

Building Transformation Networks for Consistent Evolution of Multiple Models

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften

der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

vorgelegte
Dissertation

von
Heiko Klare

aus Höxter

Tag der mündlichen Prüfung: XX. Monat XXXX
Erster Gutachter: Prof. Dr. Ralf H. Reussner
Zweiter Gutachter: Prof. Dr. Colin Atkinson

Abstract

Zusammenfassung

Contents Overview

Abstract	i
Zusammenfassung	iii
List of Figures	xiii
List of Tables	xvii
List of Algorithms	xix
List of Definitions	xxi
List of Theorems	xxiii
List of Insights	xxv
Acronyms	xxvii
I. Prologue	1
1. Introduction	3
2. Foundations and Notation	27
3. Models, Consistency and Processes	35
II. Building Correct Transformation Networks	49
4. Correctness in Transformation Networks	51
5. Proving Compatibility of Consistency Relations	97
6. Constructing Synchronizing Transformations	175
7. Orchestrating Transformation Networks	227
8. Classifying Errors in Transformation Networks	293
9. Evaluation and Discussion	317

CONTENTS OVERVIEW

A. Appendix	381
Bibliography	389

Contents

Abstract	i
Zusammenfassung	iii
List of Figures	xiii
List of Tables	xvii
List of Algorithms	xix
List of Definitions	xxi
List of Theorems	xxiii
List of Insights	xxv
Acronyms	xxvii
I. Prologue	1
1. Introduction	3
1.1. Consistency of Multiple Models	3
1.1.1. Consistency in System Engineering	3
1.1.2. Distributed and Reusable Consistency Knowledge	5
1.1.3. Orchestration of Transformation Networks	7
1.2. Consistency Specification Challenges	9
1.2.1. Correctness of Transformation Networks	10
1.2.2. Quality of Transformation Networks	15
1.2.3. Challenges Overview	18
1.3. Research Objective	18
1.3.1. Research Goal and Questions	19

1.3.2. Context and Assumptions	21
1.3.3. Contributions	22
1.3.4. Benefits	24
1.4. Thesis Outline	25
2. Foundations and Notation	27
2.1. Modeling	27
2.1.1. Models and Model Theory	27
2.1.2. Model-Driven Software Engineering	27
2.2. Modeling Formalisms and Frameworks	28
2.2.1. Meta-Object Facility	28
2.2.2. EMF and Ecore	28
2.2.3. Differences	29
2.2.4. Simplification	30
2.3. Multi-View Modeling	30
2.3.1. Orthographic Software Modeling	30
2.3.2. The VITRUVIUS Framework	31
2.4. Model Transformations	31
2.4.1. Bidirectional Transformations	31
2.4.2. Transformation Languages	31
2.5. Mathematical Notations	31
3. Models, Consistency and Processes	35
3.1. Dimensions of Consistency, Specification and Preservation	35
3.1.1. Normative and Descriptive Specification	35
3.1.2. Structural and Behavioral Consistency	36
3.1.3. Checking and Preserving Consistency	38
3.2. Consistency Specification Process	41
3.2.1. Roles	41
3.2.2. Scenarios	42
3.3. Models and Metamodels	43
3.3.1. Notation and Conventions	44
3.3.2. Modeling Elements	45
3.3.3. Assumptions	46
3.4. Running Example	46

II. Building Correct Transformation Networks	49
4. Correctness in Transformation Networks	51
4.1. Notions of Consistency and its Preservation	52
4.1.1. Intensional and Extensional Consistency Notions	52
4.1.2. Monolithic and Modular Consistency Notions	53
4.1.3. Consistency Preservation	54
4.1.4. Declarative and Imperative Specifications	57
4.1.5. Consistency Preservation Artifacts	58
4.2. Notions of Correctness for Consistency Specifications	60
4.2.1. Relative Correctness Notions	60
4.2.2. Correctness regarding Global Knowledge	61
4.2.3. Dimensions of Correctness	62
4.2.4. Correctness of Consistency Relations	63
4.3. A Formal Notion of Transformation Networks	64
4.3.1. Modular Consistency Specification	64
4.3.2. Incremental Consistency Preservation	69
4.3.3. Transformation Orchestration	77
4.3.4. Transformation Networks	82
4.4. A Fine-grained Notion of Consistency	83
4.4.1. Fine-Grained Consistency Relations	84
4.4.2. Expressiveness of Fine-Grained Relations	90
4.4.3. Application to Consistency Preservation Rules	92
4.5. Summary	94
5. Proving Compatibility of Consistency Relations	97
5.1. Towards a Notion of Compatibility	101
5.1.1. Necessity of Obsolete Relation Elements	103
5.1.2. Prevention from Finding Consistent Solutions	105
5.1.3. An Informal Notion of Compatibility	107
5.1.4. An Analysis for Compatibility of Relations	109
5.2. A Formal Notion of Compatibility	111
5.2.1. Implicit Consistency Relations	111
5.2.2. Transitive Closure of Consistency Relations	116
5.2.3. Compatibility of Consistency Relations	118
5.3. A Formal Approach to Prove Compatibility	123
5.3.1. Independence of Consistency Relations	124
5.3.2. Consistency Relation Trees	126
5.3.3. Redundancy of Consistency Relations	131

5.3.4.	Compatibility-Preserving Redundancy	136
5.3.5.	An Algorithm to Prove Compatibility	140
5.4.	A Practical Approach to Prove Compatibility	143
5.4.1.	Consistency Relations in Transformation Languages	144
5.4.2.	Consistency Relations Represented as Graphs	151
5.4.3.	Decomposition of Consistency Relations	157
5.4.4.	Redundancy Check for Consistency Relations	163
5.5.	Summary	172
6.	Constructing Synchronizing Transformations	175
6.1.	Deriving the Gap to Ordinary Transformation	177
6.1.1.	Behavior of Ordinary Transformations in Networks .	177
6.1.2.	Unidirectional Consistency Preservation Rules . . .	179
6.1.3.	Unidirectional Relations and Preservation	181
6.1.4.	Bidirectional Transformations	185
6.2.	Combining Unidirectional Consistency Preservation Rules .	186
6.2.1.	Options for Combination	187
6.2.2.	Sequencing of Consistency Preservation Rules . . .	189
6.2.3.	Execution Bounds	192
6.2.4.	Necessity for Synchronization Extension	194
6.3.	Synchronizing Bidirectional Transformations	195
6.3.1.	Partial Consistency of Models	195
6.3.2.	Transformations for Partially Consistent Models .	197
6.3.3.	Transformation Execution Termination	200
6.3.4.	Synchronizing Execution of Transformations	205
6.3.5.	Equivalence to Synchronizing Transformations . .	209
6.4.	Achieving Synchronization	212
6.4.1.	Synchronization Scenarios	213
6.4.2.	Identification of Existing Corresponding Elements .	217
6.4.3.	Model Changes To Condition Element Changes . .	221
6.5.	Summary	225
7.	Orchestrating Transformation Networks	227
7.1.	Orchestration Goals and Problem Statement	229
7.1.1.	Single Transformation Execution	230
7.1.2.	Orchestration Function Behavior	237
7.1.3.	Optimal Orchestration	243
7.1.4.	The Orchestration Problem	245

7.2.	Limitations of Orchestration Decidability	247
7.2.1.	An Algorithm for Application Functions	248
7.2.2.	Correctness and Termination of the Algorithm . . .	251
7.2.3.	Undecidability of the Orchestration Problem	254
7.2.4.	Restriction of Transformation Networks	262
7.2.5.	Confluence in Transformation Networks	265
7.3.	Conservative Approximation of the Orchestration Problem .	267
7.3.1.	Systematic Improvement of Optimality	268
7.3.2.	Dynamic Detection of Alternation	271
7.3.3.	Monotony for Avoiding Alternation	273
7.4.	A Conservative Application Algorithm	278
7.4.1.	Design Goals	279
7.4.2.	The Provenance Algorithm	281
7.4.3.	Correctness, Termination and Goal Fulfillment . .	284
7.4.4.	Provenance Identification Improvement	289
7.5.	Summary	290
8.	Classifying Errors in Transformation Networks	293
8.1.	Knowledge Levels in Transformation Network Specification	294
8.1.1.	Knowledge-Dependent Specification Levels	296
8.1.2.	Abstraction to Specification Levels	297
8.2.	Categorization of Errors in Transformation Networks . . .	298
8.2.1.	Mistakes, Faults and Failures	298
8.2.2.	Possible Failure Types	300
8.2.3.	Mistake and Fault Types	304
8.2.4.	Causal Chains	307
8.3.	Detection and Avoidance of Errors	311
8.3.1.	Error Avoidance	311
8.3.2.	Error Detection	313
8.4.	Summary	314
9.	Evaluation and Discussion	317
9.1.	Compatibility	319
9.1.1.	Goals and Methodology	319
9.1.2.	Prototypical Implementation	322
9.1.3.	Case Study	323
9.1.4.	Results and Interpretation	324
9.1.5.	Discussion and Validity	328
9.1.6.	Limitations and Future Work	330

9.2.	Errors, Orchestration and Synchronization	334
9.2.1.	Goals and Methodology	334
9.2.2.	Prototypical Implementation	340
9.2.3.	Case Studies	343
9.2.4.	Results and Interpretation	351
9.2.5.	Discussion and Validity	360
9.2.6.	Limitations and Future Work	365
9.3.	Orchestration Algorithm	368
9.3.1.	Goals and Methodology	368
9.3.2.	Scenarios	369
9.3.3.	Discussion and Validity	374
9.3.4.	Limitations and Future Work	376
9.4.	Conclusions	378
9.4.1.	Overall Limitations and Future Work	378
9.4.2.	Summary	380
A.	Appendix	381
A.1.	Compatibility Proofs	382
Bibliography	389	

List of Figures

1.1.	Tools and distributed knowledge in engineering processes	6
1.2.	Process of specifying and executing a transformation network	8
1.3.	Consistency relation for PCM and UML/Java	11
1.4.	Example for transformation orchestration	12
1.5.	Example for transformation synchronization	13
1.6.	Example for transformation contradictions	14
1.7.	Example for network topologies	16
1.8.	Problem statements and challenges	18
1.9.	Context, problems, research questions and contributions	22
2.1.	Simplified EMOF metamodeling language	28
2.2.	Simplified Ecore metamodeling language	29
3.1.	Process for preserving structural and behavioral consistency	39
3.2.	Roles in a transformation network specification process	40
3.3.	Three metamodels with exemplary consistency relations	47
4.1.	Execution alternatives of consistency preservation rules	56
4.2.	Declarative and imperative consistency specification	57
4.3.	Consistency specification execution process and artifacts	59
4.4.	Notions of correctness for consistency and its preservation	61
4.5.	Monolithic consistency relation that cannot be modularized	67
4.6.	Example for incompatible consistency relations	68
4.7.	Unidirectional consistency preservation in networks	71
4.8.	Example for necessity of a witness structure	87
4.9.	Examples for fine-grained consistency relations	88
4.10.	Conceptual model for transformation networks	94
4.11.	Example for concept visualizations	96
5.1.	Three metamodels with (in)compatible consistency relations	98
5.2.	Example for an intuitive notion of incompatibility	99

5.3.	Consistency relations that imply an empty global relation	102
5.4.	Example for obsolete elements in consistency relations	103
5.5.	Concrete scenario with obsolete relation elements	104
5.6.	Example for the unwanted rejection of a user change	106
5.7.	Exemplary overview of compatibility analysis idea	110
5.8.	Examples for consistency relation concatenation	113
5.9.	Different incompatibility scenarios	120
5.10.	Two independent sets of consistency relations	125
5.11.	A hypertree with its host graph	127
5.12.	A consistency relation tree	128
5.13.	Construction of a model tuple for a consistency relation tree . .	129
5.14.	Redundant consistency relation	133
5.15.	Incompatibility with redundant consistency relation	135
5.16.	QVT-R transformations for the running example	150
5.17.	Property graph for the running example	154
5.18.	Dual of the property graph for the running example	159
5.19.	Redundancy of a hyperedge	165
5.20.	Redundancy test overview	168
6.1.	Duplicate creation of an element	178
6.2.	Nonalignment of unidirectional relations and preservation . . .	182
6.3.	Sequencing unidirectional consistency preservation rules . . .	189
6.4.	Non-transformability in sequencing scenario	190
6.5.	Multiple execution of consistency preservation rules	192
6.6.	Synchronizing bidirectional transformation execution step . .	205
6.7.	Feature model for changes in Ecore-based models	222
7.1.	Necessity of executing a transformation multiple times	232
7.2.	Example for arbitrary bounds of transformation execution . . .	233
7.3.	Consistency preservation rules without orchestration	240
7.4.	Cycle elimination in Turing machine transition functions . . .	255
7.5.	Confluence of transformations	266
7.6.	Screenshot of the transformation network simulator	270
7.7.	Exemplary execution of the provenance algorithm	283
8.1.	Categorization of mistakes, faults and failures	301
8.2.	Adaptation of consistency relations from running example . .	307
8.3.	Examples for mistakes at each level	308

9.1.	Phases of second case study	351
9.2.	Number of occurrences of mistake types	356
9.3.	Example scenario with incompatibility	370
9.4.	Example scenario with arbitrary execution bound	373

List of Tables

2.1.	Notations for sets, tuples, sequences and functions	32
3.1.	Models, metamodels, their elements and notations	44
5.1.	Mapping between primitive type representations	170
8.1.	Knowledge levels in transformation network specification . .	295
8.2.	Avoidance and detection of mistakes at specification levels .	311
9.1.	Goals, questions, metrics for compatibility	320
9.2.	Example scenarios with compatibility classification	324
9.3.	Correctness of compatibility classification results	324
9.4.	Goals, questions, metrics for categorization and orchestration .	336
9.5.	Goals, questions, metrics for synchronization	339
9.6.	Consistency relations between PCM and UML/Java	344
9.7.	Consistency relation between UML and Java	346
9.8.	Complexity of case study transformations	347
9.9.	Number of test cases for case studies	348
9.10.	Mistakes, faults and failures in case studies	352
9.11.	Mistake types by case study phase	359
9.12.	Goals, questions, metrics for orchestration	368

List of Algorithms

1.	Proof for compatibility of consistency relations	141
2.	Merge of properties to predicates	156
3.	Enumeration of alternative paths	162
4.	Execution of a bidirectional transformation	199
5.	Synchronizing execution of a bidirectional transformation . . .	207
6.	Retrieval of corresponding elements	220
7.	Application function implementation	249
8.	Provenance application algorithm	282

List of Definitions

4.1.	Definition (Model-Level Consistency Relation)	65
4.2.	Definition (Model-Level Consistency)	66
4.3.	Definition (Change)	72
4.4.	Definition (Consistency Preservation Rule)	73
4.5.	Definition (Consistency Preservation Rule Correctness)	74
4.6.	Definition (Synchronizing Transformation)	75
4.7.	Definition (Synchronizing Transformation Correctness)	75
4.8.	Definition (Hippocratic Synchronizing Transformation)	75
4.9.	Definition (Consistency to Transformation)	76
4.10.	Definition (Transformation Orchestration Function)	77
4.11.	Definition (Transformation Generalization Function)	79
4.12.	Definition (Transformation Application Function)	80
4.13.	Definition (Transformation Application Function Correctness) .	81
4.14.	Definition (Transformation Network)	82
4.15.	Definition (Transformation Network Correctness)	82
4.16.	Definition (Condition)	84
4.17.	Definition (Consistency Relation)	85
4.18.	Definition (Consistency)	86
4.19.	Definition (Symmetric Consistency Relation Set)	89
5.1.	Definition (Consistency Relations Concatenation)	112
5.2.	Definition (Consistency Relations Transitive Closure)	116
5.3.	Definition (Compatibility)	119
5.4.	Definition (Consistency Relations Equivalence)	123
5.5.	Definition (Consistency Relation Sets Independence)	125
5.6.	Definition (Consistency Relation Tree)	128
5.7.	Definition (Compatibility-Preserving Consistency Relation) . .	131
5.8.	Definition (Redundant Consistency Relation)	132
5.9.	Definition (Left-Equal Redundant Consistency Relation) . . .	136
5.10.	Definition (Property Set)	144
5.11.	Definition (Tuple of Property Sets)	145

5.12. Definition (Property Value Set)	145
5.13. Definition (Predicate)	145
5.14. Definition (Property Matching)	146
5.15. Definition (Predicate-Based Consistency Relation)	146
5.16. Definition (Property Graph)	152
5.17. Definition (Dual of a Property Graph)	158
6.1. Definition (Unidirectional Consistency Preservation Rule)	179
6.2. Definition (Unidirectional Preservation Rule Correctness)	180
6.3. Definition (Bidirectional Transformation)	185
6.4. Definition (Bidirectional Transformation Correctness)	186
6.5. Definition (Partial Consistency)	197
6.6. Definition (Bidirectional Transformation Execution Step)	198
6.7. Definition (Partial Consistency Improvement)	201
6.8. Definition (Synchronizing Bidirectional Execution Step)	206
6.9. Definition (Synchronizing Bidirectional Transformation)	208
7.1. Definition (Orchestration)	230
7.2. Definition (Optimal Orchestration Function)	244
7.3. Definition (Optimal Application Function)	245
7.4. Definition (Orchestration Problem)	246
7.5. Definition (Orchestration Existence Problem)	246
7.6. Definition (Alternation of Apply Algorithmus)	272
7.7. Definition (Monotone Synchronizing Transformation)	274
7.8. Definition (Reactive Converging Transformations)	281

List of Theorems

5.1.	Lemma (Concatenation Consistency)	114
5.2.	Lemma (Relation Set Consistency)	117
5.3.	Lemma (Transitive Closure Consistency)	117
5.4.	Lemma (Transitive Closure Compatibility)	121
5.5.	Theorem (Independent Relation Sets Compatibility)	126
5.6.	Theorem (Consistency Relation Tree Compatibility)	129
5.7.	Lemma (Redundant Relations Equivalence)	132
5.8.	Proposition (Redundant Relations Non-Compatibility)	134
5.9.	Lemma (Left-Equal Redundancy to Redundancy)	136
5.10.	Lemma (Left-Equal Redundancy Containment)	137
5.11.	Theorem (Left-Equal Redundancy Compatibility)	138
5.12.	Corollary (Transitive Redundancy Compatibility)	139
5.13.	Theorem (Compatibility Algorithm Correctness)	140
5.14.	Theorem (Compatibility Algorithm Conservativeness)	142
6.1.	Lemma (Bidirectional Transformation Execution Consistency) .	200
6.2.	Lemma (Bidirectional Transformation Execution Termination) .	203
6.3.	Theorem (Synchronizing Transformation Termination)	207
6.4.	Theorem (Synchronizing Transformation Expressiveness)	210
7.1.	Lemma (Minimal Number of Transformation Executions)	235
7.2.	Theorem (Orchestration with Single Execution)	236
7.3.	Lemma (Application / Orchestration Function Optimality)	245
7.4.	Theorem (Orchestration / Existence Problem Equivalence)	246
7.5.	Theorem (Optimality / Orchestration Problem Equivalence)	247
7.6.	Theorem (Apply Algorithm Correctness)	252
7.7.	Theorem (Upper Bound for Shortest Consistent Orchestration) .	253
7.8.	Lemma (Halting to Orchestration Problem Reduction)	258
7.9.	Theorem (Orchestration Problem Undecidability)	260
7.10.	Corollary (Application Function Non-Optimality)	260
7.11.	Corollary (Apply Algorithm Non-Optimality)	261

7.12. Lemma (Monotone Transformation Orchestration Prefixes)	275
7.13. Theorem (Monotone Transformations Prevent Alternation)	276
7.14. Theorem (Provenance Algorithm Termination)	284
7.15. Theorem (Provenance Algorithm Correctness)	284
7.16. Theorem (Provenance Algorithm Complexity)	286
7.17. Theorem (Provenance Algorithm Design Principle)	286
7.18. Theorem (Provenance Algorithm Optimality)	287
A.1. Lemma (Consistency Relation Tree Unique Paths)	382

List of Insights

1.	Insight (Correctness Notion)	95
2.	Insight (Compatibility)	173
3.	Insight (Synchronization)	226
4.	Insight (Orchestration)	291
5.	Insight (Errors)	315

Acronyms

COTS Components-off-the-Shelf. 7, 13, 21

DSL Domain-specific Language. 5

ECU Electronic Control Unit. 4

EMF Eclipse Modeling Framework. 28, 71, 221, 322

EMOF Essential Meta Object Facility. 28, 221, 222

GQM Goal Question Metric. 318

MBSE Model-based Software Engineering. 3

MDA Model-driven Architecture. 322

MDSD Model-driven Software Development. 4

MGG Multi Graph Grammar. 297

MOF Meta Object Facility. 28, 43

OCL Object Constraint Language. 45, 172, 322, 323, 326, 327, 329, 331

PCM Palladio Component Model. 5, 6, 10–14, 16, 21, 36, 39, 104, 105, 108, 110, 177, 193, 194, 217, 218, 231–233, 341–351, 354, 358, 360, 365, 366, 370–372

QVT-O QVT Operations. 58, 93, 251

QVT-R QVT Relations. 58, 83, 93, 100, 164, 172, 185, 219, 225, 251, 297, 319, 321–323, 325, 326, 331, 362, 365

QVTD QVT Declarative. 322, 323

SMT Satisfiability Modulo Theories. 323, 326, 327, 330, 331, 333

TGG Triple Graph Grammar. 58, 175, 185, 187, 297

UML Unified Modeling Language. 4, 6, 10–14, 16, 36, 37, 42, 63, 341–351,
353–355, 358–360, 365, 366, 370–372

XML Extensible Markup Language. 5

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*.

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. In the following, we denote all these artifacts as *models*. This reduces the information to deal with to what is relevant for the development task of each person's role. 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)* [Est+07]. For example, the development of software for Electronic Control Units (ECUs) in automobiles

comprises different tools or standards for specifying the system and software architecture, such as SysML [Obj19] or AUTOSAR [Sch15], for defining the behavior, such as MATLAB/Simulink [Mat] or ASCET [ETA], and for defining the deployment on multi-core hardware architectures, such as Amalthea [ITE; 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)* [SV06]. Such a development process foresees other artifacts beyond code as primary artifacts to describe the system under construction. While code only specifies the functionality of a system, other tools can be used, for example, to define the software architecture and its deployment, such as the UML [Obj17], analyzing and predicting the software performance, such as the Palladio Simulator [Reu+16], and for specifying the requirements, like IBM Rational Doors [Lap13].

Accidental complexity due to information fragmentation

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 of overlapping information are not propagated properly 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 software architecture have to be reflected in the code. Since systems are usually developed iteratively and incrementally, there is no strict direction in which changes have to be propagated to preserve consistency, but in general any model can be changed and require updates of others.

Consistency of fragmented information

The overlaps of information, for example in the above mentioned 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. The automation of checking and of preserving consistency of information is still poorly supported in current development processes for large systems, as a recent survey has shown [Gui+18]. But automating that process is necessary to reduce the accidental complexity induced by the fragmentation of information across multiple models.

Transformations for binary consistency

A common approach to automate the process of checking and preserving consistency of models are *incremental model transformations*. Tools describe their models in specific languages, for example denoted by Extensible Markup

Language (XML) schemes or Domain-specific Languages (DSLs). A transformation specifies how models of one or more of 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 how consistency of their instances can be restored after changes in any of them, is particularly well researched. System development usually involves more than two tools and thus models of more than two languages to be kept consistent. The use of transformations to check and preserve consistency between more than two models is, however, less researched [Ste20]. It gained recent attention in a dedicated Dagstuhl seminar [Cle+19].

