management in Multi-Viewpoint System Using ASP". In: *12th Enterprise Distributed Object Computing Conference Workshops (EDOCW 2008)*. IEEE, 2008, pp. 433–440.

[Era+12] R. Eramo, I. Malavolta, H. Muccini, P. Pelliccione, and A. Pierantonio. "A model-driven approach to automate the propagation of changes among Architecture Description Languages". In: *Software and Systems Modeling* 11 (1 2012), pp. 29–53.

[ESG16] H. Ergin, E. Syriani, and J. Gray. "Design pattern oriented development of model transformations". In: *Computer Languages, Systems & Structures* 46 (2016), pp. 106–139.

[Est08] J. A. Estefan. *Survey of Model-Based Systems Engineering (MBSE) Methodologies*. Tech. rep. INCOSE Technical Publication. International Council on Systems Engineering, 2008, pp. 1–70.

[Eti+10] A. Etien, A. Muller, T. Legrand, and X. Blanc. "Combining Independent Model Transformations". In: *2010 ACM Symposium on Applied Computing (SAC 2010)*. ACM, 2010, pp. 2237–2243.

[Eti+12] A. Etien, V. Aranega, X. Blanc, and R. F. Paige. "Chaining Model Transformations". In: *1st Workshop on the Analysis of Model Transformations (AMT 2012)*. ACM, 2012, pp. 9–14.

[Fal+14] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Montperrus. "Fine-Grained and Accurate Source Code Differencing". In: *29h ACM/IEEE International Conference on Automated Software Engineering (ASE 2014)*. ACM, 2014, pp. 313–324.

[Fle+17] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi. "Model Transformation Modularization as a Many-Objective Optimization Problem". In: *IEEE Transactions on Software Engineering* 43.11 (2017), pp. 1009–1032.

[FM08] S. Fraser and D. Mancl. "No Silver Bullet: Software Engineering Reloaded". In: *IEEE Software* 25.1 (2008), pp. 91–94.

[Fos+05] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. "Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem". In: *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*. ACM, 2005, pp. 233–246.

[Fos+07] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. “Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem”. In: *ACM Transactions on Programming Languages and Systems* 29.3 (2007), 17–es.

[Fow10] M. Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010.

[Fra+18] M. Franzago, D. Di Ruscio, I. Malavolta, and H. Muccini. “Collaborative Model-Driven Software Engineering: A Classification Framework and a Research Map”. In: *IEEE Transactions on Software Engineering* 44.12 (2018), pp. 1146–1175.

[Gam+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing, 1995.

[GFS05] T. Gyimothy, R. Ferenc, and I. Siket. “Empirical validation of object-oriented metrics on open source software for fault prediction”. In: *IEEE Transactions on Software Engineering* 31.10 (2005), pp. 897–910.

[GHN10] H. Giese, S. Hildebrandt, and S. Neumann. “Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent”. In: *Graph Transformations and Model-Driven Engineering: Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*. Vol. 5765. LNCS. Springer Berlin Heidelberg, 2010, pp. 555–579.

[Gis16] B. Gischel. *EPLAN Electric P8 Reference Handbook*. 4th. Hanser Fachbuch, 2016.

[GitApp] Vitruv Tools. *Vitruvius Component-based Systems Case Study (GitHub)*. Release 0.2.0, also available in the reproduction package [Kla21]. URL: `https://github.com/vitruv-tools/Vitruv-Applications-ComponentBasedSystems` (visited on 02/15/2021).

[GitDec] A. Pepin. *Decomposition of Relations for Multi-model Consistency Preservation (GitHub)*. Also available in the reproduction package [Kla21]. URL: `https://github.com/aurelienpepin/KIT_ConsistencyPreservation_Decomposition` (visited on 02/15/2021).

[GitSim] J. Gleitze. *Transformation Network Simulator (GitHub)*. Also available in the reproduction package [Kla21]. URL: `https://github.com/jGleitz/transformationnetwork-simulator` (visited on 02/15/2021).

[GitVit] Vitruv Tools. *Vitruvius Framework (GitHub)*. Release 2.0.1, also available in the reproduction package [Kla21]. URL: `https://github.com/vitruv-tools/Vitruv` (visited on 02/15/2021).

[GKB21] J. Gleitze, H. Klare, and E. Burger. “Finding a Universal Execution Strategy for Model Transformation Networks”. In: *24th International Conference on Fundamental Approaches to Software Engineering (FASE 2021)*. Nominated for EASST best paper award. Springer International Publishing, 2021, pp. 87–107.

[Gle17] J. Gleitze. “A Declarative Language for Preserving Consistency of Multiple Models”. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2017.

[Gol11] T. Goldschmidt. “View-based textual modelling”. PhD thesis. Karlsruhe Institute of Technology (KIT), 2011.

[Gon+12] C. A. González, F. Büttner, R. Clarisó, and J. Cabot. “EMFtoCSP: A tool for the lightweight verification of EMF models”. In: *1st International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA 2012)*. IEEE, 2012, pp. 44–50.

[Gos+18] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith. *The Java Language Specification – Java SE 11 Edition*. Tech. rep. Oracle America, Inc., 2018.

[Gui+18] H. Guissouma, H. Klare, E. Sax, and E. Burger. “An Empirical Study on the Current and Future Challenges of Automotive Software Release and Configuration Management”. In: *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2018)*. IEEE, 2018, pp. 298–305.

[GW09] H. Giese and R. Wagner. “From model transformation to incremental bidirectional model synchronization”. In: *Software and Systems Modeling* 8.1 (2009), pp. 21–43.

[HB13] C. Heinzemann and S. Becker. "Executing Reconfigurations in Hierarchical Component Architectures". In: *16th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE 2013)*. ACM, 2013, pp. 3–12.

[HB19] G. Hinkel and E. Burger. "Change propagation and bidirectionality in internal transformation DSLs". In: *Software and Systems Modeling* 18.1 (2019), pp. 249–278.

[Hei+09a] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. "Closing the Gap between Modelling and Java". In: *2nd International Conference on Software Language Engineering (SLE 2009)*. Vol. 5969. LNCS. Springer Berlin Heidelberg, 2009, pp. 374–383.

[Hei+09b] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. *JaMoPP: The Java Model Parser and Printer*. Tech. rep. 10. Technische Universität Dresden, 2009.

[Hen20] L. Hennig. "Describing Consistency Relations of Multiple Models with Commonalities". Master's Thesis. Karlsruhe Institute of Technology (KIT), 2020.

[Her+12] F. Hermann, H. Ehrig, C. Ermel, and F. Orejas. "Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars". In: *15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*. Vol. 7212. LNCS. Springer Berlin Heidelberg, 2012, pp. 178–193.

[Hid+16] S. Hidaka, M. Tisi, J. Cabot, and Z. Hu. "Feature-based classification of bidirectional transformation approaches". In: *Software and Systems Modeling* 15.3 (2016), pp. 907–928.

[HKA10] F. Heidenreich, J. Kopcsek, and U. Aßmann. "Safe Composition of Transformations". In: *3rd International Conference on Theory and Practice of Model Transformations (ICMT 2010)*. Vol. 6142. LNCS. Springer Berlin Heidelberg, 2010, pp. 108–122.

[HLR08] T. Hettel, M. Lawley, and K. Raymond. "Model Synchronisation: Definitions for Round-Trip Engineering". In: *1st International Conference on Theory and Practice of Model Transformations (ICMT 2008)*. Vol. 5063. LNCS. Springer Berlin Heidelberg, 2008, pp. 31–45.

[HR04] D. Harel and B. Rumpe. “Meaningful modeling: what’s the semantics of "semantics"?” In: *IEEE Computer* 37.10 (2004), pp. 64–72.

[I25010] International Organization for Standardization. *ISO/IEC 25010:-2011 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. 2011.

[I25051] International Organization for Standardization. *ISO/IEC 25051:-2014 – Software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Requirements for quality of Ready to Use Software Product (RUSP) and instructions for testing*. 2014.

[I42010] International Organization for Standardization. *ISO/IEC/IEEE 42010:2011(E) – Systems and software engineering – Architecture description*. 2011.

[I62714] International Electrotechnical Commission. *IEC 62714-1:2018 – Engineering data exchange format for use in industrial automation systems engineering - Automation Markup Language*. 2018.

[ISH08] M.-E. Iacob, M. W. A. Steen, and L. Heerink. “Reusable Model Transformation Patterns”. In: *12th Enterprise Distributed Object Computing Conference Workshops (EDOCW 2008)*. IEEE, 2008, pp. 1–10.

[Jak+14] E. Jakumeit, S. Buchwald, D. Wagelaar, L. Dan, Á. Hegedüs, M. Herrmannsdörfer, T. Horn, E. Kalnina, C. Krause, K. Lano, M. Lepper, A. Rensink, L. Rose, S. Wätzoldt, and S. Mazanek. “A survey and comparison of transformation tools based on the transformation tool contest”. In: *Science of Computer Programming* 85 (2014), pp. 41–99.

[Jou+06] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. “ATL: A QVT-like Transformation Language”. In: *21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications Companion (OOPSLA 2006)*. ACM, 2006, pp. 719–720.

[Kah+19] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró. “Survey and classification of model transformation tools”. In: *Software and Systems Modeling* 18.4 (2019), pp. 2361–2397.

[KD17] H. König and Z. Diskin. "Efficient Consistency Checking of Interrelated Models". In: *13th European Conference on Modelling Foundations and Applications (ECMFA 2017).* Vol. 10376. LNCS. Springer International Publishing, 2017, pp. 161–178.

[KG19] H. Klare and J. Gleitze. "Commonalities for Preserving Consistency of Multiple Models". In: *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS 2019).* IEEE, 2019, pp. 371–378.

[KGZ09] J. M. Küster, T. Gschwind, and O. Zimmermann. "Incremental Development of Model Transformation Chains Using Automated Testing". In: *12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009).* Vol. 5795. LNCS. Springer Berlin Heidelberg, 2009, pp. 733–747.

[KHG11] M. Kuhlmann, L. Hamann, and M. Gogolla. "Extensive Validation of OCL Models by Integrating SAT Solving into USE". In: *49th International Conference on Objects, Models, Components, Patterns (TOOLS 2011).* Vol. 6705. LNCS. Springer Berlin Heidelberg, 2011, pp. 290–306.

[KKE18] R. Kretschmer, D. E. Khelladi, and A. Egyed. "An Automated and Instant Discovery of Concrete Repairs for Model Inconsistencies". In: *40th International Conference on Software Engineering: Companion Proceeedings (ICSE 2018).* ACM, 2018, pp. 298–299.

[KKE19] D. E. Khelladi, R. Kretschmer, and A. Egyed. "Detecting and Exploring Side Effects When Repairing Model Inconsistencies". In: *12th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2019).* ACM, 2019, pp. 113–126.

[Kla+17] H. Klare, E. Burger, M. E. Kramer, M. Langhammer, T. Saglam, and R. Reussner. "Ecoreification: Making Arbitrary Java Code Accessible to Metamodel-Based Tools". In: *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2017).* IEEE, 2017.

[Kla+19a] H. Klare, T. Saglam, E. Burger, and R. Reussner. "Applying Metamodel-based Tooling to Object-oriented Code". In: *7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019).* Vol. 1. INSTICC. SCiTePress, 2019, pp. 219–230.

[Kla+19b] H. Klare, T. Syma, E. Burger, and R. Reussner. "A Categorization of Interoperability Issues in Networks of Transformations". In: *Journal of Object Technology* 18.3 (2019). The 12th International Conference on Model Transformations (ICMT 2019), 4:1–20.