1.1.2. Distributed and Reusable Consistency Knowledge

Two general approaches for consistency of multiple models by means of transformations are *multidirectional* transformations or the combination of multiple transformations to networks of them. A single multidirectional transformations is beneficial from a theoretical perspective, because it is not prone to contradictions between the transformations to combine and it provides higher expressiveness [Ste20]. For practical application, however, multidirectional transformations suffer from missing modularity, as they require a single person or team to define the overall relations between all languages. Additionally, it is difficult to think about complex multiary relations between models of multiple languages [Ste20] and, even worse, the knowledge required 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, to perform an architecture-based prediction of the system's performance and knows how this description is reflected in the system implementation in 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 component-based architecture models in PCM. Finally, a requirements engineer may use IBM Rational Doors and know how they have to be reflected in the architecture specification

Multidirectional transformations vs. networks

Distributed knowledge about consistency

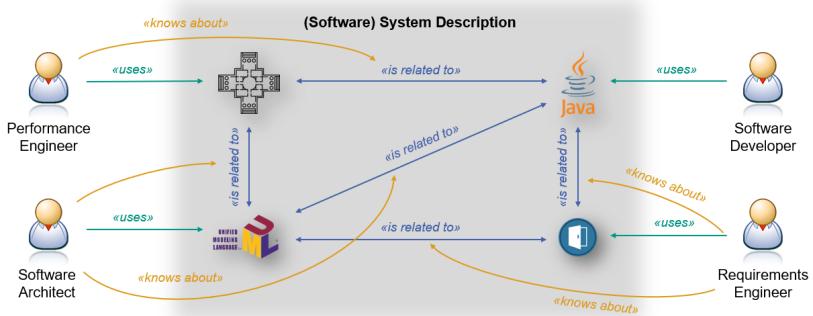


Figure 1.1.: Different tools and roles involved in an exemplary software development process and distributed knowledge about the relations between models of the different tools.

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 as soon as 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.

Reuse of
consistency
specifications

Furthermore, an automation especially proposes benefits if it is used often. A specification of consistency and its preservation between common languages, such as UML and a programming language like Java, can be reused across multiple projects. Not each 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 Components-off-the-Shelf (COTS). 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.

Context assumptions

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:

Modularity: 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.

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

1.1.3. Orchestration of Transformation Networks

Combining several modular and independently developed transformations requires their *orchestration*, i.e., the decision in which order the transformations need to be 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 [Ste20]. Such approaches either require a manual decision for the orchestration or restrict the execution to specific topologies, such as directed acyclic graphs or trees. In any case, strong assumptions to the individual transformations or the topology of the supported networks are made.

Limitations of transformation orchestration

It is yet unclear how arbitrary modular and independently developed transformations can be combined in a universal way. It is neither known how a 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 attributes of the network such as *comprehensibility*.

Universal combination of specifications

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

Envisioned properties and process

Universality: The approach shall be able to process transformation networks of arbitrary topology. In particular, specific topologies cannot be

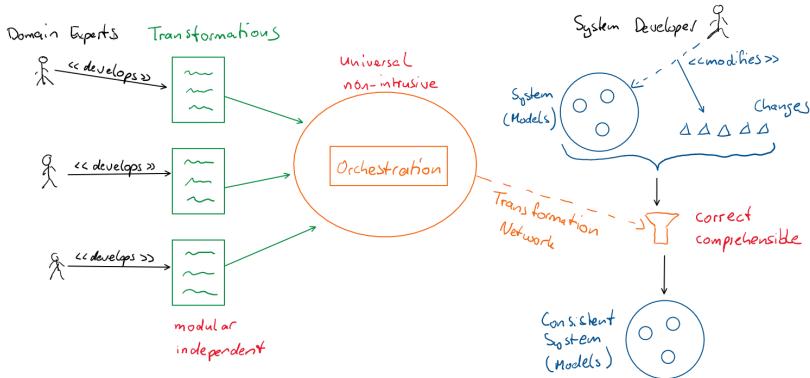


Figure 1.2.: The process of specifying and executing a transformation network. Artifacts defined by transformation developers are marked green, artifacts at runtime are marked blue and the artifacts for orchestration and application developed in this thesis are marked orange. The assumed and envisioned properties are denoted in red.

assumed or prescribed if the assumption of independent development shall be supported.

Non-intrusiveness: The approach shall be non-intrusive. 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.

Correctness: The approach shall operate correctly. If the approach applies transformations to preserve consistency of models, the result must be consistent or indicate an error.

Comprehensibility: The approach shall improve comprehensibility. If the transformations are not able to produce 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 transforma-

tions of the network are applied to these models and the performed 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 when it is not able to execute transformations such that they deliver consistent models. Additionally, we classify which kinds of errors can occur when the artifacts are not defined correctly. We also analyze how topologies of networks affect the desired properties and propose an approach of defining transformations that resolves trade-offs between the envisioned properties. Finally, we propose an approach to integrate artifacts of arbitrary languages into such a consistency process to ensure that all artifacts describing a system can be kept consistent.

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

Thesis contributions

Towards detailed problem statement

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 engineering scenario

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 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 to relate UML and Java, the extracts that we consider follow a simple one-to-one mapping. The relevant relations between classes 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.

1.2.1. Correctness of Transformation Networks

The central goal of (software) engineering, and thus also the construction of transformation networks as part of the engineering process, is to achieve *correctness* of the developed artifacts.

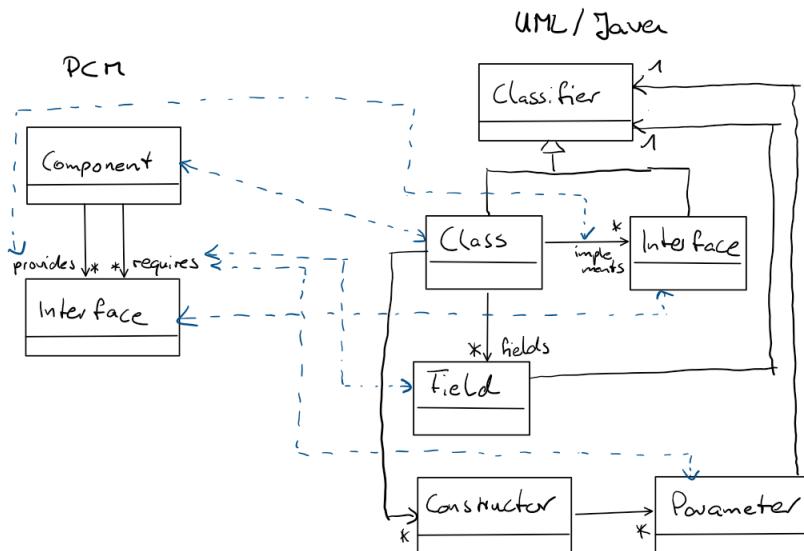


Figure 1.3.: Extract of consistency relations between component-based architectures in PCM and object-oriented design in UML/Java according to [Lan17]. Properties are omitted, each element has at least a name.

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 class in UML, which in turn creates a class in Java. Although in most cases each transformation only needs to be executed once, there can be situations that require transformation to be executed repeatedly.

Single execution of transformations

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 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

Multiple executions of same transformation

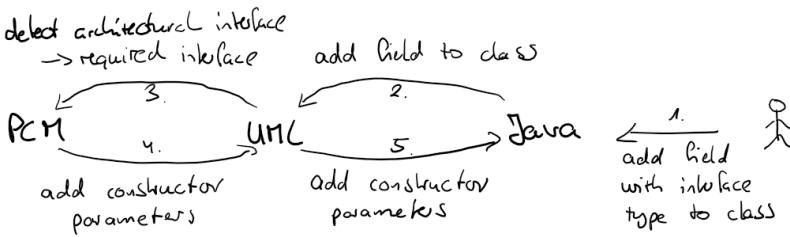


Figure 1.4.: Duplicate transformation execution after adding a field representing a required interface to a Java class.

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, in general, it is 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 then the first transformation may need to react to the changes of second one again, like in the example the transformation between UML and Java needs to react to the one between PCM and UML, because both modified the UML model. The determination how often and in which order transformations have to be executed is what we call the *orchestration challenge*.

Synchronization Challenge

We yet considered that we only have 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 that cannot be represented in UML, which require to also define a 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 method implementations and the abstract service specifications in

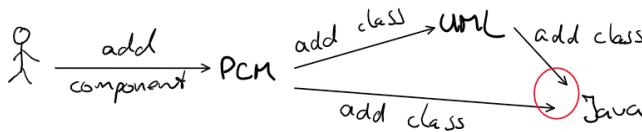


Figure 1.5.: Two transformations propagating the same information to Java.

PCM. In consequence, the graph induced by the transformation contains a cycle.

Instead of only having a transformation for that overlapping information of PCM and Java that cannot be expressed across UML, the transformation may also contain the relations already expressed across UML. The 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 that 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 no UML is used or when the transformation is not supposed for a specific network of transformations, comparable to COTS. In consequence, we need to face the situation that multiple transformations propagate the same information, i.e., they contain redundancies.

Redundancies in transformations

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 define the creation of an appropriate Java class. These transformation now have to consider that there may be another transformation that already created that class. Otherwise, there is the risk of creating a duplicate of that class or of overwriting the already created one.

Two transformation paths creating Java class

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*.

Synchronization challenge

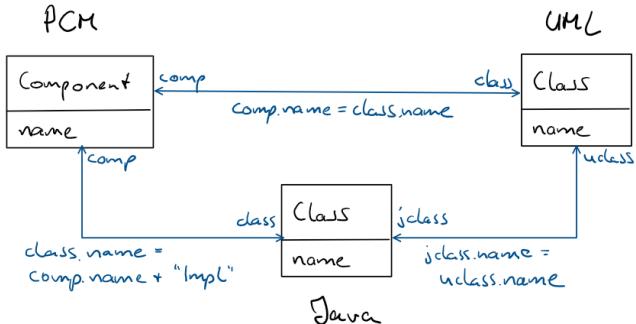


Figure 1.6.: Contradicting consistency relations between components in PCM and classes in UML and Java.

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 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 a component or class appropriate elements in the other models have to exist with 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 and the PCM is also supposed to have the same name as the UML class, it is supposed to have the name of the Java class with an “Impl” suffix, as proposed by Langhammer [Lan17].

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

Equivalence
of redundancies

Contradicting
relations

Different
notions of
consistency

If the domain experts encode the defined relations in transformations that try to 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, there are several quality properties that need to be improved. They can range from properties that are relevant for developers, such as reusability and evolvability, to properties relevant for users, such as performance, scalability and reliability. This similarly applies to transformation networks as artifacts of the (software) engineering process.

Non-termination or inconsistent termination

Contradiction challenge

Systematic knowledge on correctness issues

Quality properties of networks

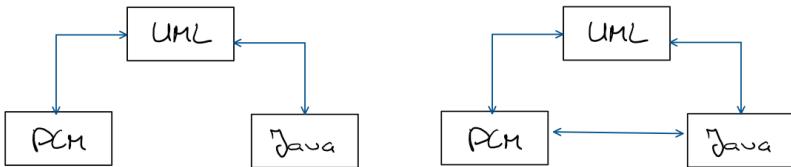


Figure 1.7.: Different transformation network topologies for PCM, UML and Java.

Properties and Topology Challenge

Focus on development properties

In this thesis, we focus on further properties regarding the development of a transformation network, such as reusability and evolvability, rather than properties of its usage, such as scalability. Reusability is of most importance, because transformation may be used in different contexts within different networks of other transformations.

Topology extremes

Consider the two networks depicted in Figure 1.7. The networks contain transformation 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 dense graph.

Topologies affect properties

It is easy to see that properties are directly affected by the network topology. A dense graph has the benefit of high reusability, because any subset of tools can be used for a development project without losing consistency. In the example, the tree network is not applicable in development projects not using UML, because then PCM and Java cannot be kept consistent. Additionally, a dense 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 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 avoid 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 dense 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*.

Topologies induce trade-offs

Language Challenge

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

Language support for specific topology

To relieve the developer from the task of following the strategy to a network of transformations to improve different properties, a suited language should be provided. Investigating design options for such a language constitutes our *language challenge*.

Language 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 that insight can be used to systematically improve different properties by building transformation networks of specific topologies. Thus, we derive the following problem statement.

Impact of topologies and usability for mitigating trade-offs unknown

Problem Statement 2

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

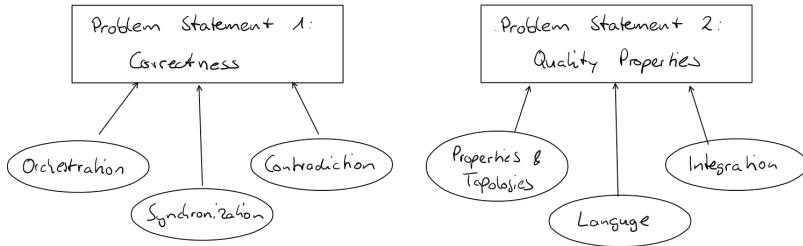


Figure 1.8.: The two identified problem statements and their challenges.

1.2.3. Challenges Overview

Summary
of problems
statements
and
challenges

We have discussed several issues regarding the construction of transformation networks. We summarize the problem statements and challenges in Figure 1.8. We made two central problem statements, one regarding the correctness of networks and another regarding the improvement of quality properties. Each of these problems is driven by three challenges. We identified orchestration, synchronization and contradiction to be central challenges for constructing *correct* transformation networks. For the improvement of quality properties, we emphasized that the relation between properties and topologies challenges the identification of a topology to mitigate property trade-offs and to define an appropriate language for that.

1.3. Research Objective

Research
questions,
assumptions
and
contributions

We have identified the central problems and specific challenges that we have to deal with when constructing transformation networks. In the following, we derive our research goal and the actual research questions that we will answer in this thesis from those challenges. Afterwards, we summarize the context and the assumptions that we make in our work. Finally, we give an overview of the contributions that we make 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 either 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.

Benefits of achieving goal

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, which of them we can achieve by proper construction of the single transformations, which we can analyze and for which we need to deal with potential incorrectness until their execution.

Correctness questions

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 yet 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 when errors occur due to incorrectness, to find out how incorrectness manifests and how severe it is.

Improving Quality Properties of Transformation Networks

The second part of our research goal concerns quality properties of transformation networks. We want to know 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.

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

RQ 2.1 What are relevant properties and topologies of transformation networks?

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 and **RQ 2.2** directly map to the properties and topologies challenge for identifying how topologies affect properties and how to use them for improving quality properties. **RQ 2.3** then maps to the language challenge for designing an appropriate language that supports the construction of an appropriate topology.

1.3.2. Context and Assumptions

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

Model-driven processes

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 COTS, we assume that transformations are developed independently. Thus, transformations may not be adapted to be used within transformation networks.

Distributed knowledge and independent development

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 Sub-section 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.

Consistency relation types

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 introduce a class in Java and a transformation has to decide whether that class shall represent a component in PCM 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 is necessary. 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 the case where user decisions need to be included.

Semi-automation

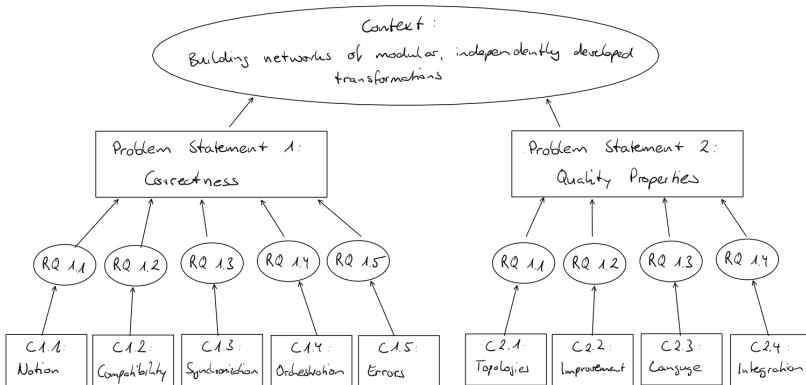


Figure 1.9.: Relation between context, problem statements, research questions and contributions

1.3.3. Contributions

Contribution structure

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. 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 the 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 transformation in that case and propose a strategy to fulfill the required property by construction.

C 1.4 (Orchestration): We prove that transformations, in general, can 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 restrictions to the transformations to achieve its decidability. We propose an algorithm for orchestration that executes the transformations according to a well-defined strategy and helps to find the cause in cases 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 also 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 between the improvement of different properties.

C 2.2 (Improvement): We propose a strategy for building a specific network topology based on auxiliary models, which make the consistency relations explicit in terms of 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 network according to the strategy of C 2.2. We discuss different design options for the language and its operationalization.

1.3.4. Benefits

Overall 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.

Benefits for researchers

Researchers can base on our definitions for correctness of transformation networks and can thus precisely contribute to particular parts of the defined correctness notions, e.g., by 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 trade-offs between quality properties induced by different network topologies.

Benefits for transformation (network) developers

The developers of actual transformation networks can be separated into the 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, i.e., of finding an execution order for transformations. 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 trade-offs between quality properties.

Benefits for transformation (network) users

Finally, the users of a transformation network, i.e., the ones 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, at all. They also profit from an orchestration strategy that supports them in finding and understanding the reasons why the networks may not be able to process certain changes to preserve consistency.

1.4. Thesis Outline

First some general considerations, notation, formal basis etc.

Structured along correctness / quality properties. Each chapter maps to one of the defined contributions

Reading process: Correctness and quality properties are almost independent, so possible to start with any of the parts. Within the parts, consecutive reading recommended. However: Correctness: Start with notion and compatibility chapter, then proceed with any of the following, no strict dependencies. Errors hard to understand without rest. Quality (read first chapter of correctness first): First three chapters depend on each, so do not jump.

In each chapter, we recapture the research question we want to answer in the beginning and point out the contribution that we make with that chapter and we close the chapter with the central insights that the chapter gave.

2. Foundations and Notation

2.1. Modeling

2.1.1. Models and Model Theory

What are models? -> Stachowiak

Definition of metamodels, ingredients: esp. static/execution semantics [Völ+13, p. 26]: static: set of constraints, type system rules language has to conform in addition to structure regarding abstract syntax execution: meaning of the program/model when it is executed, for DSL realized by execution engine

Still Völter: Abstract syntax: data structure holding the semantically relevant information of the model (not layout information), typically a tree or graph
Concrete syntax: notations with which user can express models, e.g. textual, graphical, tabular

Introduce the term "DSL" used in introduction. Refer to XML, as its used in introduction. Say that we always talk about metamodels, which are the constructs on which languages rely.

2.1.2. Model-Driven Software Engineering

Describe what MDSD is supposed to be

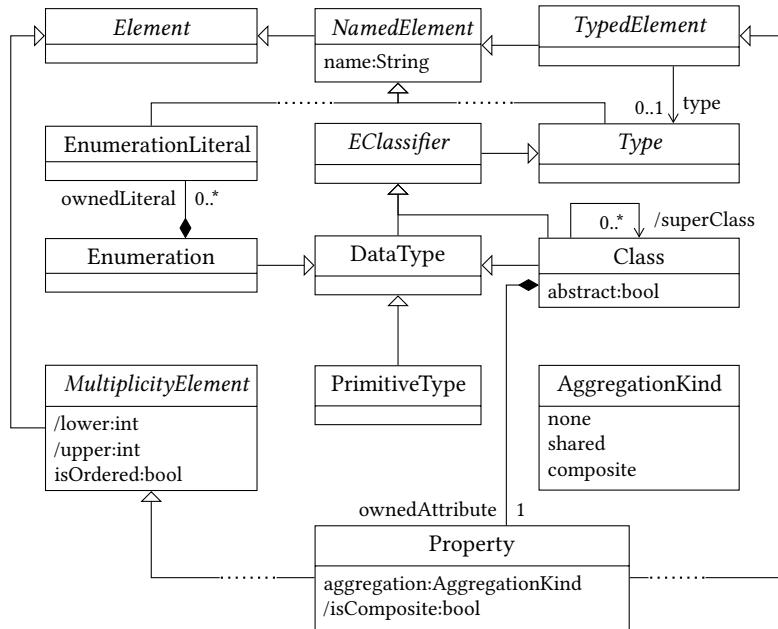


Figure 2.1.: Simplified class diagram showing central metaclasses of the EMOF metamodeling language [Obj16b, p. 27] (dotted lines denote indirect inheritance), adapted from [Kra17, Fig. 2.2].

2.2. Modeling Formalisms and Frameworks

2.2.1. Meta-Object Facility

Discuss Meta Object Facility (MOF), especially Essential Meta Object Facility (EMOF) and how it is used to model things; Discuss levels of UML; Say why MOF is the relevant formalism for us which we base our considerations on.

Modeling Levels M1–M3

2.2.2. EMF and Ecore

Discuss Eclipse Modeling Framework (EMF), Ecore and the relation EMOF

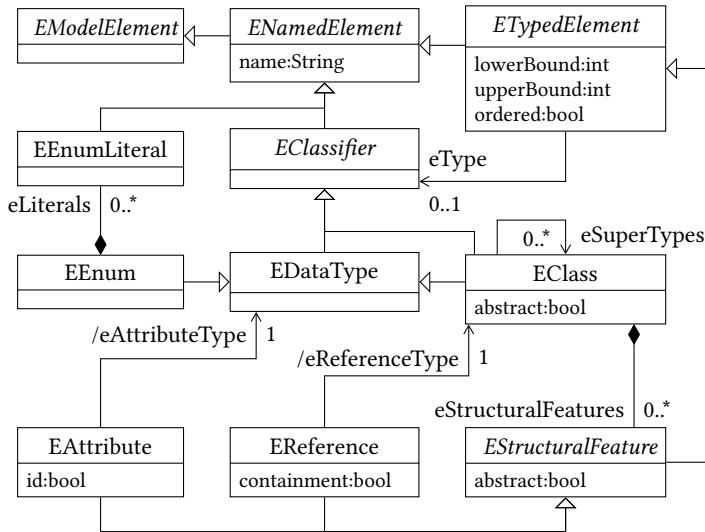


Figure 2.2.: Simplified class diagram showing central metaclasses of the Ecore metamodeling language [Ste+09, p. 97], adapted from [Kra17, Fig. 2.3].

Say that we define an own modelling formalism without precise specifications of the elements models consist of. We therefore rely on EMOF as the most general relevant standard for a modeling formalism.

Add graphics of relevant part of EMOF metamodel here, from Max' Diss

2.2.3. Differences

Discuss differences between Ecore and EMOF, especially those relevant for the changes later and how the differences in general manifest in our formal framework

FROM MAX: Different information and case distinctions are necessary to describe all possible model changes for modelling languages that follow the Essential Meta Object Facility (EMOF) standard or the Ecore variant. Both meta-modelling languages and the differences between them are described in Subsection 2.1.1. Only two differences have a major effect on our change modelling language and the specifications language that use them:

1. In EMOF, properties can be typed using metaclasses or using other data types, but in Ecore these are distinguished as references and attributes.
2. Ecore requires that all elements except for a root element are contained in exactly one container and EMOF only requires that all elements have at most one container [Obj16b, pp. 31-32].

If we only consider these two differences, then Ecore can be seen as a refinement of EMOF, which only adds a more fine-grained distinction of properties and further containment restrictions. Because of this refinement relation, we will first describe which information is necessary to represent model changes of EMOF-based models and then add further information and distinctions for Ecore-based models. Finally, we briefly explain how we made all this information available in practice using a change modelling language.

2.2.4. Simplification

Discuss how far we simplify the modeling concepts (or maybe do that later in the notation / formalization discussion at the end of foundations or in networks chapter)

2.3. Multi-View Modeling

Refer to multi-paradigm modeling, discuss essential idea of using multiple views, mainly refer to introduction

2.3.1. Orthographic Software Modeling

Focus on views, how to create and manage them. SUM idea

2.3.2. The VITRUVIUS Framework

Focus on construction approach for SUM why sticking to projective view idea of OSM. Combine models by consistency mechanisms rather than having inherent consistency.

Discuss that VITRUVIUS is the underlying motivation for this thesis. Consistency is needed to build a SUM, especially consistency between multiple models. VITRUVIUS puts an abstraction layers of views onto the contributions of this thesis and defines a process of using and applying it.

That that in this thesis the main connection to VITRUVIUS is the motivation and that implementation work for validation purposes was done with the VITRUVIUS framework. Additionally, at some points (Commonalities), the usage of developed concepts provides even more benefits when not only used standalone but in combination with further ideas of VITRUVIUS.

For VITRUVIUS, we have a simple formalism defining consistency on which the formalism in this thesis bases. However, the formalism in this thesis will be much more fine-grained.

2.4. Model Transformations

2.4.1. Bidirectional Transformations

Compare multidirectional transformations with networks of transformations. Refer for other consistency approaches to related work.