[Kla+20] H. Klare, A. Pepin, E. Burger, and R. Reussner. *A Formal Approach to Prove Compatibility in Transformation Networks*. Tech. rep. 3. Karlsruhe Reports in Informatics. Karlsruhe Institute of Technology (KIT), 2020.

[Kla+21] H. Klare, M. E. Kramer, M. Langhammer, D. Werle, E. Burger, and R. Reussner. "Enabling consistency in view-based system development – The Vitruvius approach". In: *Journal of Systems and Software* 171 (2021).

[Kla16] H. Klare. "Designing a Change-Driven Language for Model Consistency Repair Routines". Master's Thesis. Karlsruhe Institute of Technology (KIT), 2016.

[Kla18] H. Klare. "Multi-Model Consistency Preservation". In: *21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS 2018)*. ACM, 2018, pp. 156–161.

[Kla21] H. Klare. *Reproduction Package for the Dissertation on Building Transformation Networks for Consistent Evolution of Interrelated Models*. 2021. URL: `https://dx.doi.org/10.5445/IR/1000129603`.

[Kol+14] D. Kolovos, L. Rose, A. Garcia-Dominguez, and R. Paige. *The Epsilon Book*. Eclipse Foundation, Inc., 2014.

[KPP06] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. "Model Comparison: A Foundation for Model Composition and Model Transformation Testing". In: *2006 International Workshop on Global Integrated Model Management (GaMMa 2006)*. ACM, 2006, pp. 13–20.

[KPP08] D. Kolovos, R. Paige, and F. Polack. "Detecting and Repairing Inconsistencies across Heterogeneous Models". In: *1st International Conference on Software Testing, Verification, and Validation (ICST 2008)*. IEEE, 2008, pp. 356–364.

[Kra+16] M. E. Kramer, G. Hinkel, H. Klare, M. Langhammer, and E. Burger. “A Controlled Experiment Template for Evaluating the Understandability of Model Transformation Languages”. In: *2nd International Workshop on Human Factors in Modeling co-located with 19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2016).* Vol. 1805. CEUR Workshop Proceedings. CEUR-WS.org, 2016, pp. 11–18.

[Kra17] M. E. Kramer. “Specification Languages for Preserving Consistency between Models of Different Languages”. PhD thesis. Karlsruhe Institute of Technology (KIT), 2017.

[Kre+20] R. Kretschmer, D. E. Khelladi, R. E. Lopez-Herrejon, and A. Egyed. “Consistent change propagation within models”. In: *Software and Systems Modeling* (2020).

[KS06] A. Königs and A. Schürr. “MDI: A Rule-based Multi-document and Tool Integration Approach”. In: *Software and Systems Modeling* 5.4 (2006), pp. 349–368.

[Kus+13] A. Kusel, J. Etzlstorfer, E. Kapsammer, P. Langer, W. Retschitzegger, J. Schonbock, W. Schwinger, and M. Wimmer. “A Survey on Incremental Model Transformation Approaches”. In: *Workshop on Models and Evolution (ME) co-located with 16th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2013).* Vol. 1090. CEUR Workshop Proceedings. CEUR-WS.org, 2013, pp. 4–13.

[KWB03] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing, 2003.

[Läm04] R. Lämmel. “Coupled Software Transformations”. In: *1st International Workshop on Software Evolution Transformations (SET 2004).* 2004, pp. 31–35.

[Lan+14] K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, J. Terrell, and S. Zschaler. “Correct-by-construction synthesis of model transformations using transformation patterns”. In: *Software and Systems Modeling* 13.2 (2014), pp. 873–907.

[Lan+18a] K. Lano, S. Kolahdouz-Rahimi, M. Sharbaf, and H. Alfraihi. "Technical Debt in Model Transformation Specifications". In: *11th International Conference on Theory and Practice of Model Transformation (ICMT 2018)*. Vol. 10888. LNCS. Springer International Publishing, 2018, pp. 127–141.

[Lan+18b] K. Lano, S. Kolahdouz-Rahimi, S. Yassipour-Tehrani, and M. Sharbaf. "A survey of model transformation design patterns in practice". In: *Journal of Systems and Software* 140 (2018), pp. 48–73.

[Lan17] M. Langhammer. "Automated Coevolution of Source Code and Software Architecture Models". PhD thesis. Karlsruhe Institute of Technology (KIT), 2017.

[Lap13] P. A. Laplante. *Requirements Engineering for Software and Systems*. 2nd. Auerbach Publications, 2013.

[Lar+18] J. de Lara, E. Guerra, M. Chechik, and R. Salay. "Model Transformation Product Lines". In: *21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*. ACM, 2018, pp. 67–77.

[Lar+19] J. D. Lara, E. Guerra, D. D. Ruscio, J. D. Rocco, J. S. Cuadrado, L. Iovino, and A. Pierantonio. "Automated Reuse of Model Transformations Through Typing Requirements Models". In: *ACM Transactions on Software Engineering and Methodology* 28.4 (2019), 21:1–21:62.

[Leb+14] E. Leblebici, A. Anjorin, A. Schürr, S. Hildebrandt, J. Rieke, and J. Greenyer. "A Comparison of Incremental Triple Graph Grammar Tools". In: *Electronic Communications of the EASST* 67 (2014), pp. 1–14.

[LK15] M. Langhammer and K. Krogmann. "A Co-evolution Approach for Source Code and Component-based Architecture Models". In: *17. Workshop Software-Reengineering und -Evolution (WSRE 2015)*. Vol. 35. Softwaretechnik-Trends 2. GI, 2015.

[Lúc+13] L. Lúcio, S. Mustafiz, J. Denil, H. Vangheluwe, and M. Jukss. "FTG+PM: An Integrated Framework for Investigating Model Transformation Chains". In: *16th International SDL Forum: Model-Driven Dependability Engineering (DSL 2013)*. Vol. 7916. LNCS. Springer Berlin Heidelberg, 2013, pp. 182–202.

[Mal+10] I. Malavolta, H. Muccini, P. Pelliccione, and D. A. Tamburri. "Providing Architectural Languages and Tools Interoperability through Model Transformation Technologies". In: *IEEE Transactions of Software Engineering* 36.1 (2010), pp. 119–140.

[Maz+17] M. Mazkatli, E. Burger, A. Koziolek, and R. H. Reussner. "Automotive Systems Modelling with Vitruvius". In: *INFORMATIK 2017*. Vol. P-275. Lecture Notes in Informatics (LNI). GI, 2017, pp. 1487–1498.

[MBF11] S. Murer, B. Bonati, and F. J. Furrer. *Managed Evolution – A Strategy for Very Large Information Systems*. Springer Berlin Heidelberg, 2011.

[MC13] N. Macedo and A. Cunha. "Implementing QVT-R Bidirectional Model Transformations Using Alloy". In: *16th International Conference on Fundamental Approaches to Software Engineering (FASE 2013)*. Vol. 7793. LNCS. Springer Berlin Heidelberg, 2013, pp. 297–311.

[MC16] N. Macedo and A. Cunha. "Least-change bidirectional model transformation with QVT-R and ATL". In: *Software and Systems Modeling* 15.3 (2016), pp. 783–810.

[MCP14] N. Macedo, A. Cunha, and H. Pacheco. "Towards a Framework for Multidirectional Model Transformations". In: *3rd International Workshop on Bidirectional Transformations (BX 2014)*. Vol. 1133. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 71–74.

[MDA] Object Management Group (OMG). *Model Driven Architecture (MDA) – MDA Guide rev. 2.0*. 2014. URL: `https://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf`.

[Mei+19] J. Meier, H. Klare, C. Tunjic, C. Atkinson, E. Burger, R. Reussner, and A. Winter. "Single Underlying Models for Projectional, Multi-View Environments". In: *7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*. Vol. 1. INSTICC. SCiTePress, 2019, pp. 119–130.

[Mei+20] J. Meier, C. Werner, H. Klare, C. Tunjic, U. Aßmann, C. Atkinson, E. Burger, R. Reussner, and A. Winter. "Classifying Approaches for Constructing Single Underlying Models". In: *7th International Conference on Model-Driven Engineering and Software*

Development (MODELSWARD 2019). Vol. 1161. Communications in Computer and Information Science. Springer International Publishing, 2020, pp. 350–375.

[MGC13] N. Macedo, T. Guimaraes, and A. Cunha. "Model repair and transformation with Echo". In: *28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*. IEEE, 2013, pp. 694–697.

[MJC17] N. Macedo, T. Jorge, and A. Cunha. "A Feature-based Classification of Model Repair Approaches". In: *IEEE Transactions on Software Engineering* 43.7 (2017), pp. 615–640.

[MOF] Object Management Group (OMG). *OMG Meta Object Facility (MOF) Core Specification*. Version 2.5.1. 2016. URL: `http://www.omg.org/spec/MOF/2.5.1/`.

[MTD17] S. Martínez, M. Tisi, and R. Douence. "Reactive model transformation with ATL". In: *Science of Computer Programming* 136 (2017), pp. 1–16.

[MW18] J. Meier and A. Winter. "Model Consistency ensured by Metamodel Integration". In: *6th International Workshop on The Globalization of Modeling Languages (GEMOC) co-located with 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2018)*. Vol. 2245. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 408–415.

[OAPI] The Linux Foundation. *OpenAPI Initiative*. URL: `https://www.openapis.org/` (visited on 01/15/2021).

[OCL] Object Management Group (OMG). *Object Constraint Language*. Version 2.4. 2014. URL: `http://www.omg.org/spec/OCL/2.4/`.

[Ohr+18] M. Ohrndorf, C. Pietsch, U. Kelter, and T. Kehrer. "ReVision: A Tool for History-based Model Repair Recommendations". In: *40th International Conference on Software Engineering: Companion Proceeedings (ICSE 2018)*. ACM, 2018, pp. 105–108.

[Old05] J. Oldevik. "Transformation Composition Modelling Framework". In: *5th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2005)*. Vol. 3543. LNCS. Springer Berlin Heidelberg, 2005, pp. 108–114.

[OPN20] F. Orejas, E. Pino, and M. Navarro. “Incremental Concurrent Model Synchronization using Triple Graph Grammars”. In: *23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020)*. Vol. 12076. LNCS. Springer International Publishing, 2020, pp. 273–293.

[Pat69] K. Paton. “An Algorithm for Finding a Fundamental Set of Cycles of a Graph”. In: *Communications of the ACM* 12.9 (1969), pp. 514–518.

[Pep19] A. Pepin. “Decomposition of Relations for Multi-model Consistency Preservation”. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2019.

[Pep79] P. Pepper. “A study on transformational semantics”. In: *International Summer School on Program Construction*. Vol. 69. LNCS. Springer Berlin Heidelberg, 1979, pp. 322–405.

[Per+13] M. Persson, M. Törngren, A. Qamar, J. Westman, M. Biehl, S. Tripakis, H. Vangheluwe, and J. Denil. “A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems”. In: *11th International Conference on Embedded Software (EMSOFT 2013)*. IEEE, 2013, 10:1–10:10.

[Pil+08] J. von Pilgrim, B. Vanhooff, I. Schulz-Gerlach, and Y. Berbers. “Constructing and Visualizing Transformation Chains”. In: *4th European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA 2008)*. Vol. 5095. LNCS. Springer Berlin Heidelberg, 2008, pp. 17–32.