2.4.2. Transformation Languages

2.5. Mathematical Notations

For the most of our definition, we use standard mathematical notations. Whenever we derive from that within the thesis, we explicitly denote it and define the used constructed. We use specific formatting especially for sets, tuples, sequences and functions to ease their distinction. We introduce

Notations
overview

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

Table 2.1.: Notations for sets, tuples, sequences and functions

this notation in Table 2.1. Additionally, we define some additional 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., $\{a, b, \dots\}$. Likewise, we denote variables representing tuples of any kinds of elements in gothic font \mathfrak{T} and write elements forming a tuple in angle brackets, e.g., $\langle a, b, \dots \rangle$. Finally, we denote variables representing sequences of any kinds of elements by subsequent square brackets $S[]$ and the definition of a sequence of elements by putting them into square brackets, e.g., $[a, b, \dots]$. 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 $S[] = [s_1, \dots, s_n]$ as:

$$S[] + e := [s_1, \dots, s_n, e]$$

Sequences are mathematically equal to tuples, but we write them differently to make explicitly that they represent an order of potentially equal elements, rather than combining different elements of potentially different types in tuples. This is why we explicitly define an access operator for contained elements of sequences. We derive 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, which 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 described set syntax. In any case, we ensure that the meaning of the variables stay 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 it has a specific relation to another tuple, we define several operators which treat them as sets. For a tuple $t = \langle t_1, \dots, t_n \rangle$, we say that:

$$t' \in t : \Leftrightarrow \exists i \in \{1, \dots, n\} : t' = t_i$$

For two tuples t_1 and t_2 , we define:

$$t_1 \subseteq t_2 : \Leftrightarrow \forall t' \in t_1 : t' \in t_2$$

$$t_1 \cap t_2 := \{t' \mid t' \in t_1 \wedge t' \in t_2\}$$

Note that the intersection of tuples is not a tuple but a set, because we are only interested in getting the elements contained in both tuples but do not need to match their order.

In several situations, we define binary relations, which are sets of pairs, i.e., 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:

$$F_1 \circ F_2(x) := F_1(F_2(x))$$

Relation
concatena-
tion

Functions
and Compo-
sition

3. Models, Consistency and Processes

In this chapter, we discuss general terms and notions to clarify the scope of this thesis. We discuss different dimension of consistency, its specification and consistency 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 that we use for them. Finally, we introduce a running example.

3.1. Dimensions of Consistency, Specification and Preservation

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

We have yet informally considered consistency as the absence of some kind of contradictions in different models. It is, however, unclear when to consider information in models contradictory. Consistency can be considered *normatively* or *descriptively*, 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. No matter what 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 when an architecture specification, be it in UML, PCM, or some other language, is considered consistent to its realization in code, so a transformation normatively defines how consistency is considered.

Following 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 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 and especially transformation validation [AW15], to check whether a transformation is correct with respect to some expectation, reflecting 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, we can distinguish different types of consistency relations. From a pragmatic

Descriptive consistency

Normative specification in thesis

Execution semantics as distinction

perspective, we can at least differentiate between *structural* and *behavioral* consistency relations. 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 code implementation would be considered a behavioral relation, because they need to have the same execution semantics. Thus, the mechanisms for checking these different types of consistency are supposed to be different.

The distinction between structural and behavioral relations is, however, not strict. If, for whatever reasons, two Java code artifacts shall be kept consistent, although they share a behavioral relation it can reduce to a structural relation when exactly the same elements have to be contained. In consequence, one might argue that structural consistency relations are a special case of behavioral relations, for which we do not need to consider the execution semantics of the model but only its static structure.

No strict separation

From a theoretical perspective, we can distinguish between decidable consistency and undecidable consistency relations. While structural consistency relations do not consider execution semantics and are thus decidable, behavioral consistency relations will usually be undecidable, because as soon as we have Turing-complete descriptions of behavior, we cannot decide their semantic equivalence. An approach that checks behavioral consistency can be further distinguished regarding the statements he is able to make regarding consistency relations:

Decidability
as
distinction

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 behavior descriptions.

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 fulfillment of a performance requirement in the implementation.

While universally quantified statements can only be made about decidable consistency relations, existentially quantified and statistical statements can be made about both decidable and undecidable consistency relations, although preferably used for undecidable relations.

At a Dagstuhl seminar regarding multidirectional transformation [Cle+19] different consistency relation scenarios were considered, in which more than two models were related. A central insight was that probably 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, however, may not easily decompose into multiple binary relations between model pairs.

In this thesis, we focus on structural relations. 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 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. 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 are usually hard to enforce but can, in the best case, only be checked. This also includes relations that define quality properties of behavior, such as performance of a behavior specification

Structural
relations
supposed
to be
decompos-
able

Structural
relations in
thesis

Checking
and
preserving
consistency

Enforcing
structural
and
checking
behavioral
relations

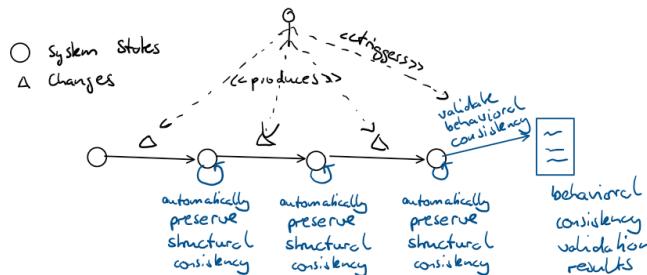


Figure 3.1.: Proposed process for continuously preserving structural and explicitly checking behavioral consistency

regarding performance requirements. On the contrary, structural consistency relations can often also be enforced, at least collecting some additional information from the developer.

In addition, structural consistency relations can and should be checked and enforced 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 to be checked 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 a 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 and checking may 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

Fine-grained preservation of structural relations

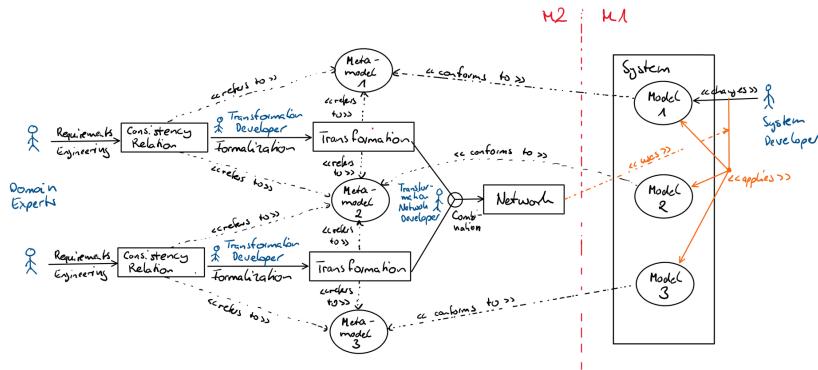


Figure 3.2.: Roles involved in a process for specifying a transformation network, their responsibilities and dependencies. Extended from [TK19].

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 may be affected by his modification with the information about the previous analysis result, such that he or she can guess whether his modification will violate the requirement. Given the information that a response time requirement of 10 milliseconds was fulfilled during the last check by an actual response time of 1 millisecond could help the developer to decide that his modification will not 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, those structural relations that we consider are supposed to be decomposable into binary relations, as discussed in Subsection 3.1.2.

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 different scenarios can be considered that induce requirements and exemplify the application contexts of our later contributions. Figure 3.2 gives an overview of the roles and the essential specification process. While that specification process is concerned with the metamodel level (M2), a transformation network is finally applied at the model level (M1) to a concrete system under development.

Roles and scenarios

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, on the other hand, 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].

Roles in the consistency process

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 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 will usually only refer to the transformation developer, as for us it is not of specific interest 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 will

Roles for the specification

refer to both transformation and transformation network developers simply as *transformation developers*.

Concrete systems are developed with the use of transformation networks by system developers, who perform changes of models via the tools they use. Thus, we also call them tool users. Usually different system developers will be responsible for different models. In our introductory example, we distinguished between software architects, developers, requirements engineers and so on. Performing changes leads to the application of the transformation network to restore consistency of the models. In this thesis, we refer to system developers 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 as a domain expert and formalize them in a transformation. Potentially, a domain expert may even be the one who develops a concrete system as a system developer.

3.2.2. Scenarios

Both for the development of transformations as well as for the combination of transformations 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 (cf. 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 specific for a project. This can occur when a project requires specific rules how elements shall be related. For example, the mapping of components to their realization in the implementation can be specific to the project [Lan17]. Project-specific transformations can, eventually, 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.

Roles for the usage

Multiple roles fulfillable by same persons

Generic and project-specific transformations

Big bang and continuous combination

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 on the compatibility of their developed transformations with others on-the-fly. This makes locating a problem easier, because only the last 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 both generic as well as project-specific transformations can be mixed in a single project, also the process of combining them may be mixed. Some of the transformations may be integrated in a big bang fashion, whereas others are continuously integrated. This can also be a result of where the transformations come from. A generic transformation cannot be continuously integrated, because it is not specific for a single project to integrate it into.

Effects of
combination
process

Mixing
processes

Notation
overview

3.3. Models and Metamodels

The most essential elements used for descriptions in this thesis are models and the metamodels describing them. We have already introduced in Chapter 2 what we consider a model and that we adhere to the modeling formalism defined by the MOF. We use a sufficiently simplified notion of models, metamodels, and their elements, for which we give an overview in Table 3.1. In the following, we introduce the used notation and its conventions, as well

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

as the elements used for modeling. Finally, we clarify assumptions that we make and discuss their impact.

3.3.1. Notation and Conventions

In general, we use variables of uppercase letters for all elements at the metamodel level (M2), such as M for a metamodel or C for a class, whereas we use lowercase letters for all elements at the model level (M1), such as m for a model and o for an object.

We use the notations for sets and tuples introduced in Section 2.5 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 usually 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

Modeling levels

Notation for model and metamodel level relations

further specified, we use the same indices to relate 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.

Elements overview

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 $I_P = \{p_1, p_2, \dots\}$ of property P . We do not need to further differentiate different types of properties into attributes and references, like it is done in other formalizations, such as the Object Constraint Language (OCL) standard [Obj14b, A.1] or the thesis of Kramer [Kra17, p. 2.3.2].

Properties and property values

We denote *meta-classes*, in the following shortly called *classes*, as tuples of properties $C = \langle P_1, \dots, P_n \rangle$. 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 as $I_C = \{o = \langle p_1, \dots, p_n \rangle \mid p_i \in I_{P_i}\}$.

Classes and objects

We denote a metamodel $M = \{C_1, \dots, C_m\}$ 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 like [Ste20], 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 $\mathfrak{M} = \langle M_1, \dots, M_k \rangle$, we denote the set that contains all sets of instances of that metamodels as $I_{\mathfrak{M}} = \{\langle m_1, \dots, m_k \rangle \mid m_i \in I_{M_i}\}$.

Metamodels and models

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

Valid models

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 will, however, usually only refer to the denoted instance sets. The statements still apply to the sets of valid object and models as subsets of considered sets.

3.3.3. Assumptions

Finite
models

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 .

Unique
elements

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 properties of these elements.

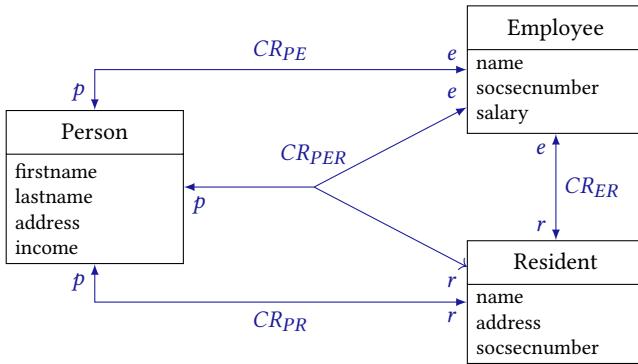
3.4. Running Example

Metamod-
els

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 the concepts in this thesis and are easy to comprehend.

Consistency
relations

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



$$\begin{aligned} CRPER = \{ & \langle p, e, r \rangle \mid p.firstname + " " + p.lastname = e.name = r.name \\ & \wedge p.address = r.address \wedge p.income = e.salary \\ & \wedge e.socsecnumber = r.socsecnumber \} \end{aligned}$$

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

$$CRPR = \{ \langle p, r \rangle \mid p.firstname + " " + p.lastname = r.name \wedge p.address = r.address \}$$

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

Figure 3.3.: Three simple metamodels for persons, employees and residents. One ternary relation $CRPRE$ between them and three binary relations $CRPE$, $CRPR$, $CRER$ between each pair of them describing consistency.

explaining the essential concepts and also several issues in this thesis. This relation can either be expressed as a ternary relation, denoted as $CRPER$, or as three binary relations $CRPE$, $CRPR$, $CRER$. Three models fulfill the ternary relation in exactly those cases in which each pair fulfills the according binary relation. The relations consist of tuples of the elements that are considered consistent, i.e., the pairs or triples of elements with the specified relation 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 salary 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 will first discuss a rather informal notion of consistency and its preservation. It is supposed to discuss 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 not require for the initial overview, but 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 already been published in [Kla+20b], [Kla+20a] and [Kla18]. In [Kla+20b], we have used a simplified version of the formalization that we introduce in this chapter and especially identified the challenge of orchestration, which is central for the formalization of transformation networks. [Kla+20a] discussed compatibility and introduced the fine-grained consistency notion, which was required for detailed statements about compatibility. In [Kla18], we especially motivated and informally derived the correctness notion that we formalize in the following and give an overview of the goal regarding correctness of transformation networks.

4.1. Notions of Consistency and its Preservation

We begin with an informal discussion of different ways to consider consistency. This especially involves *intensional* and *extensional* notions of consistency as well a *monolithic* and *modular* notions.

4.1.1. Intensional and Extensional Consistency Notions

When we consider a tuple of models, we would intuitively say that it is 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 \mathbf{m} fulfills the constraints $P : I_{\mathfrak{M}} \rightarrow \{\text{true}, \text{false}\}$. Then we can say that:

$$\mathbf{m} \text{ consistent to } P \Leftrightarrow P(\mathbf{m}) = \text{true}$$

Alternatively, one can also enumerate the consistent tuples of models. Thus a tuple of models is considered consistent if it is contained in that enumeration. This constitutes an *extensional specification* of consistency. Given such an enumeration $E = \{\mathbf{m} \mid \mathbf{m} \text{ is consistent}\}$, we can say that:

$$\mathbf{m} \text{ consistent to } E \Leftrightarrow \mathbf{m} \in E$$

It is easy to see that both kinds of specifications are equivalent. For each intensional specification the extensional one can be derived by enumerating all models that fulfill the constraints:

$$E = \{\mathbf{m} \mid P(\mathbf{m}) = \text{true}\}$$

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

$$P(\mathbf{m}) \mapsto \begin{cases} \text{true}, & \mathbf{m} \in E \\ \text{false}, & \mathbf{m} \notin E \end{cases}$$

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. This is due to the fact that he or she cannot explicitly enumerate all consistent models 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. 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 the models that are considered consistent in terms of relations, which we also call *consistency relations*.

We use extensional specifications

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, on the other hand, considers several relations between a selection of the relevant models and all together define when models are to be considered consistent.

Intuitive notion

For an extensional notion of consistency between three models m_1, m_2, m_3 , a modular specification could manifest in three relations $CR_{1,2}, CR_{1,3}, CR_{2,3}$ 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. However, we are interested in whether models are *globally* consistent to all these relations, so we can say that:

Modular specification example

m_1, m_2, m_3 are consistent : \Leftrightarrow

$$\langle m_1, m_2 \rangle \in CR_{1,2} \wedge \langle m_1, m_3 \rangle \in CR_{1,3} \wedge \langle m_2, m_3 \rangle \in CR_{2,3}$$

Modular specification for multiary relations

Due to the assumptions we defined in Subsection 1.3.2, we are interested in a modular notion of consistency. In the example, we 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 simplicity, however, we stick to modular notions of binary relations, although our considerations can be transferred to multiary ones.

4.1.3. Consistency Preservation

Consistency preservation goal

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

Consistency preservation as function

Consistency preservation can be considered a function C_P that takes (potentially inconsistent) models and returns a consistent tuple of models:

$$C_P(m) \mapsto m'$$

We would require from that function that:

$$\forall m : C_P(m) \text{ is consistent}$$

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 , C_P is supposed to fulfill that:

$$\forall m : C_P(m) \in CR$$

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

Modular consistency preservation

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 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, they return new models that may need to be merged. Imagine two functions $Cp_{1,2}$ and $Cp_{2,3}$ that preserve consistency for relations $CR_{1,2}$ and $CR_{2,3}$, respectively. Consider the input models $\langle m_1, m_2, m_3 \rangle$ that are not consistent to $CR_{1,2}$ and $CR_{2,3}$. Now if we apply the functions independently, we have $Cp_{1,2}(\langle m_1, m_2 \rangle) = \langle m'_1, m'_2 \rangle \in CR_{1,2}$ and $Cp_{2,3}(\langle m_2, m_3 \rangle) = \langle m''_2, m''_3 \rangle \in CR_{2,3}$. It is now unclear how to unify m'_2 and m''_2 to m'''_2 , such that $\langle m'_1, m'''_2 \rangle \in CR_{1,2}$ and $\langle m'''_2, m''_3 \rangle \in CR_{2,3}$.

An intuitive approach to execute the functions is their composition, i.e. a consecutive execution that does not take the original models as input but the ones delivered by the previous executions of the functions for consistency preservation. If we consecutively apply the two given functions, we know that $Cp_{1,2}(\langle m_1, m_2 \rangle) = \langle m'_1, m'_2 \rangle \in CR_{1,2}$ and $Cp_{2,3}(\langle m'_2, m_3 \rangle) = \langle m''_2, m''_3 \rangle \in CR_{2,3}$. It is, however, unclear whether $\langle m'_1, m''_2 \rangle \in CR_{1,2}$, so it may be necessary to execute $Cp_{1,2}$ 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 examples for the two strategies of executing consistency preservation rules are depicted in Figure 4.1. 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 orchestration or 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 gives the guarantee 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 others performed.

Consecutive function execution

Unification or orchestration necessary

Benefits of consecutive execution

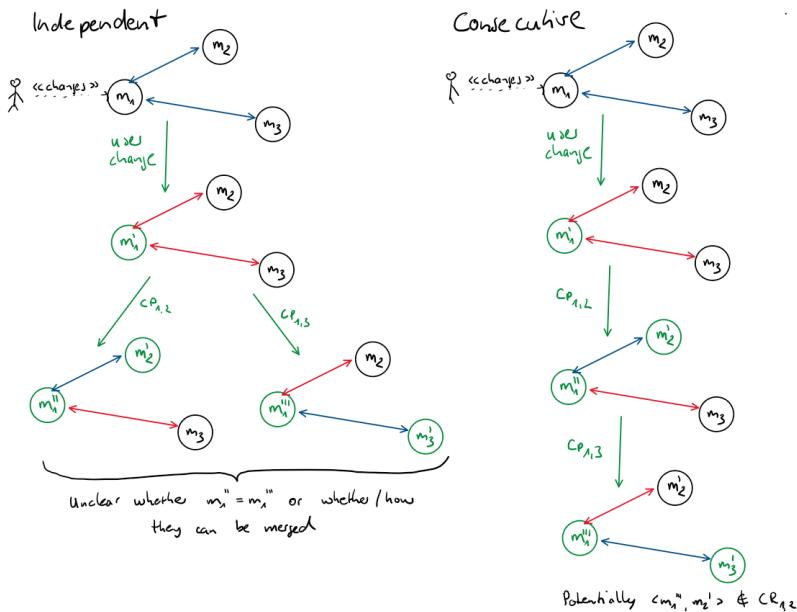


Figure 4.1.: Scenarios for independently executing consistency preservation rules on input models and consecutively executing them on the results of other rules.

Monolithic
notions as
degraded
modular
ones

Realization
options for
consistency
preserva-
tion
rules

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

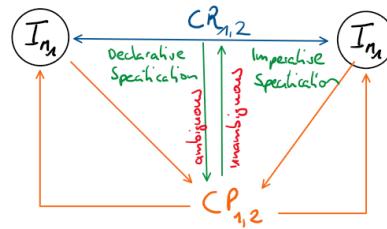


Figure 4.2.: Declarative and imperative specification of consistency relations and consistency preservation rules for two metamodel M_1 and M_2 .

4.1.4. Declarative and Imperative Specifications

We 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 itself and also the consistency preservation rule that preserves it. Instead, usually 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 and a consistency relation and consistency preservation rule is depicted in Figure 4.2.

Only define relation or function

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 is not only a single option how this function can look like. It can, for example, calculate the result with minimal differences to the input, according to some defined metrics. 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 Relations (QVT-R), or Triple Graph Grammars (TGGs).

Ambiguity in deriving rules

As a second option, a developer can define consistency preservation rules without explicitly specifying the consistency relations to which they preserve

Relations as fixed points

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 CP , the relation CR it preserves is implied by its fixed points: $CR = \{m \mid CP(m) = m\}$. 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. 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 Operations (QVT-O).

4.1.5. Consistency Preservation Artifacts

Four 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 artifacts necessary to handle consistency preservation:

Consistency Relation: Binary (or in general multiary) relations that specify when models are to be considered consistent.

Consistency Preservation Rule: Functions that restore consistency for a pair (or in general tuple) of models that became inconsistent by modification.

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.

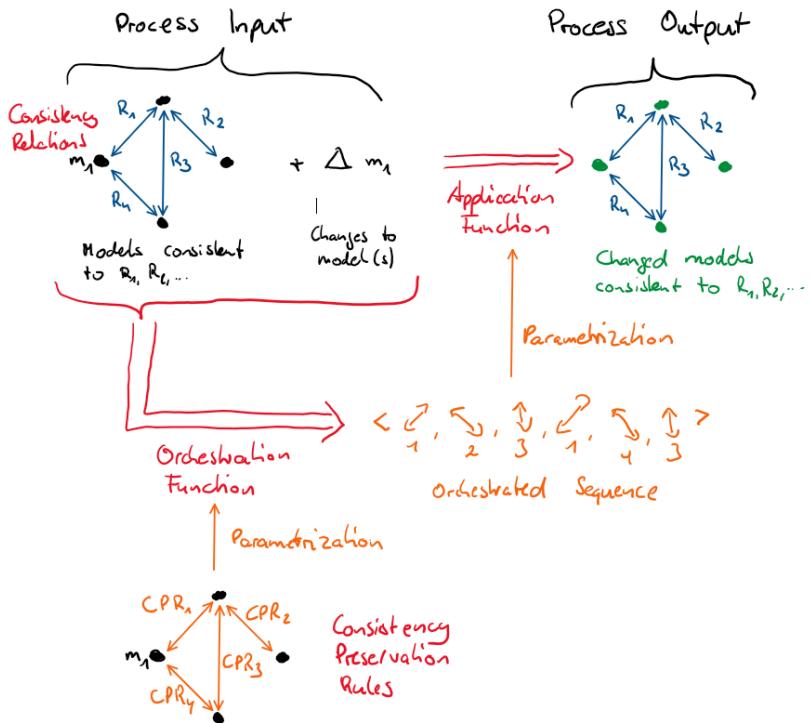


Figure 4.3.: Execution process and artifacts for a modular consistency specification. Relevant artifacts are annotated in red, changes generated by the process are depicted in green, parametrizations of the functions are depicted in orange.

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, especially to determine the behavior in cases when no orchestration is found in which the transformations can be applied to yield consistent models. 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 tuple of models that is consistent to the relations again.

Distinction between orchestration and application

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 actually consistent according to a consistency relation to be considered correct. This also conforms to the notion of correctness for transformations, which realize consistency preservation rules, defined by Stevens [Ste10]. And finally, the orchestration and application functions should execute the consistency preservation rules such that all models are consistent according to all relations afterwards.

Correctness of an artifact is always considered with respect to some other specification, be it formally defined or only an informal notion. For example, consistency relations may be supposed to be 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.

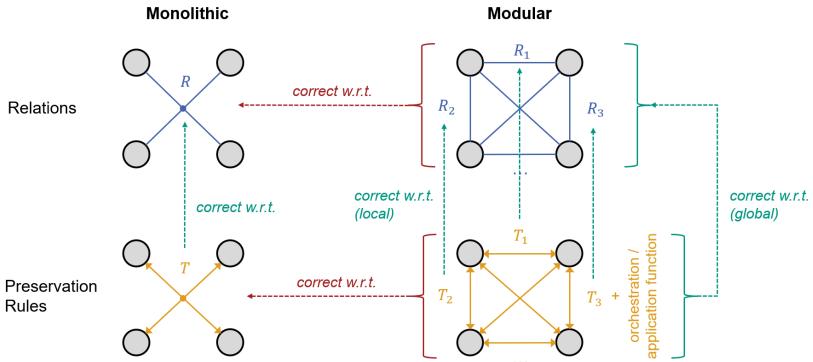


Figure 4.4: Different notions of correctness for consistency and its preservation.

4.2.2. Correctness regarding Global Knowledge

We previously distinguished between monolithic and modular notions of consistency. In the above considerations, we relate 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 tuples of models 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:

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

We may, analogously, define correctness for consistency preservation rules, an orchestration function, and an application function with respect to a single monolithic consistency 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.

Correctness
of modular
w.r.t.
monolithic
specification

4.2.3. Dimensions of Correctness

The discussed correctness notions induce two dimension: 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 dimension and correctness relations 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 means of an orchestration and application function must also be correct with respect to the combination of all relations. The latter dimension is depicted horizontally. Each modular artifact is supposed to be consistent with respect to its corresponding monolithic artifact.

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

Validation Artifacts: The artifacts to check 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. As discussed before, we assume the knowledge about relations between models to be usually distributed across several persons. Thus, there will not be such a global knowledge, thus 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 specification, we do not

further consider this notion of consistency. In this thesis, we focus on correctness between the artifacts of a modular consistency specification.

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, however, does 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 proposed in previous work [Kla+19]. This, however, suffers from the drawbacks discussed before, 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 people know how elements have to be related because they represent similar things, such as classes, or because a standard like the UML [Obj17] prescribes that. In this work, however, we do not consider such a correctness notion with respect to external, maybe not formally specified artifacts, which 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 $CR_{1,2} = \{\langle m_1, m_2 \rangle\}, CR_{1,3} = \{\langle m_1, m_3 \rangle\}, CR_{2,3} = \{\langle m_1, m'_3 \rangle\}$. 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.

Correctness
of relations

No
correctness
by construc-
tion

induction of
monolithic
relation

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

$$CR = \{\langle m_1, \dots, m_n \rangle \mid \bigwedge_{1 \leq i < j \leq n} \langle m_i, m_j \rangle \in CR_{i,j}\}$$

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

Formaliza-
tion of
transfor-
mation
networks
and
correctness

We yet discussed a general notion of consistency and its preservation with a focus on a modular way of specifying it. This notion was only specified 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 [Kla+20b]. 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 for preserving consistency, i.e., we consider specifications that transform changes within one or more models into changes in one or more other models.

Extensio-
nial
specifica-
tions are
relations

4.3.1. Modular Consistency Specification

As already discussed informally before, an extensional specification of consistency enumerates all tuples of models that are considered consistent to each other, i.e., it specifies a relation between the models.

Definition 4.1 (Model-Level Consistency Relation)

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

For a tuple of models $m = \langle m_1, \dots, m_n \rangle \in I_{\mathfrak{M}}$ we say that:

$$m \text{ consistent to } CR : \Leftrightarrow m \in CR$$

Otherwise, we call m *inconsistent to* CR .

Given a tuple of models, we consider it consistent if it is contained in the consistency relation. This conforms to consistency definitions such as the one proposed by Stevens [Ste10] for bidirectional transformations. We explicitly denote this kind of consistency relation as *model-level*, because we will later need to refine the notion of consistency relations to the level of metaclasses and need to distinguish between the two.

If a single relation describes consistency between all relevant models, consistency is directly defined by means of model tuples being contained in that relation. We call such a relation a *monolithic relation*. However, if we have a *modular* notion of consistency, i.e., a relation that does only define consistency between some of the relevant models and the global notion of consistency is defined by a combination of several such relations, we need an explicit definition for that notion. For the sake of simplicity, we focus on binary relations as a modular representation of consistency, but this definition could also be generalized to relations of arbitrary arity.

Start with coarse-grained model-level relations

Modular notions of consistency

Definition 4.2 (Model-Level Consistency)

Let $\mathfrak{M} = \langle M_1, \dots, M_n \rangle$ be a tuple of metamodels and let $CR_{i,j} \subseteq I_{M_i} \times I_{M_j}$ be a binary model-level consistency relation for any two metamodels $M_i, M_j \in \mathfrak{M}$. For a given tuple of models $m = \langle m_1, \dots, m_n \rangle \in I_{\mathfrak{M}}$, we say that m is *consistent* to $CR_{i,j}$ if, and only if, the instances of M_i and M_j are in that relation:

$$m \text{ consistent to } CR_{i,j} : \Leftrightarrow \langle m_i, m_j \rangle \in CR_{i,j}$$

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

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

Exemplary
binary
relations

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

Relation
uniqueness
assumption

For reasons of simplicity, we assume that there is 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 directionality. 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.

Expressiveness
of
modular
specifications

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 that 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

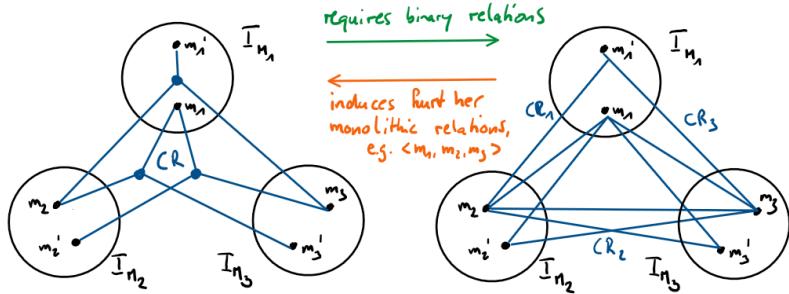


Figure 4.5.: A monolithic consistency relation which cannot be expressed by binary relations.

between all models. Second, a modular definition of consistency can easily contain contradictions, which may lead to an empty tuple of consistent models.

It is easy to see that a combination of binary relations is not able to express the same consistency relations as one monolithic relation. For example, the monolithic relation $CR = \{\langle m_1, m_2, m'_3 \rangle, \langle m_1, m'_2, m_3 \rangle, \langle m'_1, m_2, m_3 \rangle\}$ cannot be expressed by binary relations, as also 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 considered consistent to the binary relations although it is not consistent to the modular relation CR . Thus, using sets of binary relations in contrast to a single monolithic relation reduces expressiveness. Stevens [Ste20] discusses the property of a multiary relation to be expressed 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. However, we reasoned the assumption that relations need to be specified independently and thus modularly anyway, thus we have to accept that these restrictions in expressiveness exists.

Additionally, it is easy to define multiple binary relations of which each can be fulfilled by certain models, but for which no tuple of models exists that is consistent to all of them. Consider the relations $CR_1 = \{\langle m_1, m_2 \rangle\}$, $CR_2 = \{\langle m_2, m_3 \rangle'\}$, $CR_3 = \{\langle m_1, m_3 \rangle\}$, which are also depicted at the left of Figure 4.6. Although for each of these relations a consistent tuple of models

Binary relations reduce expressiveness

Contradictions of binary relations

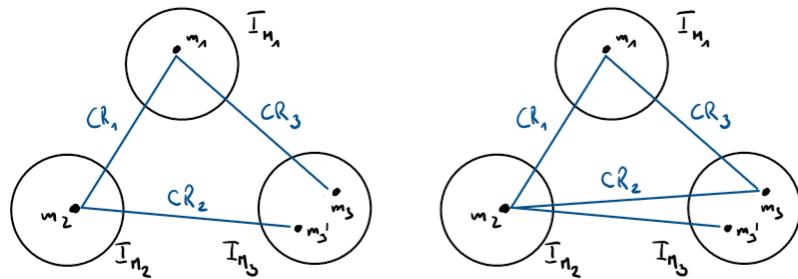


Figure 4.6.: Modular consistency relations, which together cannot be fulfilled (left) or which cannot be fulfilled for some of the consistent model pairs (right).

exists, which is exactly the one defined in each relation, no tuple of models exists that fulfills their combination. This example already demonstrates the worst case, in which no consistent models exist for a set of relations. In other cases, it may be possible that only for some models that are consistent according to one or some of the relations no model tuple exists that is consistent to all relations. Consider the relations $CR_1 = \{\langle m_1, m_2 \rangle\}$, $CR_2 = \{\langle m_2, m_3 \rangle, \langle m_2, m_3' \rangle\}$, $CR_3 = \{m_1, m_3\}$, which are also depicted at the right of Figure 4.6. In this case, the tuple $\langle m_1, m_2, m_3 \rangle$ would be considered consistent to the relations, but although $\langle m_2, m_3' \rangle \in CR_2$ there exists no $m^*_1 \in I_{M_1}$ so that $\langle m^*_1, m_2, m_3' \rangle$ is consistent to all these relations.

It is easy to see that one monolithic relation may 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 actual models and relations, but we will discuss which situations may be unintended in general, independent from the scenario. We already informally discussed this as a notion of *compatibility*, for which we investigate in Chapter 5 how far this behavior is or should be expected.

Forbidden
models
through
relations

4.3.2. Incremental Consistency Preservation

While the previous discussion only considered when models are considered consistent, it is of especial interest to ensure that consistency of models is preserved. We informally introduced such specifications as consistency preservation rules. In the following, we will restrict us 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.

While incrementality is an essential requirement whenever consistency shall be preserved to avoid information loss, inductivity may not be necessary. 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 already discussed for consistency preservation rule in general, incremental preservation rules can be realized in an either monolithic or modular way. A monolithic consistency preservation rules takes a tuple of models that is consistent to a consistency relation and a change to these models and returns another tuple of models that is consistent again. In a modular specification of consistency preservation rules, a set of such rules is given which are able to preserve consistency of a subset of the given models according to modular consistency relations. In our case, we consider such rules for two models, each of them restoring consistency according to a binary consistency relation.

Preserving consistency

Monolithic and modular consistency preservation

Drawbacks
of existing
consistency
preserva-
tion
approaches

In the 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* [Ste20]. Definitions of that terminology do usually not consider changes but only states of models and simply define a consistency preservation rule CPR for metamodels M_1 and M_2 that modifies the instance of M_2 to restore consistency as:

$$\text{CPR} : I_{M_1} \times I_{M_2} \rightarrow I_{M_2}$$

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

State-based: No information about the performed changes that led to the inconsistent state are given. This means that the specification is not aware of how the inconsistent state was reached.

Unidirectional: The unidirectionality of the specification always requires to only update one model to restore consistency.

State-based transformations always suffer from the problem that it is unknown 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 which were moved or renamed can potentially not be reconstructed, leading to elements that are deleted and created anew, 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.

Counterex-
ample for
unidirec-
tionality

Figure 4.7 depicts an example why unidirectional consistency preservation rules cannot be applied of used in combination with others. If the depicted consistency preservation rules CPR_1 and CPR_2 are executed first, CPR_3 cannot be unidirectional, as both involved models m_1 and m_3 have been modified either by the user or by another consistency preservation rule. Thus, in general, it is not possible to only consider changes in one model and unidirectionally propagate them to the other model. In consequence, the rules need to be able to deal with changes performed in both of the input models and, consequentially, need to update both models to reflect the changes in the other.

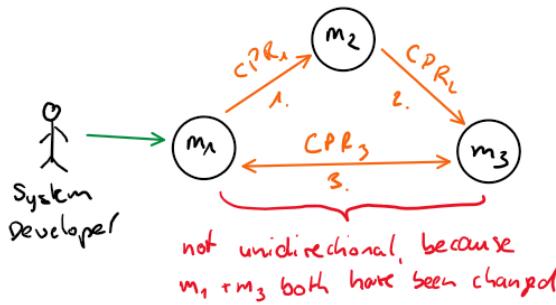


Figure 4.7.: Consistency preservation rules of which at least one cannot be unidirectional because both models are modified by user or other consistency preservation rules.

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 are able to 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 first introduce a notion of *changes* and afterwards of *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 those changes to be functions that take a model and return a new one. They are explicitly not defined for a specific model but for all instances of one metamodel. This is reasonable, 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 in EMF behave. When the elements a change affects are missing a model, applying the change may fail. This is why we consider the function describing a change to be partial.

Synchronizing
consistency
preservation
rules

Changes as
functions

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 :

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

It encodes any kind of modification, which may be just an element addition, or removal, an attribute change and so on, or any composition of changes. We denote the identity change, i.e., the change that always returns the input model, as δ_{id} :

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

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

$$\Delta_M := \{ \delta_M \mid \delta_M \subseteq I_M \times I_M \wedge (\langle m_1, m_2 \rangle, \langle m_1, m_3 \rangle \in \delta_M \Rightarrow m_2 = m_3) \}$$

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

$$\delta_{\mathfrak{M}} = \langle \delta_{M_1}, \dots, \delta_{M_n} \rangle, \delta_{M_i} \in \Delta_{M_i}$$

Likewise, we define the set of all tuples of changes in the instance tuples of \mathfrak{M} , i.e. in $I_{\mathfrak{M}}$, as $\Delta_{\mathfrak{M}}$:

$$\Delta_{\mathfrak{M}} := \Delta_{M_1} \times \dots \times \Delta_{M_n}$$

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

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

For us, it does not matter how the function describing a change 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 for those models, i.e., return the identical model, or even be undefined for those model, i.e., be partial and return \perp .

Unapplicable
changes

In fact, we do restrict the actual behavior of a change in any way. It may return an empty model, independent from the input, or may perform arbitrary changes to different models, instead of affects only specific elements. We omit such restrictions, because they are not necessary for us, although, in practice, they are usually given. Thus, further restricting the formalism would not provide any benefits.

With that notion of changes, we can define consistency preservation rules as functions receiving two models and changes to each own them and returning new changes to both models.

Definition 4.4 (Consistency Preservation Rule)

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

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

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. First, this is because we do not require it to produce any 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 does even allow 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.

Reasonable
change
behavior

Consistency
preserva-
tion
rules

Partial
consistency
preserva-
tion
rules

Confor-
mance to
existing
definitions

4. Correctness in Transformation Networks

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 4.5 (Consistency Preservation Rule Correctness)

Let CPR_{CR} be a consistency preservation rule. We call CPR_{CR} *correct* if it either returns \perp or changes that deliver models that are consistent to CR if applied to the input models again:

$$\begin{aligned} \text{CPR}_{CR} \text{ correct } &:\Leftrightarrow \forall m_1 \in I_{M_1}, m_2 \in I_{M_2}, \delta_{M_1} \in \Delta_{M_1}, \delta_{M_2} \in \Delta_{M_2} : \\ &(\exists \delta'_{M_1} \in \Delta_{M_1}, \delta'_{M_2} \in \Delta_{M_2} : \text{CPR}_{CR}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) = (\delta'_{M_1}, \delta'_{M_2}) \\ &\Rightarrow \langle \delta'_{M_1}(m_1), \delta'_{M_2}(m_2) \rangle \text{ consistent to } CR) \end{aligned}$$

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

We already discussed that it is possible to define consistency relations and derive consistency preservation rules from them or to only define the consistency preservation rules, which then imply the consistency relations by their image, i.e., the set of all models that can be derived from applying the consistency preservation rule to any models and changes for which it is defined. Anyway, 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 4.6 (Synchronizing Transformation)

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

Correctness of a transformation is then given by correctness of its consistency preservation rule.

Definition 4.7 (Synchronizing Transformation Correctness)

Let $t = \langle CR, CPR_{CR} \rangle$ be a synchronizing transformation. We say that t is correct if, and only if, CPR_{CR} is correct according to Definition 4.5:

$$t \text{ correct} : \Leftrightarrow CPR_{CR} \text{ correct}$$

Transformations are usually expected to be *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. The application of hippocracticness to synchronizing transformations can be defined as follows.

Definition 4.8 (Hippocratic Synchronizing Transformation)

Let $t = \langle CR, CPR_{CR} \rangle$ be a transformation for metamodels M_1 and M_2 . We say that t is *hippocratic* if, and only if, it returns the input changes if their application to the input models yields consistent models:

$$\begin{aligned} t \text{ hippocractic} : &\Leftrightarrow \forall m_1 \in I_{M_1}, m_2 \in I_{M_2}, \delta_{M_1} \in \Delta_{M_2}, \delta_{M_2} \in \Delta_{M_2} : \\ &\langle \delta_{M_1}(m_1), \delta_{M_2}(m_2) \rangle \in CR \Rightarrow CPR_{CR}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) = (\delta_{M_1}, \delta_{M_2}) \end{aligned}$$

Although hippocracticness is not a necessary requirement for our considerations in most cases, it is usually a desired property in practice [Ste10]. One benefit of hippocracticness 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

Transformation
correctness

Hippocratic
ness

Benefits of
hippocratic
ness

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 will only refer to transformations rather than consistency relations and consistency preservation rules if the distinction is not necessary. We will 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 4.9 (Consistency to Transformation)

Let $t = \langle CR, C_{PR_{CR}} \rangle$ be a synchronizing transformation. We say that any tuple of models m is *consistent to t* if, and only if, it is consistent to its consistency relation, i.e.

$$m \text{ consistent to } t \Leftrightarrow m \text{ consistent to } CR.$$

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

$$m \text{ consistent to } \mathbb{t} \Leftrightarrow \forall t \in \mathbb{t} : m \text{ consistent to } t.$$

Although Definition 4.7 precisely defines when we consider a transformation to be correct, it is unclear how to define a transformation that fulfills such a correctness property. In particular, most existing languages for specifying transformation are restricted to input changes to one model or at least to delivering changes to one model. We will thus discuss how we can achieve a correct synchronizing transformation by means of 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

Having multiple transformations between several metamodels requires their orchestration, i.e., the decision which transformation have to be executed in which order, if consistency between instances of those metamodels shall be preserved after changes. We discussed that transformations, or more precisely their consistency preservation rules, may be executed independently and thus concurrently, which requires their results to be unified, or to execute them consecutively. We already identified the drawbacks of a concurrent execution approach, including the necessity to define unification operators and the missing guarantee to be consistent to any consistency relation after such a unification. This is why we follow the approach of consecutively executing transformations.

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

Multiple transformations need orchestration

Orchestration function to determine execution order

Definition 4.10 (Transformation Orchestration Function)

Let \mathbf{t} be a set of transformations with consistency relations and according consistency preservation rules for metamodels \mathfrak{M} . A transformation orchestration function $\text{ORC}_{\mathbf{t}}$ for these transformations is a function that delivers a sequence of transformations for given models and changes:

$$\text{ORC}_{\mathbf{t}} : (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \rightarrow \mathbf{t}^{<\mathbb{N}}$$

$\mathbf{t}^{<\mathbb{N}}$ denotes all finite sequences in \mathbf{t} , i.e., $\mathbf{t}^{<\mathbb{N}} := \emptyset \cup \mathbf{t}^1 \cup \mathbf{t}^2 \cup \dots$

Repetitions in orchestration

According to this definition, the orchestration functions 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.

It is obvious that without further restrictions to the transformations it is possible that, for given transformations, models and changes to them, an

Existence of reasonable orchestrations

orchestration function cannot find an execution order that returns a consistent tuple of models after certain changes. This can be because there is no such order, because the transformations make local decisions to restore consistency for two models that are in no case consistent with the other transformations. Additionally, even if such an order exists, it may not be possible to find it.

We will discuss these problems in detail in Chapter 7 and investigate whether we can find restrictions for the transformations that ensure that an orchestration that delivers a consistent result always exists. We will also prove that without further restrictions the decision problem whether an orchestration exists that leads to a consistent result is undecidable. Due to these degrees of freedom, the definition does not further restrict that an orchestration of transformations has to lead to a consistent result.

An orchestration function does only determine an order of transformations. Intuitively speaking, consistency for given models and changes to them can be preserved by requesting an orchestration from that function and executing the transformations in the given order. We make this process explicit by defining an *application function* that is able to perform 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 4.4 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 tuples of metamodels and a conforming signature for their input and output, which eases the description of their concatenation.

Decidability
of existence
of orches-
tration

Explicit
function for
transfor-
mation
appli-
cation

Generaliza-
tion
function for
transfor-
mation
concatena-
tion

Definition 4.11 (Transformation Generalization Function)

Let $t = \langle CR, \text{CPR}_{CR} \rangle$ be a transformation for metamodels M_i, M_k . Let $\mathfrak{M} = \langle M_1, \dots, M_i, \dots, M_k, \dots, M_n \rangle$ be a tuple of metamodels containing M_i and M_k . A transformation generalization function $\text{GEN}_{\mathfrak{M}, t}$ for metamodels \mathfrak{M} and transformation t is a partial function:

$$\text{GEN}_{\mathfrak{M}, t} : (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \rightarrow (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \cup \{\perp\}$$

It generalizes the consistency preservation rule CPR_{CR} of 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 CPR_{CR} for the corresponding models to the given change tuple. Let $m \in I_{\mathfrak{M}}$ be a model tuple and let $\delta_{\mathfrak{M}} = \langle \delta_{M_1}, \dots, \delta_{M_i}, \dots, \delta_{M_k}, \dots, \delta_{M_n} \rangle$ be a change tuple. We define $\langle \delta'_{M_i}, \delta'_{M_k} \rangle := \text{CPR}_{CR}(m_i, m_k, \delta_{M_i}, \delta_{M_k})$. Then the generalization function is defined as:

$$\text{GEN}_{\mathfrak{M}, t}(m, \delta_{\mathfrak{M}})$$

$$= \begin{cases} \perp, & \langle \delta'_{M_i}, \delta'_{M_k} \rangle = \perp \\ (m, \langle \delta_{M_1}, \dots, \delta'_{M_i}, \dots, \delta'_{M_k}, \dots, \delta_{M_n} \rangle), & \text{otherwise} \end{cases}$$

Like consistency preservation rules, a generalization function must also be partial 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 \perp in exactly those cases in which the consistency preservation rule is generalized does so.

The generalization function is a universally-defined auxiliary function that is only necessary to formalize the concepts. It must neither be defined individually for a specific transformation, nor must it be explicitly specified by a developer of transformations at all.

Finally, either the orchestration function or an application function must also be able to reflect the cases in which no execution order of transformation can be found that restores consistency. In accordance to the terminology of Stevens [Ste20], we call those cases *unresolvable*. From a theoretical perspective, it does not make a difference whether the orchestration or the application function makes that decision. Finally, the orchestration function

Partial generalization function

Generalization function is universal

Dealing with unsolvable cases

could also directly be encoded into the application function from a theoretical perspective. However, from a practical perspective we may want to be able to find an execution order although there is no order that results in a consistent state, to be able to find out why it is not possible to restore consistency, thus, 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 \perp . We indicate those cases with the result \perp .

Definition 4.12 (Transformation Application Function)

Let \mathbb{t} be a set of synchronizing transformations for consistency relations \mathbb{CR} on metamodels $\mathfrak{M} = \langle M_1, \dots, M_n \rangle$ and $\text{ORC}_{\mathbb{t}}$ an orchestration function for them. A transformation application function $\text{APP}_{\text{ORC}_{\mathbb{t}}}$ for these rules is a partial function:

$$\text{APP}_{\text{ORC}_{\mathbb{t}}} : (I_{\mathfrak{M}}, \Delta_{\mathfrak{M}}) \rightarrow I_{\mathfrak{M}} \cup \{\perp\}$$

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 in \mathbb{CPR} . Thus, it has to fulfill the following condition:

$$\begin{aligned} & \forall m \in \{m \in I_{\mathfrak{M}} \mid m \text{ consistent to } \mathbb{CR}\} : \forall \delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \\ & (\exists m' \in I_{\mathfrak{M}} : \text{APP}_{\text{ORC}_{\mathbb{t}}}(m, \delta_{\mathfrak{M}}) = m' \Rightarrow \\ & \quad \exists t_1, \dots, t_m \in \mathbb{t} : \exists \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \text{ORC}_{\mathbb{t}}(m, \delta_{\mathfrak{M}}) = [t_1, \dots, t_m] \\ & \quad \wedge \text{GEN}_{\mathfrak{M}, t_1} \circ \dots \circ \text{GEN}_{\mathfrak{M}, t_m}(m, \delta_{\mathfrak{M}}) = (m, \delta'_{\mathfrak{M}}) \wedge \delta'_{\mathfrak{M}}(m) = m') \end{aligned}$$

While the previous definition does not restrict in which cases \perp 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 is, however, generally

impossible, because that decision problem is undecidable. In consequence, the orchestration function and application function need to operate conservatively, i.e., return \perp although there might be 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 4.13 (Transformation Application Function Correctness)

Let APP_{ORC_t} be an application function for an orchestration function ORC_t for transformations t . Let \mathbb{CR} be the set of consistency relations for which the transformations in t are defined. We say that APP_{ORC_t} is *correct* if, and only if, its result is either \perp or consistent to \mathbb{CR} for all inputs:

$$\begin{aligned} APP_{ORC_t} \text{ correct} &:\Leftrightarrow \forall m \in I_M : \forall \delta_M \in \Delta_M : m \text{ consistent to } \mathbb{CR} \\ &\Rightarrow APP_{ORC_t}(m, \delta_M) = \perp \vee APP_{ORC_t}(m, \delta_M) \text{ consistent to } \mathbb{CR} \end{aligned}$$

This, in fact, is a rather weak notion of correctness. Actually, an application function that always returns \perp is correct according to that definition. Due to the fact that the orchestration and application function have to operate conservatively, a binary correctness notion is not that relevant anyway. Rather the degree of conservativeness, i.e., how often it returns no result although one exists, and how to improve it is of special interest. The question how such an orchestration can or should look like was introduced as **RQ 1.4** and the degrees of freedom as well as a concrete approach will be presented as contribution **C 1.4** in Chapter 7.