[PMR16] R. F. Paige, N. Matragkas, and L. 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.

[PRV08] M. Petrenko, V. Rajlich, and R. Vanciu. “Partial Domain Comprehension in Software Evolution and Maintenance”. In: *16th IEEE International Conference on Program Comprehension (ICPC 2008)*. IEEE, 2008, pp. 13–22.

[PSM02] Ping Yu, T. Systa, and H. Muller. “Predicting fault-proneness using OO metrics. An industrial case study”. In: *6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*. IEEE, 2002, pp. 99–107.

[QVT] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Version 1.3. 2016. URL: `http://www.omg.org/spec/QVT/1.3/`.

[RB05] J. R. Rumbaugh and M. R. Blaha. *Object-Oriented Modeling and Design with UML*. Pearson Education, 2005.

[RE12a] A. Reder and A. Egyed. "Computing Repair Trees for Resolving Inconsistencies in Design Models". In: *27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)* (2012), pp. 220–229.

[RE12b] A. Reder and A. Egyed. "Incremental Consistency Checking for Complex Design Rules and Larger Model Changes". In: *15th International Conference on Model Driven Engineering Languages and Systems (MODELS 2012)*. Vol. 7590. LNCS. Springer Berlin Heidelberg, 2012, pp. 202–218.

[Reu+16] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016.

[RST19] J. Reineke, C. Stergiou, and S. Tripakis. "Basic problems in multi-view modeling". In: *Software and Systems Modeling* 18.3 (2019), pp. 1577–1611.

[Sağ20] T. Sağlam. "A Case Study for Networks of Bidirectional Transformations". Master's Thesis. Karlsruhe Institute of Technology (KIT), 2020.

[Sal+15] R. Salay, S. Kokaly, A. Di Sandro, and M. Chechik. "Enriching megamodel management with collection-based operators". In: *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*. IEEE, 2015, pp. 236–245.

[Sal+20] R. Salay, S. Kokaly, A. Di Sandro, N. L. Fung, and M. Chechik. "Heterogeneous megamodel management using collection operators". In: *Software and Systems Modeling* 19.1 (2020), pp. 231–260.

[Sax+17] E. Sax, R. Reussner, H. Guissouma, and H. Klare. *A Survey on the State and Future of Automotive Software Release and Configuration Management*. Tech. rep. 11. Karlsruhe Reports in Informatics. Karlsruhe Institute of Technology (KIT), 2017.

[SC13] C. Szabo and Y. Chen. “A Model-Driven Approach for Ensuring Change Traceability and Multi-Model Consistency”. In: *22nd Australasian Software Engineering Conference (ASWEC 2013)*. IEEE, 2013, pp. 127–136.

[Sch15] O. Scheid. *AUTOSAR Compendium - Part 1: Application and RTE*. AUTOSAR - Compendium Series. CreateSpace Independent Publishing Platform, 2015.

[Sch95] A. Schürr. “Specification of graph translators with triple graph grammars”. In: *20th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1994)*. Vol. 903. LNCS. Springer Berlin Heidelberg, 1995, pp. 151–163.

[SDP20] A. Sahay, D. Di Ruscio, and A. Pierantonio. “Understanding the Role of Model Transformation Compositions in Low-Code Development Platforms”. In: *23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings (MODELS 2020)*. ACM, 2020, pp. 1–5.

[Sem+16] O. Semeráth, C. Debreceni, Á. Horváth, and D. Varró. “Incremental Backward Change Propagation of View Models by Logic Solvers”. In: *19th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. ACM, 2016, pp. 306–316.

[SG08] J. Sánchez Cuadrado and J. García Molina. “Approaches for Model Transformation Reuse: Factorization and Composition”. In: *1st International Conference on Theory and Practice of Model Transformations (ICMT 2008)*. Vol. 5063. LNCS. Springer Berlin Heidelberg, 2008, pp. 168–182.

[Simu] MathWorks. *Simulink – Simulation and Model-Based Design – MATLAB & Simulink*. URL: `https://www.mathworks.com/products/simulink.html` (visited on 01/15/2021).

[SK03] S. Sendall and W. Kozaczynski. “Model transformation: the heart and soul of model-driven software development”. In: *IEEE Software* 20.5 (2003), pp. 42–45.

[SK16] M. Strittmatter and A. Kechaou. *The Media Store 3 Case Study System*. Tech. rep. 1. Karlsruhe Reports in Informatics. Karlsruhe Institute of Technology (KIT), 2016.

[SME08] R. Salay, J. Mylopoulos, and S. Easterbrook. "Managing Models through Macromodeling". In: *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*. IEEE, 2008, pp. 447–450.

[SME09] R. Salay, J. Mylopoulos, and S. Easterbrook. "Using Macromodels to Manage Collections of Related Models". In: *21st International Conference on Advanced Information Systems Engineering (CAiSE 2009)*. Vol. 5565. LNCS. Springer Berlin Heidelberg, 2009, pp. 141–155.

[Son+12] H. S. Son, W. Y. Kim, R. Y. Kim, and H.-G. Min. "Metamodel Design for Model Transformation from Simulink to ECML in Cyber Physical Systems". In: *International Conferences on Computer Applications for Graphics, Grid Computing, and Industrial Environment*. Vol. 351. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012, pp. 56–60.

[Sta+06] T. Stahl, M. Völter, J. Bettin, A. Haase, and S. Helsen. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Ltd., 2006.

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973.

[Ste+09] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. 2nd ed. Addison-Wesley Professional, 2009.

[Ste08a] P. Stevens. "A Landscape of Bidirectional Model Transformations". In: *International Summer School on Generative and Transformational Techniques in Software Engineering II (GTTSE 2007)*. Vol. 5235. LNCS. Springer Berlin Heidelberg, 2008, pp. 408–424.