Conservativeness
more
relevant

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 is 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 4.14 (Transformation Network)

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

$$\mathfrak{N} = \langle \mathbb{t}, \text{ORC}_{\mathbb{t}}, \text{APP}_{\text{ORC}_{\mathbb{t}}} \rangle$$

Correctness of a transformation network is given by correctness of both the individual transformations as well as the application function, according to Definition 4.7 and Definition 4.13. We say that the transformations ensure *local consistency*, because they are able to achieve consistency locally for two models, whereas the application function achieves *global consistency* by applying the individual transformations in a ways such that all models are consistent according to all transformations afterwards.

Definition 4.15 (Transformation Network Correctness)

Let $\mathfrak{N} = \langle \mathbb{t}, \text{ORC}_{\mathbb{t}}, \text{APP}_{\text{ORC}_{\mathbb{t}}} \rangle$ be a transformation network. We say that \mathfrak{N} is *correct* if, and only if, its transformations in \mathbb{t} as well as the application function $\text{APP}_{\text{ORC}_{\mathbb{t}}}$ is correct:

$$\mathfrak{N} \text{ correct} : \Leftrightarrow \forall t \in \mathbb{t} : t \text{ correct} \wedge \text{APP}_{\text{ORC}_{\mathbb{t}}} \text{ correct}$$

We have already indicated 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 already suggested that the consistency relations of the transformations are considered correct by definition, as there is no other specification to which they have to be correct, but 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 yet given a common definition of consistency [Ste10] by enumerating consistent pairs of models in a relation (see Section 4.1). 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 when architectural components and object-oriented classes are consistent and interfaces in two models are consistent, rather than considering when the overall models containing all these elements are consistent.

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

This reflects a natural understanding of consistency and especially makes it easier to make statements about dependencies between consistency relations,

Fine-grained understanding of consistency

Representation in transformation languages

Benefits of fine-grained notion

which we will 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 [Kla+20a]. They are based on the work of Kramer [Kra17, Section 2.3.2, 4.1.1].

4.4.1. Fine-Grained Consistency Relations

Conditions 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 4.16 (Condition)

A condition \mathbb{C} for a class tuple $\mathfrak{C}_{\mathbb{C}} = \langle C_{\mathbb{C},1}, \dots, C_{\mathbb{C},n} \rangle$ is a set of object tuples with:

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

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

$$m \text{ contains } c : \Leftrightarrow \exists m \in \mathfrak{M} : c \subseteq m$$

Models containing conditions *Conditions* represent object tuples that instantiate the same tuple of classes. They are supposed to occur in models that fulfill a certain condition regarding consistency, i.e., they define the objects that can occur in the previously mentioned pairs of consistency relations, which we specify later. 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 the

metamodel of such a model has to contain all the classes in the class tuple of the condition. We use these conditions to define consistency relations as the co-occurrence of condition elements.

Definition 4.17 (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 $\mathfrak{C}_{l,CR}, \mathfrak{C}_{r,CR}$ with $\mathfrak{C}_{l,CR} = \mathfrak{C}_{\mathfrak{C}_{l,CR}}$ and $\mathfrak{C}_{r,CR} = \mathfrak{C}_{\mathfrak{C}_{r,CR}}$:

$$CR \subseteq \mathfrak{C}_{l,CR} \times \mathfrak{C}_{r,CR}$$

We call a pair of condition elements $\langle c_l, c_r \rangle \in CR$ a *consistency relation pair*. For a set of models m and a consistency relation pair $\langle c_l, c_r \rangle$, we say that:

$$m \text{ contains } \langle c_l, c_r \rangle : \Leftrightarrow m \text{ contains } c_l \wedge m \text{ contains } c_r$$

Without loss of generality, we assume that each condition element of both conditions occurs in at least one consistency relation pair:

$$\forall c \in \mathfrak{C}_l : \exists \langle c_l, c_r \rangle \in CR : c = c_l \wedge \forall c \in \mathfrak{C}_r : \exists \langle c_l, c_r \rangle \in CR : c = c_r$$

A consistency relation according to Definition 4.17 is a set of pairs of object tuples, which are supposed to indicate the tuples of objects that are considered consistent with each other, i.e., if a model contains one of the left object tuples occurs in the relation one of the related right object tuples has to occur in a model as well. It is based on two conditions that define relevant object tuples in each of the two metamodels and defines the ones that are related to each other. Based on these consistency relations, we can define a fine-grained notion of consistency.

Definition 4.18 (Consistency)

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

\mathbf{m} consistent to CR : \Leftrightarrow

$$\begin{aligned} \exists W \subseteq CR : & (\forall \langle c_{l,1}, c_{r,1} \rangle, \langle c_{l,2}, c_{r,2} \rangle \in W : \\ & \langle c_{l,1}, c_{r,1} \rangle = \langle c_{l,2}, c_{r,2} \rangle \vee (c_{l,1} \neq c_{l,2} \wedge c_{r,1} \neq c_{r,2})) \\ & \wedge \forall \langle c_l, c_r \rangle \in W : \mathbf{m} \text{ contains } c_l \wedge \mathbf{m} \text{ contains } c_r \\ & \wedge \forall c'_l \in \mathbb{C}_{l,CR} : \mathbf{m} \text{ contains } c'_l \Rightarrow c'_l \in \mathbb{C}_{l,W} \end{aligned}$$

We call such a W a *witness structure* for consistency of \mathbf{m} to CR , and for all elements $\langle w_l, w_r \rangle \in W$, we call w_l and w_r *corresponding* to each other.

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

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

Consistency
by fine-
grained
relations

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

Witness
structure

Consider the exemplary consistency relations 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 in lower case 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

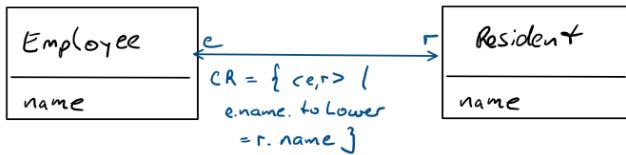


Figure 4.8.: A consistency relation derived from Figure 3.3, which depicts the necessity of a witness structure to ensure that only one employee out of those with differently capitalized names is allowed to correspond to a resident with the same name.

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 lower case. 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 lower case 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 alteration 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 same 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 = "John". However, our definition of consistency only requires that for each condition element of 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 will discuss in more detail in the following. 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 element in a model for which consistency is defined by a consistency relation may only have one corresponding

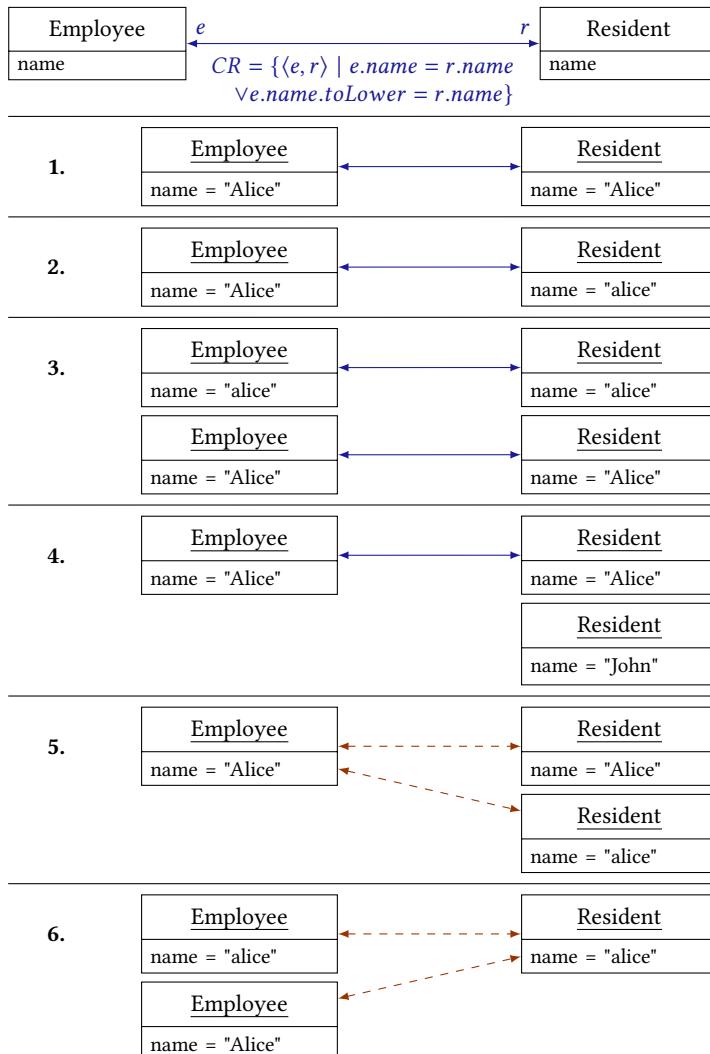


Figure 4.9.: A consistency relation between employee and resident and six example model pairs: model pairs 1–4 being consistent with an appropriate witness structure W shown in blue and model pair 5 and 6 being inconsistent with an inappropriate mapping structure shown in red and dashed.

element in the model. In this case, there are two residents respectively two employees that could be considered consistent to the employee respectively resident, thus there is no appropriate witness structure with a unique mapping between the elements as required by the consistency definition.

As mentioned before, we define the notion of consistency in a unidirectional way, which means that a consistency relation may define that some elements c_r are required to occur in a tuple of models if some elements 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 also be an employee. To achieve a bijective consistency definition, for each consistency relation CR its transposed relation $CR^T = \{\langle c_l, c_r \rangle \mid \langle c_r, c_l \rangle \in CR\}$ 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 would be required by the transposed relation. We call sets of consistency relations that contain only bijective definitions of consistency *symmetric*.

Unidirectional notion

Definition 4.19 (Symmetric Consistency Relation Set)

Let \mathbb{CR} be a set of consistency relations. We say that \mathbb{CR} is *symmetric* if for each contained relation its transposed one is also contained:

$$\mathbb{CR} \text{ is symmetric } :\Leftrightarrow$$

$$\forall CR \in \mathbb{CR} : \exists CR' \in \mathbb{CR} : CR' = CR^T$$

Any description of bijective consistency relations can be achieved by defining a symmetric set of consistency relations. We chose to define consistency in a unidirectional way due to two reasons:

Reasons for unidirectionality

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 bidirectional for determining consistency.

2. We consider networks of consistency relations, in which, as we will see later, 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.

Explicit
trace
models

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 4.18. We do, however, not explicitly consider such an explicit trace model in this formalism due to two reasons [Kla+20b]. 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 assumed such identifying information 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, thus always defining consistency between one concrete and one correspondence 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 [Ste20; Cle+19]. We will later discuss practical benefits of having an explicit trace model for consistency preservation 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

Expressiveness
of
fine-grained
consistency

The model-level consistency notion of Definition 4.2 is established and based on notions used by several researchers. The given fine-grained notion of consistency according to Definition 4.18 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 correspondences, but we did not argue if and why fine-grained relations are an actual refinement, such that statements about model-level consistency relation used for our general formal framework also apply to fine-grained consistency 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 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:

$$CR = \{m \mid m \text{ consistent to } \mathbb{CR}\}$$

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 a left-hand side element occurs in a model, then the right-hand side has to occur in another. If two models are considered consistent by a model-level consistency relation, they will also be consistent to the accordingly constructed fine-grained relation, because there is a witness structure that contains exactly the two consistent models. If there is, however, 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 consistency relation. According to the definition of consistency for fine-grained relations (see Definition 4.18), 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 is actually 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 other, e.g., being an employee in one model, the same person cannot be a student in another model, is actually not a

Implication
of
model-level
relation

Model-level
relations
more
expressive

Additional
semantics
in
consistency
relations

consistency constraint for information shared between models, but 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 can not prevent specific elements from existing at all. For example, a consistency relation may define that for a component in an architecture model there has to be a corresponding class in the object-oriented design model, 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 additional restrictions of valid models, which we have already discussed as a restriction of I_M for a metamodel M in Section 3.3.

Summarizing, we know 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, which cannot be expressed with 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 as well.

4.4.3. Application to Consistency Preservation Rules

As mentioned before, the fine-grained notion of consistency does also fit well to how specifications in transformation languages consider consistency. They allow to define rules that relate only some classes by relations, conforming to fine-grained consistency relations, from which then 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 for it.

It may easily happen that the execution of one transformation rule leads to the violation of the consistency relation of another one, which introduces de-

Restriction
of valid
models

Insight
transferabil-
ity
between
notions

Fine-
grained
notion in
transforma-
tion
languages

Conflicts
between
transfor-
mation
rules

pendencies between the individual transformation rules. Thus, a combination of such 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 (cf. [Obj16a, p. 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], thus we assume that a transformation has 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 yet defined for model-level consistency relations in Definition 4.4. This can, however, easily be 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 4.18 rather than consistency to a single model-level consistency relation according to Definition 4.2.

A consistency preservation rule CPR_{CR} for a set of consistency relations CR according to Definition 4.17 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 4.5:

$$\begin{aligned} & \forall m_1 \in I_{M_1}, m_2 \in I_{M_2}, \delta_{M_1} \in \Delta_{M_1}, \delta_{M_2} \in \Delta_{M_2} : \\ & \exists \delta'_{M_1} \in \Delta_{M_1}, \delta'_{M_2} \in \Delta_{M_2} : \langle \delta'_{M_1}, \delta'_{M_2} \rangle = \text{CPR}_{\text{CR}}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) \\ & \Rightarrow \langle \delta'_{M_1}(m_1), \delta'_{M_2}(m_2) \rangle \text{ consistent to CR} \end{aligned}$$

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

Likewise, we consider a synchronizing transformation according to Definition 4.6 as a pair of fine-grained consistency relations and a consistency preservation rules for them, thus $t = \langle \text{CR}, \text{CPR}_{\text{CR}} \rangle$. Again, in conformance with Definition 4.7, we call such a transformation t correct if, and only if, its consistency preservation rule is correct.

Extension of
Consistency
preservation
rules

Consistency
preserva-
tion for
fine-
grained
relations

Transforma-
tions for
fine-
grained
relations

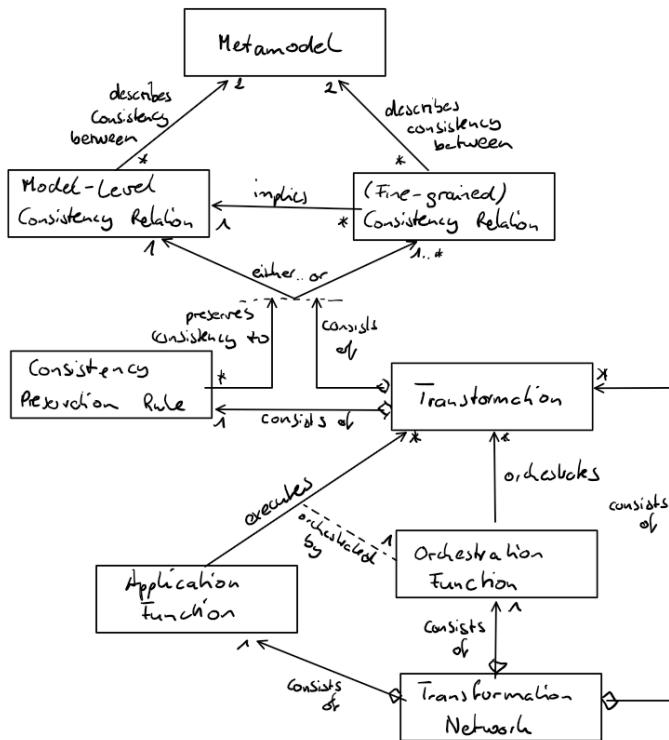


Figure 4.10.: A conceptual model for the terms and artifacts introduced for transformation networks and their relations. Adapted from [Kla+20b].

4.5. Summary

Insight

In this chapter, we have discussed notions of correctness for transformation networks and the artifacts it consists of, and we have precisely defined that 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 depicted in Figure 4.10. In summary, we provided the following insight in this chapter.

Insight 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 due to 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 have to fulfill some notion of compatibility to be useful, as they can otherwise prevent transformations from finding consistent models.

In the following chapters, we will 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 will 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 in blue with 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. We depict consistency preservation rules in orange 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 $r(name = e.name)$. In addition, we annotate conditions to the

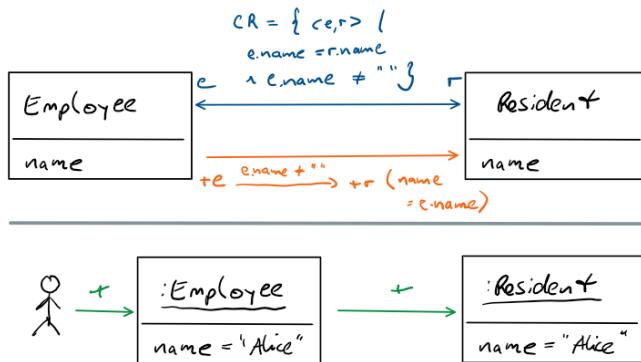


Figure 4.11.: Example for visualization of consistency relations, consistency preservation rules and their execution.

consistency preservation rules, such as $e.name \neq \text{enquote}$ in the example, which restricts the resident creation to the case in which the employee has a non-empty name. We usually specify only parts of a consistency preservation rule, e.g., 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, in green. In the example, we visualize the addition of an employee by a user, denoted with a +, which leads to the addition of a resident, for example, because of the execution of the above introduced consistency preservation rule.

Consistency relations correct by construction

5. Proving Compatibility of Consistency Relations

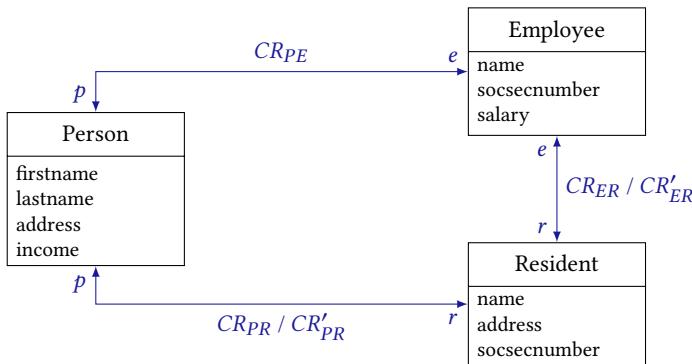
We have defined in Chapter 4 that transformations, from which we construct transformation networks, are composed of consistency relations and consistency preservation rules that preserve them. We focus on binary relations and according preservation rules, which relate two metamodels. While we precisely 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 implicitly given) 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 (cf. 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 the 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.

Contradictions in relations

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 that example, the consistency relations CR_{PE} , CR_{PR} and CR_{ER} are fulfilled if for each person (and each employee and resident analogously) in the models

Intuitive compatibility in running example



$$\begin{aligned}
 CR_{PE} &= \{(p, e) \mid p.firstname + " " + p.lastname = e.name \wedge p.income = e.salary\} \\
 CR_{PR} &= \{(p, r) \mid p.firstname + " " + p.lastname = r.name \wedge p.address = r.address\} \\
 CR'_{PR} &= \{(p, r) \mid p.lastname + " " + p.firstname = r.name \wedge p.address = r.address\} \\
 CR_{ER} &= \{(e, r) \mid e.name = r.name \wedge e.socsecnumber = r.socsecnumber\} \\
 CR'_{ER} &= \{(e, r) \mid e.name.toLowerCase = r.name \wedge e.socsecnumber = r.socsecnumber\}
 \end{aligned}$$

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+20a, Fig. 1].

there is exactly one employee and one resident that fulfill the relations for names and further attributes defined by the consistency relations. According to our notion of consistency relations defined in Definition 4.17, it is of special importance that there is always only one such corresponding element, e.g., that there are not two employees with different name capitalizations fulfilling the relation to a single resident. Intuitively, these consistency relations are *compatible*, as they lead to a reasonable set of model tuples that are considered consistent to each other.

In contrast, considering consistency relation CR'_{PR} instead of CR_{PR} , the relations can never be fulfilled, because the concatenation of `firstname` and `lastname` from person to employee and from person to resident is conflicting. The relation between employees and persons assumes `firstname` and `lastname` to be concatenated in the same order, whereas the relation between residents

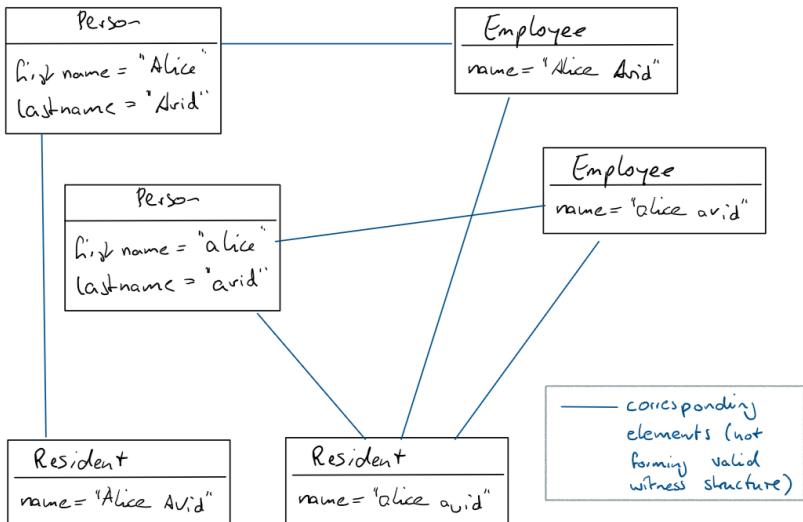


Figure 5.2.: Elements required by the consistency relations in Figure 5.1 for a resident with the name “Alice Avid”.

and persons assumes them to be concatenated vice versa and 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 others with *firstname* and *lastname* swapped and prolonged with a comma to exist. As this cannot be fulfilled with finite models, 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 lower case 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-lower case name requires a person with equally capitalized first and last name to exist. This, in consequence requires an employee with an equally capitalized name to exist. The relation CR'_{ER} now requires a resident with the name written in lower case to exist, which, again, requires a person with the lower case name to exist. This, in turn, requires an employee with the lower case name to exist as well. In consequence, the resident with the lower case name would correspond to both the employee with the origi-

Further incompatibility in modified running example

nal and the lower case 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-lower case name. This is what we will 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 combined consistency relations.

Such incompatibilities are unwanted, as they indicate that developers have different, contradictory notions of consistency. Additionally, the contradictions can easily lead to the situation that the transformations are not able to find consistent models or at least that their orchestration for finding consistency models becomes unnecessarily difficult. Therefore, in this chapter we first discuss some scenarios to identify an intuitive notion of compatibility, which we then use to define a precise notion of *compatibility*. Afterwards, we develop a formal, inductive approach to prove compatibility of relations, which we base on a formal framework for which we prove correctness. We then derive a practical approach for the transformation language QVT-R that uses that formal framework. The approach is based on the insight that consistency relations having a specific kind of tree structure are compatible and that removing a specific kind of redundant relations is compatibility-preserving. 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 notions; 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 transformation are incompatible if the language, in which the relations are described, is undecidable, such as QVT-R. We can, however, at least conservatively prove that transformations are compatible. 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 both on-the-fly during transformation development, if developed

Incompatibilities affect consistency preservation

Provability of compatibility

for a specific scenario, or a posteriori during their combination, according to the scenarios introduced in Section 3.2. Especially in the first scenario, developers can immediately react to the introduction of incompatibilities during transformation development.

We have published the central contributions of this chapter, including the formal and practical approach for validating compatibility, in [Kla+20a]. Parts of some sections of this chapter are also literally taken from that publication, which we will 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.

Publication
of contribu-
tions

5.1. Towards a Notion of Compatibility

We start with general considerations on model-level consistency relations, be they 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, \dots, M_n and pairwise consistency relations $CR_{i,j}$ is defined by:

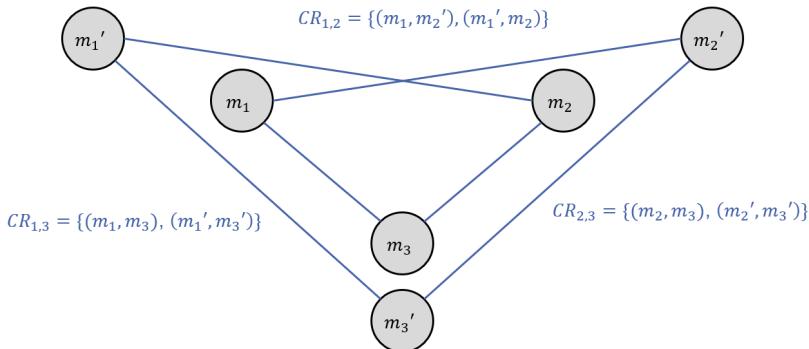
$$CR = \{ \langle m_1, \dots, m_n \rangle \mid \bigwedge_{1 \leq i < j \leq n} \langle m_i, m_j \rangle \in CR_{i,j} \}$$

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.

Modular
relations
induce
monolithic
ones

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. We would thus consider such relations incompatible. Figure 5.3 shows an extended version of the example already given in Subsection 4.2.4, inducing an empty global relation. This is an abstraction of the concrete examples, which we have already discussed for our running example, in which modified consistency relations lead to an empty set of consistent model tuples, due to conflicting conversions and concatenations of names between persons, residents and employees.

Empty
induced
global
relations

**Figure 5.3.:** Example for consistency relations that imply an empty global relation.

Goal of identifying incompatible relations

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 may want to identify that 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 transformation are executed would produce one new person, employee, and resident, or change each of the existing ones, such that *firstname* and *lastname* are swapped and a comma 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 it can prevent the execution of transformations in a transformation network from termination.

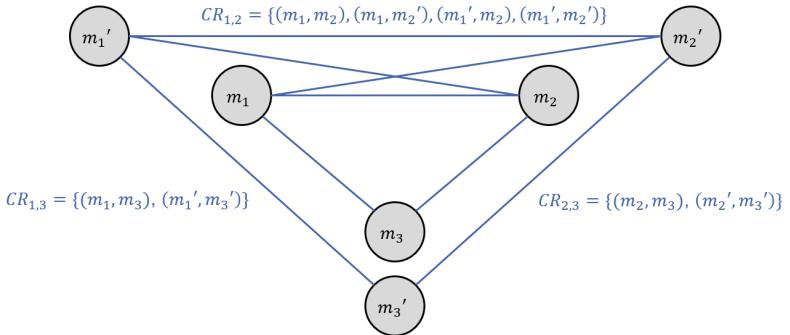


Figure 5.4.: Example for obsolete model pairs in consistency relation $CR_{1,2}$, which can never occur in a globally consistent tuple of models.

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 that 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.

Obsolete
model pairs

While this point of view may be reasonable for the consistency relations only, as we are finally only 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, which 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 develop-

Forbidding
obsolete
model pairs

5. Proving Compatibility of Consistency Relations

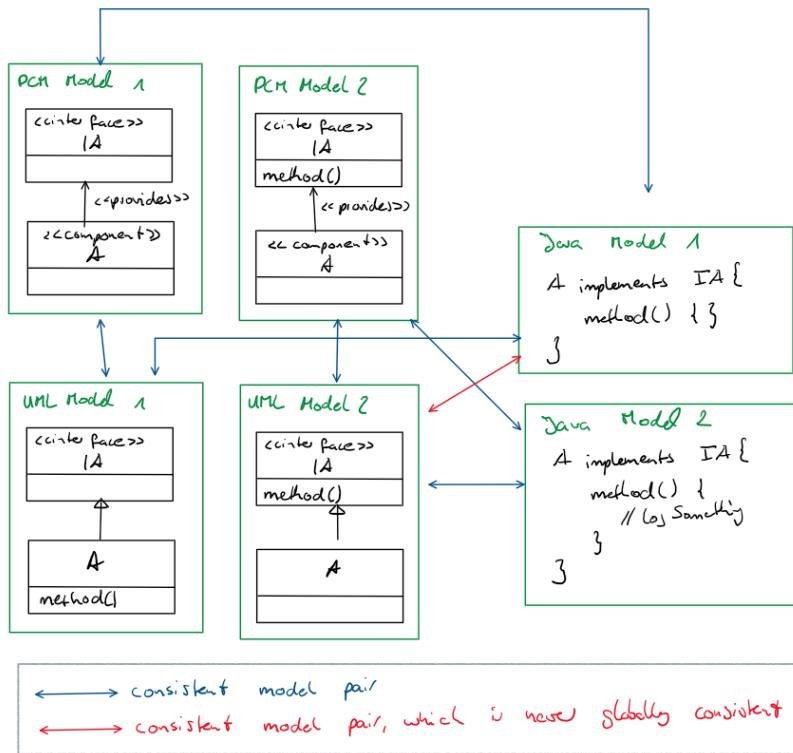


Figure 5.5.: Example for an obsolete model pair in consistency relations between PCM, UML and Java: The empty Java method realization is locally consistent to the UML class model that defines the method in the class interface, but cannot be globally consistent because it realizes a PCM component, for which the consistency relation requires at least a default implementation.

ment 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. In fact, for each UML model a usually infinite number of consistent Java models exists, containing arbitrary implementations of the methods. In addition, PCM models and UML class models

are consistent when components are realized as classes implementing the provided interfaces of the components and thus their methods. Analogously, each component is represented by a Java class implementing the provided interface. 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 has at least some default implementation, be it logging or something more component-specific. If we would now consider 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 that realizes a component and that is defined by 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 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 that, and of modular reuse, because the transformation is then tied to the scenario in which 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.

No proper
notion of
incompati-
bility

5.1.2. Prevention from Finding Consistent Solutions

To identify a proper notion of incompatibility, we now 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 consistency relations. If a user performs a change of m_2 to m'_2 , we may assume the following transformation executions: 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_{1m_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 transformation network executed the transformations in a way such that the original models are returned, which are actually consistent but reject the user change.

Reverting
user
changes

5. Proving Compatibility of Consistency Relations

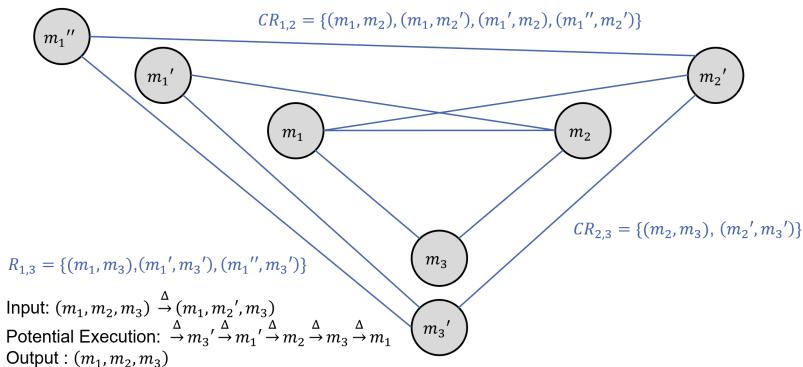


Figure 5.6.: Example for the rejection of a user change because of consistency relations containing model pairs that are never globally consistent.

Bad local
selections

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 actually expect the transformations to find the latter triple of models as a consistent result, as otherwise, like in the above example, the original models are returned, which actually rejects the user change. The problem results from the model m_1' being present in the consistency relations, but not being consistent in any globally consistent model tuple. For each of the transformations the local selection of m_1' is fine, as there are models to which it is locally consistent according to each consistency relation on its own.

Difference
to obsolete
elements

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 defined in the class interface, although they are locally consistent.

Relation to
user
changes

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 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 which reflects 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. 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 pair of a consistency relation 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 was such a model, as it was locally consistent to m_2 and m'_3 , but those two are inconsistent.

We can distinguish 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, so the transformation should better select that one. If there was no such m''_1 , then m_2 and m'_3 would be in the same situation not occurring 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

Property of
problem-
atic
models

Cases
introducing
problem-
atic
models

more reasonable when not concerning the complete model, like proposed with authoritative models by Stevens [Ste20], but only the part considered by a specific rule that describes consistency, such as a rule specifying the relation between classes and components, rather than between the complete metamodels of PCM and UML. This is one of the reasons why we provided the formalization of fine-grained consistency relations in Definition 4.17 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 occurs in no 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 pair of a consistency relation, 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. If a user performs a change, we want to ensure that if consistency of the modified elements is restricted by a consistency relation, there should be at least 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 led 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 which contain each person twice, once with its proper name and once with swapped first and last name. Although we

Notion of compatibility

Reasonability for preservation process

Reflection of semantic contradictions

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 it is generally necessary to not disallow that case.

Obviously, the given notion of compatibility is a property of a set of consistency relations and not of a single consistency relation on its own. We may also 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.

Context dependency

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 an overview of that approach, we first briefly introduce the central idea based on the given informal notion of compatibility, which we first introduced in [Kla18] and [Kla+19].

Compatibility analysis overview

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.

Compatibility of trees

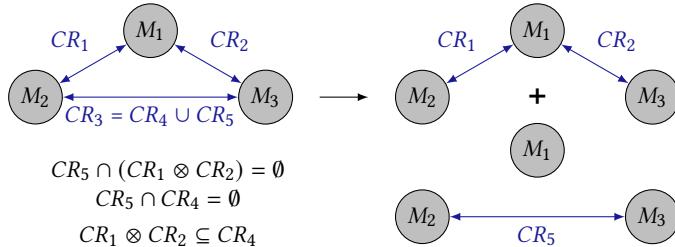


Figure 5.7.: Example for the decomposition of independent and removal of redundant consistency relations for analyzing compatibility. Adapted from [Kla18].

Compatibility-preserving properties

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

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, virtually, 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 applying the virtual removal of redundant relations 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: Independence of fine-grained consistency relations is a second compatibility-preserving property. 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 independent, because modifications in components and classes do never 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 have consistency relations CR_1, CR_2, CR_3 between three metamodels. 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 virtually remove CR_4 and consider CR_1 and CR_2 independently from CR_5 , inductively due to its independence from CR_4 . The result are two independent trees of relations, which are inherently compatible. Since redundancy and independence are compatibility-preserving, this proves that the original relations are compatible.

Example for compatibility analysis

5.2. A Formal Notion of Compatibility

In this section, we precisely define our, yet informally introduced notion of *compatibility*. For that, we use the fine-grained notion of consistency and 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, even literally, taken from our published article on proving compatibility [Kla+20a].

Formal notion of fine-grained consistency and compatibility

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. However, such consistency relations imply 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 , for which we define a notion for the concatenation of relations. The goal of this notion is to provide a relation that is induced by the concatenated

ones. This means, if a model is consistent to the concatenation, it should also be consistent to the single relations, as otherwise the concatenation would introduce additional consistency constraints. To achieve this, the following definition makes appropriate restrictions to the derived consistency relation pairs.

Definition 5.1 (Consistency Relations Concatenation)

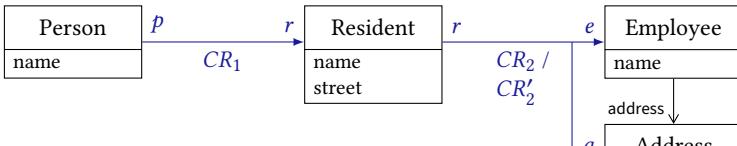
Let CR_1, CR_2 be two consistency relations. The concatenation is defined as follows:

$$\begin{aligned} CR_1 \otimes CR_2 := & \{ \langle c_l, c_r \rangle \mid \\ & \exists \langle c_l, c_{r,1} \rangle \in CR_1 : \exists \langle c_{l,2}, c_r \rangle \in CR_2 : c_{r,1} \subseteq c_{l,2} \\ & \wedge \forall \langle c_l, c'_{r,1} \rangle \in CR_1 : \exists \langle c'_{l,2}, c'_{r,2} \rangle \in CR_2 : c'_{l,2} \subseteq c'_{r,1} \} \end{aligned}$$

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 all 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 concatenated relation are also consistent to the single relations: First, it is important that a pair of consistency relations CR_1, CR_2 is only combined if the left condition element of the consistency relation pair from CR_2 is a subset of the right condition element of the consistency relation pair $\langle c_l, c_r \rangle$ of CR_1 . Second, it is necessary that for all elements c_r in the right side of CR_1 , to which a condition element c_l is considered consistent, there must be a matching condition element, i.e. a subset of c_r , in the left condition of CR_2 . Otherwise, in both cases the occurrence of c_l in a model tuple would not necessarily impose any consistency requirement by CR_2 . In the following, we explain these two requirements at an example.

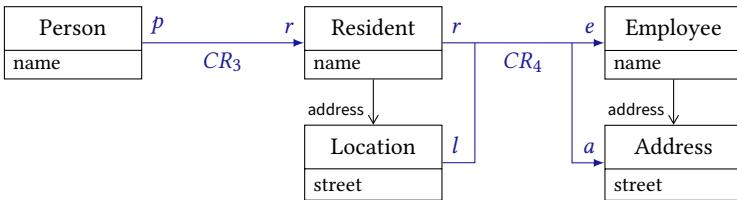
Example 5.1. Figure 5.8 extends the initial example (Figure 5.1) with further classes in the consistency relations, such that they do not only relate single classes to each other. It defines an address for employees and in the second example also a location for the address of residents, which are represented in additional classes. Both examples contain a consistency relation CR_1 respectively CR_3 between persons and residents, which define that for each person a resident



$$CR_1 = \{\langle p, r \rangle \mid p.name = r.name\}$$

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

$$CR'_2 = \{\langle r, (e, a) \rangle \mid \langle r, (e, a) \rangle \in CR_2 \wedge r.street \neq ''\}$$



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

$$CR_4 = \{\langle (r, l), (e, a) \rangle \mid r.name = e.name \wedge l.street = a.street\}$$

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+20a].

with the same name has to exist. The examples provide different options for consistency relation between residents (with locations) and employees with addresses (CR_2, CR'_2, 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 an arbitrary street name. In consequence, $CR_1 \otimes CR_2$ defines a relation for each person with an employee having the same name and all addresses with 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 address it is not required that an employee exists. This results in $CR_1 \otimes CR'_2 = \emptyset$, because for any person, there must not be an employee, as the person can be consistent to a resident with an empty street name. 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 must not necessarily 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. $CR_4^T \otimes CR_3^T$: The concatenation of the transposed relations $CR_4^T \otimes CR_3^T$ is not empty, but actually 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 due to its first restriction, because $c_{l,2}$, i.e., the resident related to a person by CR_3^T is a subset of $c_{r,1}$, 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, CR_2 be two consistency relations and let $CR = CR_1 \otimes CR_2$ be their concatenation. For all model tuples $m \in I_M$ the following statement holds:

$$m \text{ consistent to } \{CR_1, CR_2\} \Rightarrow m \text{ consistent to } CR$$

Proof. For any tuple of models m that is consistent to CR_1 and CR_2 , take the witness structure W_1 that witnesses consistency of m to CR_1 and W_2 that witnesses consistency of m to CR_2 . Now consider the composed witness

structure $W = W_1 \otimes W_2$. Let us assume there were $\langle c_l, c_r \rangle, \langle c'_l, c'_r \rangle \in W$ with $c_l = c'_l$ and $c_r \neq c'_r$. Per definition c_l only occurs in one $\langle c_l, c_{r,1} \rangle \in W_1$. So there must be two $\langle c_{l,2}, c_r \rangle, \langle c'_{l,2}, c'_r \rangle \in CR_2$ with $c_{l,2} \subseteq c_{r,1}$ and $c'_{l,2} \subseteq c_{r,1}$. However, since $c_{l,2}$ and $c'_{l,2}$ contain instances of the same classes and are both subsets of the same other object tuples $c_{r,1}$, we have $c_{l,2} = c'_{l,2}$. So we know that:

$$\begin{aligned} \forall \langle c_{l,1}, c_{r,1} \rangle, \langle c_{l,2}, c_{r,2} \rangle \in W : \\ \langle c_{l,1}, c_{r,1} \rangle = \langle c_{l,2}, c_{r,2} \rangle \vee c_{l,1} \neq c_{l,2} \wedge c_{r,1} \neq c_{l,2} \end{aligned}$$

Additionally, since W_1 and W_2 are witness structures for consistency of m to CR_1 and CR_2 , the model tuple contains all condition elements in W_1 and W_2 . Consequentially, 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:

$$\forall \langle c_l, c_r \rangle \in W : m \text{ contains } c_l \wedge m \text{ contains } c_r$$

Finally, let us assume that:

$$\exists c'_l \in \mathbb{C}_{l,CR} : m \text{ contains } c'_l \wedge c'_l \notin \mathbb{C}_{l,W}$$

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

$$\forall c'_l \in \mathbb{C}_{l,CR} \mid m \text{ contains } c'_l : c'_l \in \mathbb{C}_{l,W}$$

Summarizing, we have shown that W fulfills all requirements to a witness structure according to Definition 4.18 for m being consistent to CR , so we know that m *consistent to* CR . \square

5.2.2. Transitive Closure of Consistency Relations

We can use this notion of concatenation to define a transitive closure for sets of consistency relations, 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 consistency relations concatenation is well-defined in the sense that it does not introduce further restrictions for consistency, we are also able to show that the transitive closure does not restrict consistency in comparison to the set of consistency relation 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 concatenations of relations in \mathbb{CR} . That means, the transitive closure contains consistency relations that relate all elements that are directly or indirectly related due to \mathbb{CR} .

The transitive closure of a consistency relation set does not further restrict consistency in comparison to the original set by construction of concatenation, 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 in the following 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:

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

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

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

We already know for any two consistency relations CR_1, CR_2 and all model tuples m that if m consistent to $\{CR_1, CR_2\}$, then m consistent to $CR_1 \otimes CR_2$ due to Lemma 5.1. Inductively applying that argument to CR_1, \dots, CR_k shows that for all models m with m consistent to $\{CR_1, \dots, CR_k\}$ we know that m consistent to CR . \square

As a direct result of the previous lemma, we can now 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 set of consistency relations for a tuple of metamodels \mathfrak{M} . Then for all sets of models $m \in I_{\mathfrak{M}}$ it is true that:

$$m \text{ consistent to } \mathbb{CR} \Leftrightarrow m \text{ consistent to } \mathbb{CR}^+$$

Proof. 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 4.18 for consistency, which requires models be consistent to all consistency relations in a set to be considered consistent, thus only restricting

the set of consistent model tuples by adding further consistency relations. In consequence, it holds that:

$$\mathbf{m} \text{ consistent to } \mathbb{C}\mathbb{R}^+ \Rightarrow \mathbf{m} \text{ consistent to } \mathbb{C}\mathbb{R}$$

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

$$\mathbf{m} \text{ consistent to } \mathbb{C}\mathbb{R} \Rightarrow \mathbf{m} \text{ consistent to } \mathbb{C}\mathbb{R}^+$$

In consequence, models are considered consistent equally for $\mathbb{C}\mathbb{R}$ and its transitive closure $\mathbb{C}\mathbb{R}^+$. \square

5.2.3. Compatibility of Consistency Relations

Based on the fine-grained notion of consistency in terms of consistency relations, we can now precisely formulate our initially informal notion of *compatibility* of consistency relations. We stated that we consider consistency relation incompatible if they are somehow 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. To capture that in a definition, we consider relations compatible if for all condition elements in the consistency relations, i.e., for every tuple of objects for which consistency is somehow constrained by requiring further elements to exist in a tuple of models to consider it consistent, a consistent model containing those objects can be found. In consequence, a consistency relation is not allowed to prevent objects for which other relations specify consistency from existing in consistent models.

Definition 5.3 (Compatibility)

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

$$\begin{aligned} \mathbb{CR} \text{ compatible } &:\Leftrightarrow \forall CR \in \mathbb{CR} : \forall c \in \mathbb{C}_{l, CR} : \exists m \in I_{\mathfrak{M}} : \\ &m \text{ contains } c \wedge m \text{ consistent to } \mathbb{CR} \end{aligned}$$

We call a set of consistency relation \mathbb{CR} *incompatible* if it does not fulfill the definition of compatibility.

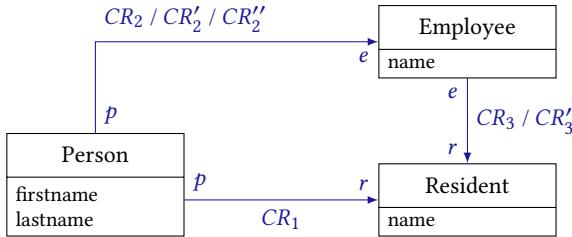
Definition 5.3 formalizes the notion of *non-contradictory* relations by requiring that a relation may not restrict that an object tuple, for which consistency is defined in any consistency relation, cannot occur in a model tuple anymore. We exemplify this notion of compatibility on an extract of the 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 respectively 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.

$\{CR_1, CR_1^T, CR'_2, CR'^T_2, CR_3, CR_3^T\}$: These relations are obviously not compatible, because for each person with $firstname = f$ and $lastname = l$, another person with $firstname = f + ;$ and $lastname = l$ has to exist 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. However, models are assumed to be finite, so there is no such model tuple and the relations are incompatible.

$\{CR_1, CR_1^T, CR'_2, CR'^T_2, CR_3, CR_3^T\}$: These relations are compatible, although one might not expect that. The relations define that for a resident with $firstname = f$ and $lastname = l$ another resident with $firstname = l$ and



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

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

$$CR'_2 = \{\langle p, e \rangle \mid e.name = p.firstname + " " + p.lastname\}$$

$$CR''_2 = \{\langle p, e \rangle \mid e.name = p.lastname + " " + p.firstname\}$$

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

$$CR'_3 = \{\langle r, e \rangle \mid r.name = e.name.toLowerCase\}$$

Figure 5.9.: Three metamodels with different options of consistency relations. The sets $\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3, CR_3^T\}$ and $\{CR_1, CR_1^T, CR_2^T, CR_2'', CR_2'', CR_3, CR_3^T\}$ 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+20a].

lastname = f has to exist, so that the tuple of models is consistent. Although that behavior may not be intuitive, it does not violate the definition of compatibility, because for any object in the relations, a consistent model can be constructed. In general, such a behavior cannot be forbidden, because comparable behavior might be expected, such as that for a software component an implementation class as well a utility class with different names are created due to different relations, which leads to comparable behavior as in the example. Finally, such a relation would not prevent a consistency repair routine from finding a consistent tuple of models. So this can be seen as a semantic problem that requires further relation-specific knowledge, as it is necessary to know that a first name should never be mapped to a last name in our example.

$\{CR_1, CR_1^T, CR_2, CR_2^T, CR_3, CR_3^T\}$: These consistency relations reflect the ones of our motivational example in Figure 5.1. According to the informal notion of incompatibility that we motivated in the introduction with that example,

our 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 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 resident 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 4.18 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 of consistency relations is especially to prevent consistency repair routines of model transformation from non-termination, as may occur especially in the second scenario, where an infinitely large model would be required to fulfill the consistency relations.