[Ste08b] P. Stevens. "Towards an Algebraic Theory of Bidirectional Transformations". In: *4th International Conference Graph Transformations (ICGT 2008)*. Vol. 5214. LNCS. Springer Berlin Heidelberg, 2008, pp. 1–17.

[Ste10] P. Stevens. "Bidirectional model transformations in QVT: semantic issues and open questions". In: *Software and Systems Modeling* 9.1 (2010), pp. 7–20.

[Ste12] P. Stevens. "Observations relating to the equivalences induced on model sets by bidirectional transformations". In: *Electronic Communications of the EASST* 49 (2012), pp. 1–16.

[Ste14] P. Stevens. "Bidirectionally Tolerating Inconsistency: Partial Transformations". In: *17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014)*. Vol. 8411. LNCS. Springer Berlin Heidelberg, 2014, pp. 32–46.

[Ste18] P. Stevens. "Is Bidirectionality Important?" In: *14th European Conference on Modelling Foundations and Applications (ECMFA 2018)*. Vol. 10890. LNCS. Springer International Publishing, 2018, pp. 1–11.

[Ste20a] P. Stevens. "Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels". In: *Software and Systems Modeling* 19.4 (2020), pp. 935–958.

[Ste20b] P. Stevens. "Maintaining consistency in networks of models: bidirectional transformations in the large". In: *Software and Systems Modeling* 19.1 (2020), pp. 39–65.

[Stü+18] P. Stünkel, H. König, Y. Lamo, and A. Rutle. "Multimodel Correspondence Through Inter-model Constraints". In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming (Programming 2018)*. ACM, 2018, pp. 9–17.

[Stü+20] P. Stünkel, H. König, Y. Lamo, and A. Rutle. "Towards Multiple Model Synchronization with Comprehensive Systems". In: *23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020)*. Vol. 12076. LNCS. Springer International Publishing, 2020, pp. 335–356.

[SWS12] R. Salay, S. Wang, and V. Suen. "Managing Related Models in Vehicle Control Software Development". In: *15th International Conference on Model Driven Engineering Languages and Systems (MODELS 2012)*. Vol. 7590. LNCS. Springer Berlin Heidelberg, 2012, pp. 383–398.

[Sym18] T. Syma. “Multi-model Consistency through Transitive Combination of Binary Transformations”. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2018.

[SysML] Object Management Group (OMG). *OMG System Modeling Language (OMG SysML)*. Version 1.6. 2019. URL: `https://www.omg.org/spec/SysML/1.6/`.

[SZK15] L. Samimi-Dehkordi, B. Zamani, and S. Kolahdouz-Rahimi. “From trace-based inter-model validation to bidirectional model synchronization with reconciliation”. In: *5th International Conference on Computer and Knowledge Engineering (ICCKE 2015)*. IEEE, 2015, pp. 123–130.

[SZK16] L. Samimi-Dehkordi, B. Zamani, and S. Kolahdouz-Rahimi. “Bidirectional model transformation approaches – a comparative study”. In: *6th International Conference on Computer and Knowledge Engineering (ICCKE 2016)*. IEEE, 2016, pp. 314–320.

[SZK18] L. Samimi-Dehkordi, B. Zamani, and S. Kolahdouz-Rahimi. “EVL+Strace: a novel bidirectional model transformation approach”. In: *Information and Software Technology* 100 (2018), pp. 47–72.

[TA15] F. Trollmann and S. Albayrak. “Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models”. In: *8th International Conference on Theory and Practice of Model Transformations (ICMT 2015)*. Vol. 9152. LNCS. Springer International Publishing, 2015, pp. 214–229.

[TA16] F. Trollmann and S. Albayrak. “Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models”. In: *9th International Conference on Theory and Practice of Model Transformations (ICMT 2016)*. Vol. 9765. LNCS. Springer International Publishing, 2016, pp. 91–106.

[Tar+99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. “N degrees of separation: multi-dimensional separation of concerns”. In: *1999 International Conference on Software Engineering (ICSE)*. IEEE, 1999, pp. 107–119.

[Tar71] R. Tarjan. “Depth-first search and linear graph algorithms”. In: *12th Annual Symposium on Switching and Automata Theory (SWAT 1971)*. IEEE, 1971, pp. 114–121.

[TK19] M. Tichy and H. Klare. "Human Factors: Interests of Transformation Developers and Users". In: *Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491)*. Vol. 8. Dagstuhl Reports 12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, pp. 16–20.

[TME19] M. A. Tröls, A. Mashkoor, and A. Egyed. "Live and Global Consistency Checking in a Collaborative Engineering Environment". In: *34th ACM/SIGAPP Symposium on Applied Computing (SAC 2019)*. ACM, 2019, pp. 1776–1785.

[Tro+15] J. Troya, A. Bergmayr, L. Burgueño, and M. Wimmer. "Towards systematic mutations for and with ATL model transformations". In: *8th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2015)*. IEEE, 2015, pp. 1–10.

[TSR18] J. Troya, S. Segura, and A. Ruiz-Corts. "Automated inference of likely metamorphic relations for model transformations". In: *Journal of Systems and Software* 136 (2018), pp. 188–208.

[TZL16] S. Y. Tehrani, S. Zschaler, and K. Lano. "Requirements Engineering in Model-Transformation Development: An Interview-Based Study". In: *9th International Conference on Theory and Practice of Model Transformations (ICMT 2016)*. Vol. 9765. LNCS. Springer International Publishing, 2016, pp. 123–137.

[UML] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML)*. Version 2.5.1. 2017. URL: `https://www.omg.org/spec/UML/2.5.1/`.

[Val+12] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer, and L. Hamann. "Formal Specification and Testing of Model Transformations". In: *12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*. Vol. 7320. LNCS. Springer Berlin Heidelberg, 2012, pp. 399–437.

[Van+06] B. Vanhooff, S. Van Baelen, A. Hovsepyan, W. Joosen, and Y. Berbers. "Towards a Transformation Chain Modeling Language". In: *6th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS 2006)*. Vol. 4017. LNCS. Springer Berlin Heidelberg, 2006, pp. 39–48.

[Van+07] B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, and Y. Berbers. "UniTI: A Unified Transformation Infrastructure". In: *10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*. Vol. 4735. LNCS. Springer Berlin Heidelberg, 2007, pp. 31–45.

[Var+06] D. Varró, S. Varró–Gyapay, H. Ehrig, U. Prange, and G. Taentzer. "Termination Analysis of Model Transformations by Petri Nets". In: *3rd International Conference on Graph Transformations (ICGT 2006)*. Vol. 4178. LNCS. Springer Berlin Heidelberg, 2006, pp. 260–274.

[Var+16] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi. "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework". In: *Software and Systems Modeling* 15.3 (2016), pp. 609–629.

[Vea+12] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. "Symbolic Finite State Transducers: Algorithms and Applications". In: *ACM SIGPLAN Notices* 47.1 (2012), pp. 137–150.

[VL03] H. Vangheluwe and J. de Lara. "Computer automated multi-paradigm modelling: meta-modelling and graph transformation". In: *2003 Winter Simulation Conference (WSC 2003)*. Vol. 1. IEEE, 2003, pp. 595–603.

[Völ+13] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. C. L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[WA18] C. Werner and U. Assmann. "Model Synchronization with the Role-oriented Single Underlying Model". In: *13th International Workshop on Models@run.time co-located with 21st ACM/IEEE International Conference on Model Driven Engineering Languages (MODELS 2018)*. Vol. 2245. CEUR Workshop Proceedings. CEUR-WS.org, 2018, pp. 62–71.

[WA20] N. Weidmann and A. Anjorin. "Schema Compliant Consistency Management via Triple Graph Grammars and Integer Linear Programming". In: *23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020)*. Vol. 12076. LNCS. Springer International Publishing, 2020, pp. 315–334.

[Wag+11] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault. "Towards a General Composition Semantics for Rule-Based Model Transformation". In: *14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011)*. Vol. 6981. LNCS. Springer Berlin Heidelberg, 2011, pp. 623–637.

[Wag08] D. Wagelaar. "Composition Techniques for Rule-Based Model Transformation Languages". In: *1st International Conference on Theory and Practice of Model Transformations (ICMT 2008)*. Vol. 5063. LNCS. Springer Berlin Heidelberg, 2008, pp. 152–167.

[Wei+19] N. Weidmann, A. Anjorin, E. Leblebici, and A. Schürr. "Consistency Management via a Combination of Triple Graph Grammars and Linear Programming". In: *12th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2019)*. ACM, 2019, pp. 29–41.

[Wer+18] C. Werner, H. Schön, T. Kühn, S. Götz, and U. Aßmann. "Role-Based Runtime Model Synchronization". In: *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2018)*. IEEE, 2018, pp. 306–313.

[Wer16] D. Werle. "A Declarative Language for Bidirectional Model Consistency". Master's Thesis. Karlsruhe Institute of Technology (KIT), 2016.

[WFA20] N. Weidmann, L. Fritsche, and A. Anjorin. "A Search-Based and Fault-Tolerant Approach to Concurrent Model Synchronisation". In: *13th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2020)*. SLE 2020. ACM, 2020, pp. 56–71.

[Woh+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012.

[Wol+15] C. Wolff, L. Krawczyk, R. Höttger, C. Brink, U. Lauschner, D. Fruhner, E. Kamsties, and B. Igel. "AMALTHEA – Tailoring tools to projects in automotive software development". In: *8th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS 2015)*. Vol. 2. IEEE, 2015, pp. 515–520.

[WS20] N. Weidmann and S. Sauer. “Applying Bidirectional Transformations in Industrial Contexts: Challenges and Solutions”. In: *22. Workshop Software-Reengineering und -Evolution (WSRE 2020)*. GI, 2020.

[WVD10] D. Wagelaar, R. Van Der Straeten, and D. Deridder. “Module superimposition: a composition technique for rule-based model transformation languages”. In: *Software and Systems Modeling* 9.3 (2010), pp. 285–309.

[Xio+07] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. “Towards Automatic Model Synchronization from Model Transformations”. In: *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM, 2007, pp. 164–173.

[Xio+09] Y. Xiong, H. Song, Z. Hu, and M. Takeichi. “Supporting Parallel Updates with Bidirectional Model Transformations”. In: *2nd International Conference on Theory and Practice of Model Transformations (ICMT 2009)*. Vol. 5563. LNCS. Springer Berlin Heidelberg, 2009, pp. 213–228.

[Xio+13] Y. Xiong, H. Song, Z. Hu, and M. Takeichi. “Synchronizing concurrent model updates based on bidirectional transformation”. In: *Software and Systems Modeling* 12.1 (2013), pp. 89–104.

[Yie+09] A. Yie, R. Casallas, D. Wagelaar, and D. Deridder. “An Approach for Evolving Transformation Chains”. In: *12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*. Vol. 5795. LNCS. Springer Berlin Heidelberg, 2009, pp. 551–555.

[Yie+12] A. Yie, R. Casallas, D. Deridder, and D. Wagelaar. “Realizing Model Transformation Chain interoperability”. In: *Software and Systems Modeling* 11.1 (2012), pp. 55–75.