Finally, analogously to the equivalence of a set of consistency relations \mathbb{CR} and its transitive closure \mathbb{CR}^+ in regards to consistency of a tuple of models, 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 for a tuple of metamodels \mathfrak{M} . It holds that:

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

Proof. The reverse direction of the equivalence is given by definition, since compatibility of a sub of consistency relations implies compatibility of any 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 $CR \in \mathbb{C}\mathbb{R}^+ \setminus \mathbb{C}\mathbb{R}$. According to Definition 5.2 for the transitive closure, each $CR \in \mathbb{C}\mathbb{R}^+ \setminus \mathbb{C}\mathbb{R}$ is a concatenation of consistency relations $CR_1, \dots, CR_k \in \mathbb{C}\mathbb{R}$. In consequence of that definition we know that $\mathbb{C}_{I,CR} \subseteq \mathbb{C}_{I,CR_1}$, so it is given that:

$$\begin{aligned} \forall c_l \in \mathbb{C}_{I,CR} : \exists c'_l \in \mathbb{C}_{I,CR_1} : \forall m \in I_{\mathfrak{M}} : \\ m \text{ contains } c_l \Rightarrow m \text{ contains } c'_l \end{aligned} \quad (5.1)$$

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

$$\begin{aligned} \forall c'_l \in \mathbb{C}_{I,CR_1} : \exists m \in I_{\mathfrak{M}} : \\ m \text{ contains } c'_l \wedge m \text{ consistent to } \mathbb{C}\mathbb{R} \end{aligned} \quad (5.2)$$

Because of Equation 5.1 and Equation 5.2, we know that:

$$\begin{aligned} \forall c_l \in \mathbb{C}_{I,CR} : \exists m \in I_{\mathfrak{M}} : \\ m \text{ contains } c_l \wedge m \text{ consistent to } \mathbb{C}\mathbb{R} \end{aligned} \quad (5.3)$$

Furthermore, Lemma 5.3 states that for all model tuples $m \in I_{\mathfrak{M}}$ it is true that:

$$m \text{ consistent to } \mathbb{C}\mathbb{R} \Leftrightarrow m \text{ consistent to } \mathbb{C}\mathbb{R}^+ \quad (5.4)$$

In consequence of equations 5.3 and 5.4, we know that:

$$\begin{aligned} \forall CR \in \mathbb{C}\mathbb{R}^+ \setminus \mathbb{C}\mathbb{R} : \forall c' \in \mathbb{C}_{I,CR} : \exists m \in I_{\mathfrak{M}} : \\ m \text{ contains } c' \wedge m \text{ consistent to } \mathbb{C}\mathbb{R}^+ \end{aligned} \quad (5.5)$$

Second, we consider $CR \in \mathbb{C}\mathbb{R}$. Due to the definition of compatibility of $\mathbb{C}\mathbb{R}$ and Lemma 5.3 showing equality of consistency of m regarding $\mathbb{C}\mathbb{R}$ and $\mathbb{C}\mathbb{R}^+$ it is true that:

$$\begin{aligned} \forall CR \in \mathbb{C}\mathbb{R} : \forall c' \in \mathbb{C}_{I,CR} : \exists m \in I_{\mathfrak{M}} : \\ m \text{ contains } c' \wedge m \text{ consistent to } \mathbb{C}\mathbb{R}^+ \end{aligned} \quad (5.6)$$

With Equation 5.5 and Equation 5.6, we have shown compatibility of $\mathbb{C}\mathbb{R}^+$ if $\mathbb{C}\mathbb{R}$ is compatible. \square

5.3. A Formal Approach to Prove Compatibility

In this section, we use the definition of compatibility to derive a formal approach for proving compatibility of consistency relations. The approach 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 if a consistency relation tree that is equivalent to the set of relations can be found by only removing redundant relations from that set. Finding 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 for witnessing compatibility is based on a definition of equivalence for sets of consistency relations. We consider two sets of consistency relations equivalent if they consider the same sets of models as consistent:

Definition 5.4 (Consistency Relations Equivalence)

Let $\mathbb{CR}_1, \mathbb{CR}_2$ be two sets of consistency relations defined for a tuple of metamodels \mathfrak{M} . We say that:

$$\begin{aligned} \mathbb{CR}_1 \text{ equivalent to } \mathbb{CR}_2 &\Leftrightarrow \forall m \in I_{\mathfrak{M}} : \\ m \text{ consistent to } \mathbb{CR}_1 &\Leftrightarrow m \text{ consistent to } \mathbb{CR}_2 \end{aligned}$$

The goal of our approach is to find a set of consistency relations that is compatible and equivalent to a given consistency relation set. We will later use

equivalence to introduce a specific notion of redundancy that is compatibility-preserving. In the following, we first consider structures of consistency relation sets that are inherently compatible and afterwards consider redundancy as a means to find an equivalent representation of a relation set that has such a structure.

We first consider two essential properties of a consistency relation set that lead to its inherent compatibility:

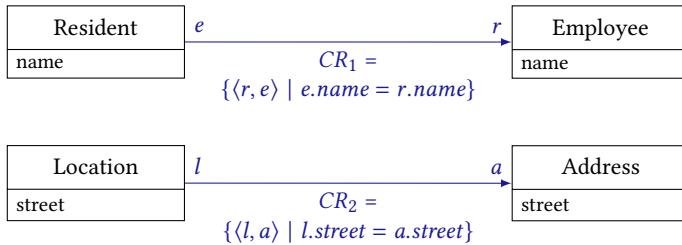
Composability: We show that the union of independent, compatible sets of consistency relations is compatible.

Trees: We show that relations fulfilling a special notion of *consistency relation trees* are inherently compatible.

In consequence, we know that a consistency relation set that is composed of independent subsets of consistency relation trees is inherently compatible. Afterwards, we discuss how to find redundant consistency relations to be able to reduce and decompose sets of relations into compositions of independent consistency relation trees.

5.3.1. Independence of Consistency Relations

We consider consistency relation sets as independent if there are no transitive consistency relations induced by relations from both sets, i.e., for each object in a model consistency is only restricted by one of those sets.

**Figure 5.10.**: Two independent (sets of) consistency relations. Taken from [Kla+20a].**Definition 5.5** (Consistency Relation Sets Independence)

Let \mathbb{CR}_1 and \mathbb{CR}_2 be two sets of consistency relations. We say that:

\mathbb{CR}_1 and \mathbb{CR}_2 are independent : \Leftrightarrow

$$\forall CR \in \mathbb{CR}_1 : \forall CR' \in \mathbb{CR}_2 : \forall CR_1, \dots, CR_k \in \mathbb{CR}_1 \cup \mathbb{CR}_2 :$$

$$CR \otimes CR_1 \otimes \dots \otimes CR_k \otimes CR' = \emptyset$$

$$\wedge CR' \otimes CR_1 \otimes \dots \otimes CR_k \otimes CR = \emptyset$$

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

$$\forall \mathbb{CR}_1, \mathbb{CR}_2 \subseteq \mathbb{CR} : \mathbb{CR}_1 \cap \mathbb{CR}_2 = \emptyset \wedge \mathbb{CR}_1 \cup \mathbb{CR}_2 = \mathbb{CR}$$

$$\Rightarrow \neg(\mathbb{CR}_1 \text{ and } \mathbb{CR}_2 \text{ are independent})$$

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 objects that are related by the consistency relations, they are considered independent according to Definition 5.5.

An important property of independent sets of consistency relations 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 compatible sets of consistency relations. Then $\mathbb{CR}_1 \cup \mathbb{CR}_2$ is compatible.

Proof. Since \mathbb{CR}_1 is compatible, per definition there is a model tuple m for each condition element c of the left condition of each consistency relation in \mathbb{CR}_1 that contains c and that is consistent to \mathbb{CR}_1 . Taking such an m , we create a new m' by removing all elements from m , which are contained in any condition elements in any consistency relation in \mathbb{CR}_2 and thus potentially require other elements to occur to be considered consistent to that consistency relation. In consequence, m' does not contain any condition elements from consistency relations in \mathbb{CR}_2 and is thus consistent to \mathbb{CR}_2 by definition. Additionally, 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 relations in \mathbb{CR}_1 , which require the existence of the removed elements. In consequence, for each condition element c of each consistency relation in \mathbb{CR}_1 there is a model tuple that contains c and that is consistent to $\mathbb{CR}_1 \cup \mathbb{CR}_2$. The analogous argumentation applies to the consistency relations in \mathbb{CR}_2 , which is why 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

We want to find a property or specific structure of consistency relation sets, which leads to inherent consistency of the contained relations. If we can then prove for a set of relations that we reduce it to such a structure in a compatibility-preserving way, we know that the relations are compatible. Intuitively, such a structure is given by a kind of trees, because then there are

Intuitively,
trees are
compatible

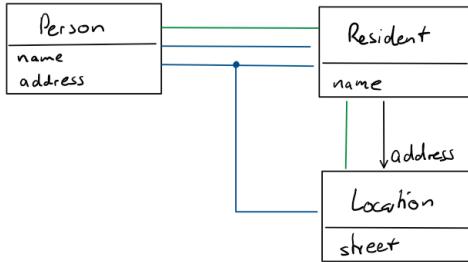


Figure 5.11.: A hypertree (blue edges) with its host graph (green edges).

no two concatenations of consistency relations which can relate elements in a contradictory way.

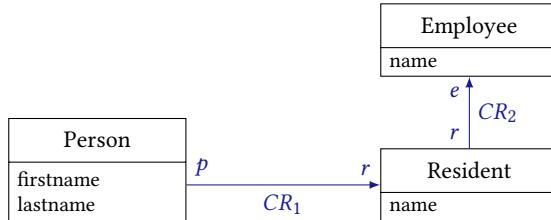
Since consistency relations put tuple of classes into relation, it may seem natural to use a hypergraph, consisting of the classes as vertices and the class tuples of the consistency relations as hyperedges, to describe them, and expect that if the hypergraph induced by the consistency relations forms a hypertree¹, it is compatible. An example for a hypertree is depicted in Figure 5.11. It consists of two hyperedges, one only relating persons and residents and one relating persons with residents and their address locations. This is a scenario that may occur when one consistency relation puts persons and residents with their names into relation, and another describes the relation of their addresses. These relations form a hypertree, as there is a tree, depicted in the figure, of which the vertices of all hyperedges are connected subtrees.

Relations
may be
compatible
if their
induced hy-
pergraphs
are
hypertrees

The relations in the example may, however, not be necessarily compatible. They can, for example, define a contradictory relation between the names of persons and residents. In general, hypertrees induced by consistency relations are not necessarily compatible, because hyperedges can be subsets of other hyperedges and the relations inducing these hyperedges may contradict each other. Additionally, the hyperedges to do only put sets of classes into relation and are not able distinguish between the sets belonging to different metamodels. Thus, we have to exclude that the same classes are put into

Relations
inducing
hypertrees
are not
necessarily
compatible

¹ A hypergraph is a hypertree if there is a tree, such that every edge of the hypergraph is a set of vertices of a connected subtree of the tree [Bra+98]. Such a tree is called a *host graph*.



$$CR_1 = \{(p, r) \mid r.name = p.firstname + " " + p.lastname\}$$

$$CR_2 = \{(r, e) \mid r.name = e.name\}$$

Figure 5.12.: A consistency relation tree $\{CR_1, CR_1^T, CR_2, CR_2^T\}$. Taken from [Kla+20a].

relation by multiple consistency relations in a different way. This leads to our definition consistency relation trees.

Definition 5.6 (Consistency Relation Tree)

Let \mathbb{CR} be a symmetric, connected set of consistency relations. We say:

\mathbb{CR} is a consistency relation tree : \Leftrightarrow

$$\forall CR = CR_1 \otimes \dots \otimes CR_m \in \mathbb{CR}^+ :$$

$$\forall CR' = CR'_1 \otimes \dots \otimes CR'_n \in \mathbb{CR}^+ \setminus CR :$$

$$\forall s, t \mid s \neq t : CR_s \neq CR_t^T \wedge CR'_s \neq CR'_t^T$$

$$\Rightarrow \mathfrak{C}_{l,CR} \cap \mathfrak{C}_{l,CR'} = \emptyset \vee \mathfrak{C}_{r,CR} \cap \mathfrak{C}_{r,CR'} = \emptyset$$

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

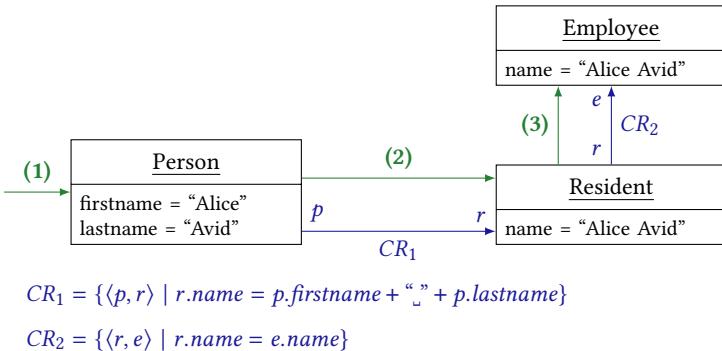


Figure 5.13.: An example for constructing a model with the condition element of CR_1 containing the person named "Alice Avid" for a consistency relation tree according to the consistency relations in Figure 5.12. Taken from [Kla+20a].

Example 5.4. Figure 5.12 depicts a rather simple consistency relation tree. Persons are related to residents and residents are related to employees, all having the same names respectively a concatenation of *firstname* and *lastname*, by the relations CR_1, CR_2 , as well as their transposed relations CR_1^T, 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. 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 may be put into relation by consistency relations but there is only a subset of them that is put into relation by different consistency relations. We can now prove that a set of consistency relations that is a consistency relation tree is always compatible. To not disturb the reading flow due to the complexity of the proof for that statement, the complete proof with an auxiliary lemma can be found in Section A.1. We only provide a proof sketch in the following.

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 Section A.1. It is based on the proven insight that in a consistency relation tree, starting with each of the consistency relations 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 induce no cycle between any of the classes in the metamodels. We can use this insight to define a construction approach for such sequences given a set of consistency relations that forms a consistency relation tree. For proving compatibility, we need to show that for each condition element in a consistency relation we are able to find a consistent model tuple that contains this condition element. Thus, starting from each condition element of each relation, we add a corresponding element according to the consistency relation. We then inductively add further elements for the just added elements, which are required by further consistency relations. Due to the defined properties of consistency relation trees, we can show that this construction is always possible and terminates with a consistent tuple of models.

A simple example for that construction is depicted in Figure 5.13, based on the relations in the consistency relation tree in Figure 5.12, 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 have shown that consistency relation sets fulfilling a special notion of 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.

However, consistency relations fulfill such a structure only in specific cases. In general, like in our motivational example in Figure 5.1, there may be different consistency relations putting the same elements into relation, such

that the definition for consistency relation trees is not fulfilled. In the following, we discuss how to find a consistency relation tree that is equivalent to a given set of consistency relations, such that this equivalence witnesses compatibility.

5.3.3. Redundancy of Consistency Relations

We have introduced specific structures of consistency relations that are inherently compatible. If a given set of consistency relations does not represent one of those structures, especially because there are multiple consistency relations putting the same classes into relation, it is unclear whether such a set is compatible.

In the following, we present an approach to reduce a set of consistency relations to a structure of independent consistency relation trees. The essential idea is to find relations within the set, which do not change compatibility of the consistency relation set whether or not they are contained in it. An approach that finds such relations and—virtually—removes them from the set until the remaining relations form a set of 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:

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

To be able to find such a compatibility-preserving relation, 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 specify a notion of redundancy in the following.

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:

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

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 as consistent.

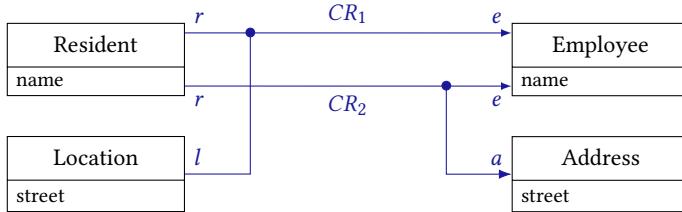
Lemma 5.7 (Redundant Relations Equivalence)

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. Like 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 4.18 for consistency, which requires models be consistent to all consistency relations in a set to be considered consistent, thus restricting the set of consistent model tuples by adding further consistency relations. In consequence, it holds that:

$$m \text{ consistent to } \mathbb{CR} \Rightarrow m \text{ consistent to } \mathbb{CR} \setminus \{CR\}$$

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 to be consistent that are also considered consistent to another consistency relation in the transitive closure of the consistency relation set.



$$\begin{aligned}
 CR_1 &= \{(r, l, e) \mid r.name \neq "" \\
 &\quad \wedge (r.name = e.name \vee r.name = e.name.toLowerCase)\} \\
 CR_2 &= \{(r, (e, a)) \mid r.name = e.name \wedge a.street \neq "\"
 \end{aligned}$$

Figure 5.14.: Redundant consistency relation CR_1 in $\{CR_1, CR_2\}$. Taken from [Kla+20a].

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} :

$$\mathbf{m} \text{ consistent to } (\mathbb{CR} \setminus \{CR\})^+ \Rightarrow \mathbf{m} \text{ consistent to } \mathbb{CR}$$

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. In consequence, the previous implication is also true for $\mathbb{CR} \setminus \{CR\}$ rather than $(\mathbb{CR} \setminus \{CR\})^+$. Summarizing, \mathbb{CR} and $\mathbb{CR} \setminus \{CR\}$ are equivalent. \square

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 Figure 5.14, 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 exists, whereas CR_2 requires that already for an existing resident. Furthermore, the residents for which CR_1 defines any consistency requirements are a subset of those for which CR_2 defines consistency requirements, as CR_1 does not make any statements about residents having an empty name. Thus, the left condition elements of CR_1 are a subset of those of CR_2 . In consequence, if CR_1 requires consistency for a resident and a location, CR_2 requires it anyway, because it already defines consistency for the contained resident.

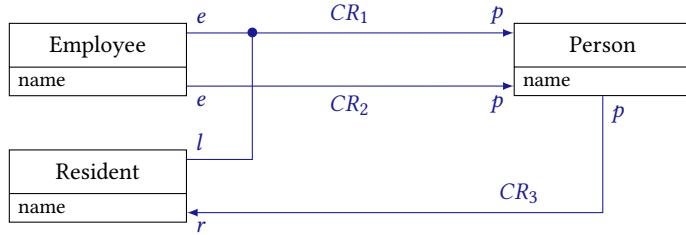
Additionally, a redundant consistency relation, such as CR_1 , must have weaker requirements for the elements at the right side, such that one of the consistent right condition elements is contained anyway because another relation already required them. This means that the relation may have a smaller set of classes, of whom instances are required to consider the models consistent, and there may be more condition elements of the right side that are considered consistent with condition elements of the left side to not restrict the elements considered consistent. In case of CR_1 , it only requires an employee to exist for a resident compared to CR_2 , which also requires a non-empty address to exist. Additionally, CR_1 does not restrict the employees that are considered consistent to employees compared to 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, our intuitive redundancy definition is not compatibility-preserving.

Proposition 5.8 (Redundant Relations Non-Compatibility)

Let \mathbb{CR} be a compatible set of consistency relations and let CR be a consistency relation that is redundant in $\mathbb{CR} \cup \{CR\}$. Then CR is not necessarily compatibility-preserving, i.e., $\mathbb{CR} \cup \{CR\}$ is not necessarily compatible.

Proof. We prove the proposition by providing a counterexample. Consider the example in Figure 5.15. CR_2 relates each employee to a person with the same name and CR_3 relates each person to a resident with the same



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

$$CR_2 = \{(e, p) \mid e.name = p.name\}$$

$$CR_3 = \{(p, r) \mid r.name = p.name.toLower\}$$

Figure 5.15.: 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+20a].

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 respectively employee 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 $\{CR_1, CR_2, CR_3\}$ according to Definition 5.8, because 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 only those in which the model also contains a resident such that the employee name is the one of the resident in uppercase.

However, $\{CR_1, CR_2, CR_3\}$ is not compatible. Intuitively, this is due to the fact that CR_1 and CR_3 define an incompatible mapping between the names of residents and persons. This is also reflected by Definition 5.3 for compatibility. Take a model with an employee and a resident named A . This is a condition element in \mathbb{C}_{l, CR_1} . Consequentially, CR_1 requires a person A to exist. Furthermore CR_3 requires a resident with name a to exist. In consequence, there are two tuples of employees and residents, both with employee A and one with resident A respectively resident a each, for which a consistent person with name A is required by CR_1 . However, CR_1 actually forbids to have two residents, 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 there is only one person that to which both can be mapped, namely the one with the uppercase name, so there is no witness structure with a unique mapping as required by Definition 4.18 for consistency. This example shows that adding a redundant consistency relation to a compatible set of consistency relations does not lead to a compatible consistency relation set. \square

5.3.4. Compatibility-Preserving Redundancy

In consequence of Proposition 5.8, we need a stronger definition of redundancy that is compatibility-preserving. In the example in Figure 5.15 showing Proposition 5.8, we have seen that it is problematic if a redundant consistency relation considers more classes in its left condition than the relation it is redundant to. Therefore, we restrict 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:

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

The definition of left-equal redundancy is similar to the redundancy definition but restricts the notion of redundancy to cases in which the left condition of the redundant consistency relation CR considers the same classes than 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, apart from the additional restriction for the class tuples, redundancy of a left-equal redundant relation is a direct implication of the definition. \square

Before showing that left-equal redundancy is compatibility-preserving, we introduce an auxiliary lemma that shows that if a model tuple contains any left condition element of a left-equal redundant relation, i.e., if that redundant relation requires the model tuple to contain corresponding elements for that object tuple to be consistent, there is also 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 set of consistency relations $\mathbb{C}\mathbb{R}$ for a tuple of metamodels \mathfrak{M} . Then it holds that:

$$\begin{aligned} \exists CR' \in (\mathbb{C}\mathbb{R} \setminus \{CR\})^+ : \forall c_l \in \mathbb{C}_{l,CR} : \exists c'_l \in \mathbb{C}_{l,CR'} : \\ \forall m \in I_{\mathfrak{M}} : m \text{ contains } c'_l \Rightarrow m \text{ contains } c_l \end{aligned}$$

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

$$\begin{aligned} \exists CR' \in (\mathbb{C}\mathbb{R} \setminus \{CR\})^+ : \forall m \in I_{\mathfrak{M}} : \\ m \text{ consistent to } CR' \Rightarrow m \text{ consistent to } CR \wedge \mathfrak{C}_{l,CR} = \mathfrak{C}_{l,CR'} \end{aligned}$$

This implies that:

$$\exists CR' \in (\mathbb{C}\mathbb{R} \setminus \{CR\})^+ : \forall c_l \in \mathbb{C}_{l,CR} : c_l \in \mathbb{C}_{l,CR'}$$

Because if there was a $c_l \in \mathbb{C}_{l,CR}$ so that $c_l \notin \mathbb{C}_{l,CR'}$, then the model tuple m only consisting of c_l would be consistent to CR' , because it does not require any other elements to exist for considering the model tuple consistent, whereas there is at least one $\langle c_l, c_r \rangle \in CR$, so that m needs to contain c_r for considering m 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 from $\mathbb{C}_{I,CR'}$ that a model tuple m contains if it contains an element from $\mathbb{C}_{I,CR}$, which proves the statement in the lemma. \square

Theorem 5.11 (Left-Equal Redundancy Compatibility)

Let $\mathbb{C}\mathbb{R}$ be a compatible set of consistency relations for a tuple of metamodels \mathfrak{M} and let CR be a consistency relation that is left-equal redundant in $\mathbb{C}\mathbb{R} \cup \{CR\}$. Then $\mathbb{C}\mathbb{R} \cup \{CR\}$ is compatible.