The Karlsruhe Series on Software Design and Quality

ISSN 1867-0067

Band 1 **Steffen Becker**
Coupled Model Transformations for QoS Enabled Component-Based Software Design.
ISBN 978-3-86644-271-9

Band 2 **Heiko Koziolek**
Parameter Dependencies for Reusable Performance Specifications of Software Components.
ISBN 978-3-86644-272-6

Band 3 **Jens Happe**
Predicting Software Performance in Symmetric Multi-core and Multiprocessor Environments.
ISBN 978-3-86644-381-5

Band 4 **Klaus Krogmann**
Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis.
ISBN 978-3-86644-804-9

Band 5 **Michael Kuperberg**
Quantifying and Predicting the Influence of Execution Platform on Software Component Performance.
ISBN 978-3-86644-741-7

Band 6 **Thomas Goldschmidt**
View-Based Textual Modelling.
ISBN 978-3-86644-642-7

Band 7 **Anne Koziolek**
Automated Improvement of Software Architecture Models for Performance and Other Quality Attributes.
ISBN 978-3-86644-973-2

Band 8 **Lucia Happe**
Configurable Software Performance Completions through Higher-Order Model Transformations.
ISBN 978-3-86644-990-9

Band 9 **Franz Brosch**
Integrated Software Architecture-Based Reliability Prediction for IT Systems.
ISBN 978-3-86644-859-9

Band 10 **Christoph Rathfelder**
Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation.
ISBN 978-3-86644-969-5

Band 11 **Henning Groenda**
Certifying Software Component Performance Specifications.
ISBN 978-3-7315-0080-3

Band 12 **Dennis Westermann**
Deriving Goal-oriented Performance Models by Systematic Experimentation.
ISBN 978-3-7315-0165-7

Band 13 **Michael Hauck**
Automated Experiments for Deriving Performance-relevant Properties of Software Execution Environments.
ISBN 978-3-7315-0138-1

Band 14 **Zoya Durdik**
Architectural Design Decision Documentation through Reuse of Design Patterns.
ISBN 978-3-7315-0292-0

Band 15 **Erik Burger**
Flexible Views for View-based Model-driven Development.
ISBN 978-3-7315-0276-0

Band 16 **Benjamin Klatt**
Consolidation of Customized Product Copies into Software Product Lines.
ISBN 978-3-7315-0368-2

Band 17 **Andreas Rentschler**
Model Transformation Languages with Modular Information Hiding.
ISBN 978-3-7315-0346-0

Band 18 **Omar-Qais Noorshams**
Modeling and Prediction of I/O Performance in Virtualized Environments.
ISBN 978-3-7315-0359-0

Band 19 **Johannes Josef Stammel**
Architekturbasierte Bewertung und Planung von Änderungsanfragen.
ISBN 978-3-7315-0524-2

Band 20 **Alexander Wert**
Performance Problem Diagnostics by Systematic Experimentation.
ISBN 978-3-7315-0677-5

Band 21 **Christoph Heger**
An Approach for Guiding Developers to Performance and Scalability Solutions.
ISBN 978-3-7315-0698-0

Band 22 **Fouad ben Nasr Omri**
Weighted Statistical Testing based on Active Learning and Formal Verification Techniques for Software Reliability Assessment.
ISBN 978-3-7315-0472-6

Band 23 **Michael Langhammer**
Automated Coevolution of Source Code and Software Architecture Models.
ISBN 978-3-7315-0783-3

Band 24 **Max Emanuel Kramer**
Specification Languages for Preserving Consistency between Models of Different Languages.
ISBN 978-3-7315-0784-0

Band 25 **Sebastian Michael Lehrig**
Efficiently Conducting Quality-of-Service Analyses by Templating Architectural Knowledge.
ISBN 978-3-7315-0756-7

Band 26 **Georg Hinkel**
Implicit Incremental Model Analyses and Transformations.
ISBN 978-3-7315-0763-5

Band 27 **Christian Stier**
Adaptation-Aware Architecture Modeling and Analysis of Energy Efficiency for Software Systems.
ISBN 978-3-7315-0851-9

Band 28 **Lukas Märtin**
Entwurfsoptimierung von selbst-adaptiven Wartungsmechanismen für software-intensive technische Systeme.
ISBN 978-3-7315-0852-6

Band 29 **Axel Busch**
Quality-driven Reuse of Model-based Software Architecture Elements.
ISBN 978-3-7315-0951-6

Band 30 **Kiana Busch**
An Architecture-based Approach for Change Impact Analysis of Software-intensive Systems.
ISBN 978-3-7315-0974-5

Band 31 **Misha Strittmatter**
A Reference Structure for Modular Metamodels of Quality-Describing Domain-Specific Modeling Languages.
ISBN 978-3-7315-0982-0

Band 32 **Markus Frank**
Model-Based Performance Prediction for Concurrent Software on Multicore Architectures. A Simulation-Based Approach.
ISBN 978-3-7315-1146-5

Band 33 **Manuel Gotin**
QoS-Based Optimization of Runtime Management of Sensing Cloud Applications.
ISBN 978-3-7315-1147-2

Band 34 **Heiko Klare**
Building Transformation Networks for Consistent Evolution of Interrelated Models.
ISBN 978-3-7315-1132-8

The Karlsruhe Series on
Software Design and Quality

Edited by Prof. Dr. Ralf Reussner

During the development of complex software systems, stakeholders specify their concerns in different artifacts, such as code, design diagrams, or deployment descriptions. Due to redundancies between such artifacts, ensuring that they describe the system in a consistent way is crucial. Transformations are well suited to automate the process of preserving consistency for pairs of artifacts upon changes in one of them.

In this book, we investigate how developers can combine independently developed and reusable transformations to networks such that they preserve consistency between more than two artifacts. From a formal definition of these networks, we derive synchronization, compatibility, and orchestration as central challenges, and we develop approaches to solve them. In addition, we propose a construction approach to mitigate trade-offs between quality properties of networks, such as their maintainability, and a language that supports developers in applying the approach.

ISBN 978-3-7315-1132-8

ISSN 1867-0067
ISBN 978-3-7315-1132-8

Gedruckt auf FSC-zertifiziertem Papier

9 783731 511328 >