Proof. Due to left-equal redundancy of CR in $\mathbb{C}\mathbb{R} \cup \{CR\}$, which also implies general redundancy according to Definition 5.8, $\mathbb{C}\mathbb{R}$ and $\mathbb{C}\mathbb{R} \cup \{CR\}$ are equivalent, according to Lemma 5.7. Due to that equivalence, we know that for any model tuple $m \in I_{\mathfrak{M}}$:

$$m \text{ consistent to } \mathbb{C}\mathbb{R} \Leftrightarrow m \text{ consistent to } \mathbb{C}\mathbb{R} \cup \{CR\} \quad (5.1)$$

It follows from Definition 5.3 for compatibility and Equation 5.1:

$$\begin{aligned} \forall CR' \in \mathbb{C}\mathbb{R} : \forall c_l \in \mathbb{C}_{I,CR'} : \exists m \in I_{\mathfrak{M}} : \\ m \text{ contains } c_l \wedge m \text{ contains } \mathbb{C}\mathbb{R} \cup \{CR\} \end{aligned} \quad (5.2)$$

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

$$\begin{aligned} \forall CR' \in \mathbb{C}\mathbb{R}^+ : \forall c_l \in \mathbb{C}_{I,CR'} : \exists m \in I_{\mathfrak{M}} : \\ m \text{ contains } c_l \wedge m \text{ consistent to } \mathbb{C}\mathbb{R}^+ \end{aligned} \quad (5.3)$$

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

$$\begin{aligned} \exists CR' \in \mathbb{C}\mathbb{R}^+ : \forall c_l \in \mathbb{C}_{I,CR} : \exists c'_l \in \mathbb{C}_{I,CR'} : \forall m \in I_{\mathfrak{M}} : \\ m \text{ contains } c'_l \Rightarrow m \text{ contains } c_l \end{aligned} \quad (5.4)$$

The combination of Equation 5.3 and Equation 5.4 gives:

$$\begin{aligned} \exists CR' \in \mathbb{CR}^+ : \forall c_l \in \mathbb{C}_{l,CR} : \exists c'_l \in \mathbb{C}_{l,CR'} : \\ (\forall m \in I_{\mathfrak{M}} : m \text{ contains } c'_l \Rightarrow m \text{ contains } c_l) \\ \wedge (\exists m \in I_{\mathfrak{M}} : m \text{ contains } c'_l \wedge m \text{ consistent to } \mathbb{CR}^+) \end{aligned}$$

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

$$\begin{aligned} \forall c_l \in \mathbb{C}_{l,CR} : \exists m \in I_{\mathfrak{M}} : \\ m \text{ contains } c_l \wedge m \text{ consistent to } \mathbb{CR}^+ \end{aligned}$$

Due to Equation 5.1 and Lemma 5.3, which shows equality of consistency for a consistency relation set and its transitive closure, this is equivalent to:

$$\begin{aligned} \forall c_l \in \mathbb{C}_{l,CR} : \exists m \in I_{\mathfrak{M}} : \\ m \text{ contains } c_l \wedge m \text{ consistent to } \mathbb{CR} \cup \{CR\} \end{aligned} \quad (5.5)$$

The combination of Equation 5.2 and Equation 5.5 shows that $\mathbb{CR} \cup \{CR\}$ fulfills Definition 5.3 for compatibility. \square

Corollary 5.12 (Transitive Redundancy Compatibility)

Let \mathbb{CR} be a compatible set of consistency relations and let CR_1, \dots, CR_k be consistency relations with:

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

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

Proof. This is an inductive implication of Theorem 5.11, because \mathbb{CR} is compatible and sequentially adding CR_i to $\mathbb{CR} \cup \{CR_1, \dots, CR_{i-1}\}$ ensures that $\mathbb{CR} \cup \{CR_1, \dots, CR_i\}$ is compatible, because $\mathbb{CR} \cup \{CR_1, \dots, CR_{i-1}\}$ was compatible as well. \square

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, \dots, CR_k according to Corollary 5.12 such that we know that $\mathbb{CR} \setminus \{CR_1, \dots, CR_k\}$ is compatible, then it is proven that \mathbb{CR} is compatible.

5.3.5. An Algorithm to Prove Compatibility

In the previous sections, we have proven the following three central insights:

1. Compatibility is composable: If independent sets of consistency relations are compatible, then their union is compatible as well (Theorem 5.5).
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, then that set is compatible (Theorem 5.6).
3. Left-equal redundancy is compatibility-preserving: Adding a left-equal redundant consistency relation to a compatible set of consistency relations, that set unified with the redundant relation is still 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—virtually—from the set leads to a set of independent consistency relation trees, it is proven that the initial set of consistency relations is compatible.

An algorithm that realizes this procedure is given in Algorithm 1. 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 there it contains a redundant relations. It is easy to see that this algorithm is correct, as it implements the proven findings of the previous sections. This does, however, not mean that implementing the sub-procedures is trivial. We will provide a practical approach to realize them in the subsequent section.

Theorem 5.13 (Compatibility Algorithm Correctness)

Algorithm 1 is correct, i.e., it only returns TRUE if the given consistency relations set $\mathbb{C}\mathbb{R}$ is actually compatible.

Algorithm 1. Proof for compatibility of consistency relations.

```

1: procedure PROVECOMPATIBILITY( $\mathbb{CR}$ )
2:    $isTree \leftarrow \text{IsRELATIONTREE}(\mathbb{CR})$ 
3:   if  $isTree$  then
4:     return TRUE
5:   end if
6:    $hasIndependentSubsets \leftarrow \text{HASINDEPENDENTSUBSETS}(\mathbb{CR})$ 
7:   if  $hasIndependentSubsets$  then
8:      $\{\mathbb{CR}_1, \mathbb{CR}_2\} \leftarrow \text{FINDINDEPENDENTSUBSETS}(\mathbb{CR})$ 
9:      $isFirstSetCompatible \leftarrow \text{PROVECOMPATIBILITY}(\mathbb{CR}_1)$ 
10:     $isSecondSetCompatible \leftarrow \text{PROVECOMPATIBILITY}(\mathbb{CR}_2)$ 
11:    return  $isFirstSetCompatible \wedge isSecondSetCompatible$ 
12:   end if
13:    $CR_{redundant} \leftarrow \text{FINDREDUNDANTRELATION}(\mathbb{CR})$ 
14:   if  $CR_{redundant} \neq \emptyset$  then
15:      $\mathbb{CR}' \leftarrow \mathbb{CR} \setminus CR_{redundant}$ 
16:     return PROVECOMPATIBILITY( $\mathbb{CR}'$ )
17:   end if
18:   return FALSE
19: end procedure

```

Proof. We make a case distinction for the situation in which the algorithm returns a result:

1. When the consistency relation set is a tree, the algorithm directly returns TRUE (Lines 2–5), which is correct according to Theorem 5.6.
2. When 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 6–12), which is correct according to Theorem 5.5.
3. When 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 13–17), which is correct according to Corollary 5.12.
4. In all other cases, the algorithm returns FALSE (Line 18). \square

The algorithm is, however, also *conservative*. If the approach finds redundant relations, such that a consistency relation set can be reduced to a set of independent consistency relations 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 may be compatibility-preserving relations that do not fulfill the definition of left-equal redundancy. Furthermore, an actual technique to identify left-equal redundant relations may not be able to find all of them automatically, as we will see later.

Theorem 5.14 (Compatibility Algorithm Conservativeness)

Algorithm 1 is conservative, i.e., it is correct, but if it returns FALSE the given consistency relations set $\mathbb{C}\mathbb{R}$ is not necessarily incompatible.

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. \square

The example given in the proof for conservativeness shows that the strictness of our definition for left-equal redundancy (Definition 5.9). We will, however, see in the evaluation 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 how actual transformations, at the example of QVT-R, can be represented in a graph-based structure, such that it 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 the means of an SMT solver applied to the constraints defined in QVT-R relations.

5.4. A Practical Approach to Prove Compatibility

The formal approach adopted in the previous section demonstrates that deriving a consistency relation tree from a set of consistency relations \mathbb{CR} is an effective way to prove compatibility. It is especially a consequence of Theorem 5.6. Given that proving compatibility amounts to the construction of a consistency relation tree, this result lends itself well to an operationalization. To this end, we propose an algorithm that turns the proof of compatibility into an operational procedure. For the most part, this algorithm is based on results previously developed and described in detail in the master’s thesis of Pepin [Pep19], which was supervised by the author of this thesis, and, as introduced before, is mostly taken from the article [Kla+20a] describing those results.

Constructing a consistency relation tree can be achieved by finding and virtually removing every redundant consistency relation in a given set of consistency relation set \mathbb{CR} . Removing redundant relations in a consistency relation set to generate a tree, or a set of independent trees, is called *decomposition*. Designing a decomposition procedure requires to represent consistency relations in actual model transformation languages and to provide a way to test the redundancy of a consistency relation. We first highlight a mapping between the consistency framework developed in the previous sections and the QVT-R transformation language through the use of *predicates*. As a consequence, results achieved with consistency relations become also applicable with QVT-R. Then, we design a fully automated decomposition procedure that takes a *consistency specification*, i.e., a set of QVT-R transformations, as an input and virtually removes as many redundant consistency relations as possible. In the decomposition procedure, each consistency relation removal is a two-step process. First, a potentially redundant relation and an alternative concatenation of consistency relations are identified. Then, a redundancy test is performed: it answers whether it is possible or not to remove the candidate relation using the alternative concatenation. Uncoupling the search for candidates from the decision-making makes it possible to plug in different strategies to test redundancy. This section focuses on the first step, i.e., setting up a structure suited to the detection of possibly redundant relations and finding candidates for the redundancy test.

5.4.1. Consistency Relations in Transformation Languages

According to Definition 4.17, consistency relations are built by enumerating valid co-occurring condition elements. However, developers do not enumerate valid models when writing transformations. They rather describe patterns for models to be considered consistent and sometimes how consistency is restored after a model was modified. In relational transformation languages, developers define consistency as a set of criteria that models must fulfill. Criteria are expressed using metamodel elements (i.e., class properties), as objects are only distinguished by their contents. For example, an Employee object and a Person object are considered consistent if their name attributes are equal.

Criteria are equivalent to predicates, i.e., Boolean-valued filter functions: consistency relations are then defined as sets of pairs of condition elements for which the predicate evaluates to TRUE. Therefore, we move from an extensional to an intensional, programming-like definition of consistency relations, making it easier to link consistency relations with QVT-R transformations. In Subsection 4.1.1, we have shown that both types of specifications provide equal expressiveness.

5.4.1.1. Properties, Property Values and Predicates

We first define concepts that allow the intensional construction of consistency relations. The main idea is to select some properties in each metamodel and to define a predicate that filters values of these properties.

Definition 5.10 (Property Set)

A property set for a class C is a subset \mathbb{P}_C of properties of C , i.e., $\mathbb{P}_C = \{P_{C,1}, \dots, P_{C,n}\}$ such that $P_{C,i} \in C$.

A property set models a choice of properties that play a role in the definition of a predicate in order to distinguish consistent and non-consistent condition elements. Not all properties have to be used to describe consistency, particularly in incremental model transformations.

Definition 5.11 (Tuple of Property Sets)

For a class tuple \mathfrak{C} , it is possible to build a tuple of property sets by defining a property set for every class, i.e., $\mathfrak{P}_{\mathfrak{C}} = \langle \mathbb{P}_{C_1}, \dots, \mathbb{P}_{C_n} \rangle = \langle \{P_{C_1,1}, \dots, P_{C_1,m}\}, \dots, \{P_{C_n,1}, \dots, P_{C_n,k}\} \rangle$.

Tuples generalize the use of property sets to class tuples, because conditions themselves are made up of class tuples.

Definition 5.12 (Property Value Set)

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

Just as a property set is a subset of properties of a class C , a property value set is a subset of property values of an object o that instantiates C . The property value set is a fragment of o that provides enough information to evaluate consistency.

Definition 5.13 (Predicate)

A predicate for two class tuples \mathfrak{C}_l and \mathfrak{C}_r is a triple $\pi = (\mathfrak{P}_{\mathfrak{C}_l}, \mathfrak{P}_{\mathfrak{C}_r}, f_\pi)$ where $\mathfrak{P}_{\mathfrak{C}_l}$ (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 that takes instances of $\mathfrak{P}_{\mathfrak{C}_l}$ and $\mathfrak{P}_{\mathfrak{C}_r}$ as an input, i.e., $f_\pi : I_{\mathfrak{P}_{\mathfrak{C}_l}} \times I_{\mathfrak{P}_{\mathfrak{C}_r}} \rightarrow \{\text{TRUE, FALSE}\}$.

For readability purposes, it is sometimes useful to group all the properties used by a predicate within the same set. As a consequence, the property collection P_π of a predicate $\pi = (\mathfrak{P}_{\mathfrak{C}_l}, \mathfrak{P}_{\mathfrak{C}_r}, f_\pi)$ is defined as:

$$P_\pi = (\bigcup_j \mathfrak{P}_{\mathfrak{C}_l,j}) \cup (\bigcup_k \mathfrak{P}_{\mathfrak{C}_r,k})$$

The definition of a predicate involves the choice of some properties for each class that occurs in one of the two class tuples of a consistency relation CR .

It also involves the definition of an appropriate function f_π that answers whether instances of these properties, i.e., property values, evaluate to TRUE or FALSE. In the former case, objects containing these property values match the predicate: the associated consistency relation pair is in CR . In the latter case, objects do not match the predicate and are not considered consistent, i.e., do not occur in CR . The expression of f_π is the choice of the developer who defines it according to consistency criteria.

Predicates model the way consistency relations are defined in model transformation languages. Objects can only be distinguished by their property values. Thus, the distinction between consistent and non-consistent pairs of condition elements is always based on some attribute or reference values.

5.4.1.2. Predicate-Based Consistency Relations

Definition 5.14 (Property Matching)

A property value set $\mathbb{p}_C = \{p_{C,1}, \dots, p_{C,n}\}$ matches an object o if and only if

$$o \in I_C \wedge \forall p_{C,i} : p_{C,i} \in o$$

Similarly, a tuple of property value sets $\mathbb{p}_{\mathfrak{C}} = \langle \mathbb{p}_{C_1}, \dots, \mathbb{p}_{C_n} \rangle$ matches a tuple of objects $\mathfrak{o} = \langle o_1, \dots, o_k \rangle$ if and only if $|\mathbb{p}_{\mathfrak{C}}| = |\mathfrak{o}|$ and $\forall i : \mathbb{p}_{C_i}$ matches o_i .

Definition 5.15 (Predicate-Based Consistency Relation)

Let \mathfrak{c}_l and \mathfrak{c}_r be two conditions for two class tuples $\mathfrak{C}_{\mathfrak{c}_l}$ and $\mathfrak{C}_{\mathfrak{c}_r}$. Let Π be a set of predicates for $\mathfrak{C}_{\mathfrak{c}_l}$ and $\mathfrak{C}_{\mathfrak{c}_r}$. A Π -based consistency relation CR_Π is a subset of pairs of condition elements such that:

$$\begin{aligned} CR_\Pi = \{ & (c_l, c_r) \mid \forall (\mathfrak{p}_{\mathfrak{C}_{\mathfrak{c}_l}}, \mathfrak{p}_{\mathfrak{C}_{\mathfrak{c}_r}}, f_\pi) \in \Pi : \\ & \exists \mathfrak{p}\mathfrak{C}_{\mathfrak{c}_l} \in \mathfrak{P}\mathfrak{C}_{\mathfrak{c}_l}, \mathfrak{p}\mathfrak{C}_{\mathfrak{c}_r} \in \mathfrak{P}\mathfrak{C}_{\mathfrak{c}_r} : \\ & \mathfrak{p}_{\mathfrak{C}_{\mathfrak{c}_l}} \text{ matches } c_l \wedge \mathfrak{p}_{\mathfrak{C}_{\mathfrak{c}_r}} \text{ matches } c_r \wedge f_\pi(p_{\mathfrak{C}_{\mathfrak{c}_l}}, p_{\mathfrak{C}_{\mathfrak{c}_r}}) = \text{TRUE} \} \end{aligned}$$

The construction of a predicate-based consistency relation is of importance for the practicality of model transformation languages. The developer can produce a consistency specification by retaining some object properties and imposing conditions on values of these properties via a predicate function. Then, the construction of the consistency relation fully amounts to the evaluation of the predicate function.

Example 5.6. *The following example demonstrates how to build a consistency relation CR_{PR} based on predicates between Person and Resident metamodels, according to the example in Figure 5.1. CR_{PR} ensures that the name of a Resident object is the concatenation of the first name and the last name of a Person object. It also ensures that both objects have the same address. First, CR_{PR} involves one class in each metamodel, resulting in two class tuples: $\mathfrak{C}_P = \langle \text{Person} \rangle$ and $\mathfrak{C}_R = \langle \text{Resident} \rangle$. There are two conditions to achieve consistency, which are equal names and equal addresses, so CR_{PR} will be made up of two predicates. The first predicate needs the `firstname` and `lastname` attributes in Person and the `name` in Resident, so $\mathfrak{P}_{\mathfrak{C}_P,1} = \langle \{\text{firstname}, \text{lastname}\} \rangle$ and $\mathfrak{P}_{\mathfrak{C}_R,1} = \langle \{\text{name}\} \rangle$. Similarly, $\mathfrak{P}_{\mathfrak{C}_P,2} = \langle \{\text{address}\} \rangle$ and $\mathfrak{P}_{\mathfrak{C}_R,2} = \langle \{\text{address}\} \rangle$. The functions of the predicate, shortly denoting name as n , `firstname` as fn , `lastname` as ln , as well as address of Person as a_P and of Resident as a_R , look as follows:*

$$f_{\pi,1}(\langle \{n\} \rangle, \langle \{fn, ln\} \rangle) = \begin{cases} \text{TRUE} & \text{if } n = fn + " " + ln \\ \text{FALSE} & \text{otherwise} \end{cases}$$

$$f_{\pi,2}(\langle \{a_P\} \rangle, \langle \{a_R\} \rangle) = \begin{cases} \text{TRUE} & \text{if } a_P = a_R \\ \text{FALSE} & \text{otherwise} \end{cases}$$

CR_{PR} is a Π -based consistency relation where Π is the set of predicates $\{(\mathfrak{P}_{\mathfrak{C}_P,1}, \mathfrak{P}_{\mathfrak{C}_R,1}, f_{\pi,1}), (\mathfrak{P}_{\mathfrak{C}_P,2}, \mathfrak{P}_{\mathfrak{C}_R,2}, f_{\pi,2})\}$.

5.4.1.3. Consistency Relations in QVT-R Transformations

There is a variety of model transformation languages [CH03]. Like programming languages, transformation languages can be divided into two main paradigms: *declarative* languages that focus on *what* transformations should perform and *imperative* languages that describe *how* transformations should be performed. The QVT standard [Obj16a] provides three transformation languages: Operational, Core and Relations.

```
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 { PRECOND }] [where { INVARIANT }]
    }
    [top] relation R2 { ... }
}
```

Listing 5.1.: Simplified structure of a QVT-R transformation. Taken from [Kla+20a].

The most relevant language for consistency specification is QVT Relations (QVT-R). It is a declarative and relational language that shares many concepts with the consistency framework developed before. It lends itself well to mathematical formalization [Ste10]. As with consistency relations, QVT-R supports bidirectionality. Transformations written in QVT-R can be used with two execution modes. First, a *checkonly* mode to check that models fulfill consistency relations. Second, an *enforce* mode to repair consistency in a given direction if not all relations are fulfilled. The simplified structure of a QVT-R transformation is as follows and also depicted in Listing 5.1.

A QVT-R transformation can check or repair consistency of models it receives as parameters. Models are typed models, i.e., their structure conforms to a type defined by the metamodel. Each `transformation` is composed of `relations`, which define the rules for objects of both models to be consistent. 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 already invoked. The QVT-R mechanism for checking consistency is based on pattern matching. Shared information between objects of different models is represented by variables assigned to class properties. These variables contain values that must remain consistent from one object to another. Therefore, there must exist some assignment that matches all patterns at the same time for classes in a relation to be consistent.

```

fstn: String; lstn: String;
inc: Integer;

domain pers p:Person {
    firstname=fstn, lastname=lstn,
    income=inc
};

domain emp e:Employee {
    name=fstn + ' ' + lstn,
    salary=inc
};

```

Listing 5.2.: Two domains, each with one domain pattern. Taken from [Kla+20a].

More precisely, each QVT-R relation contains two domains that contain themselves *domain patterns*. In QVT terminology, a domain pattern is a variable instantiating a class. Values that this variable can take are constrained by conditions on its properties. These conditions, known as *property template items* (PTIs), are OCL constraints [Obj14b]. OCL operations provide the ability to describe more complex constraints than equalities between property values and variables. In Listing 5.2, each domain has one pattern. These patterns filter Person objects (with three PTIs) and Employee objects (with two PTIs), 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 the fact that the name of the employee equals the concatenation of the first name and the last name of the person and the fact that both instances 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.

QVT-R relations are defined intensionally. In *checkonly* mode, a relation does not check that metamodel instances are consistent by looking for them in an existing set of pairs of consistent models. It rather evaluates the existence of a value that fulfills all property template items in domain patterns. These patterns can be regarded as predicates. Thus, it is relevant to correlate QVT-R relations and predicate-based consistency relations. One relation in QVT-R can be translated into one or more predicates. The main idea is to extract properties that are bound to the same QVT-R variables: having QVT-R variables in common means that values of these properties are interrelated.

5. Proving Compatibility of Consistency Relations

<pre> import personMM : personmm.ecore'; import employeeMM : employeemm.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}; } } </pre>	<pre> import personMM : personmm.ecore'; import residentMM : residentmm.ecore'; 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}; } } </pre>	<pre> import employeeMM : employeemm.ecore'; import residentMM : residentmm.ecore'; 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}; } } </pre>
--	--	---

Figure 5.16.: Three binary QVT-R transformations forming a consistency specification, based on the relations in Figure 5.1. Taken from [Kla+20a].

Properties are separated to build two tuples of property sets, one for each metamodel. Then, a predicate function is generated by extracting OCL constraints. The triplet that groups these objects together is a predicate. A formal construction of predicates from QVT-R will be presented in the subsequent section.

As a result, QVT-R is a language suitable for writing consistency relations according to our formalism. The decomposition procedure presented in this section treats a set of (binary) QVT-R transformations as a consistency relation set and checks its compatibility. The QVT-R transformations for the example in Figure 5.1 are depicted in Figure 5.16.

5.4.2. Consistency Relations Represented as Graphs

In the subsequent subsections, we introduce a decomposition procedure to prove compatibility, which relies on an algorithmic way to detect redundant consistency relations. We defined the notion of redundancy in Definition 5.9 at the level of classes rather than complete models, like consistency is defined in transformation languages. In predicate-based consistency relations, consistency is explicitly defined for properties of classes through the use of predicates. This is an even more fine-grained notion than the formal one given at the level of classes. It is, however, equivalent, because the relation between two properties of two classes imposes a relation between these two classes containing arbitrary values of all other values of instances of these two classes. Comparing such consistency relations of different transformations to evaluate redundancy is what we call a *redundancy test*.

Consistency specifications induce a graph, which consists of class properties that are related by edges that are labeled with the predicates defining the consistency relations. For our decomposition procedure, we use such a graph representation, as it allows us to apply graph algorithms for determining independent und redundant consistency relations. The decomposition procedure then operates in two phases. First, the decomposition procedure creates this structure out of QVT-R transformations. Then, it refers to consistency relation definitions inside it to detect redundant relations and check compatibility of the consistency specification.

In the following, we first the so called *property graph*, which is the graph induced by consistency relations that brings the graph characterization of transformations at the level of class properties and predicates. Such a structure can be represented as a hypergraph with a labeling.

Definition 5.16 (Property Graph)

Let $\mathbb{CR} = \{CR_i\}_{i=1}^n$ 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 $\mathcal{M} = (\mathcal{H}, l)$, such that $\mathcal{H} = (V_{\mathcal{H}}, E_{\mathcal{H}})$ is a hypergraph and $l: E_{\mathcal{H}} \rightarrow \{\text{TRUE, FALSE}\}^{I_{\mathfrak{P}_{\mathfrak{C}_l}} \times I_{\mathfrak{P}_{\mathfrak{C}_r}}}$ is a hyperedge labeling:

- $V_{\mathcal{H}}$ is the set of vertices, i.e., the set of all properties used in all predicates:

$$V_{\mathcal{H}} = \bigcup_{i=1}^n \bigcup_{\pi \in \Pi_i} P_{\pi}$$

- $E_{\mathcal{H}}$ is the set of hyperedges, i.e., $E_{\mathcal{H}} \subseteq \mathcal{P}(V_{\mathcal{H}}) \setminus \{\emptyset\}$. For a property graph, hyperedges are made up of properties that occur in the same predicate:

$$E_{\mathcal{H}} = \bigcup_{i=1}^n \bigcup_{\pi \in \Pi_i} \{P_{\pi}\}$$

- l is a function that labels each hyperedge with its corresponding predicate function:

$$\forall i \in \{1, \dots, n\}, \forall \pi = (\mathfrak{P}_{\mathfrak{C}_l}, \mathfrak{P}_{\mathfrak{C}_r}, f_{\pi}) \in \Pi_i : l(P_{\pi}) = f_{\pi}$$

The idea behind the property graph is to group properties that participate in the definition of the same predicate. When the consistency relation set is not a tree, some properties may be used in the definition of multiple consistency relations. For example, an employee's name must be consistent with a resident's name and a person's first and last name. When properties are vertices, such groups form hyperedges. For a consistency relation to be redundant, there must be other relations that share properties with it. As a consequence, the property graph is useful to detect independent sets of consistency relations as well as cycles of hyperedges (which form alternative concatenations for redundant relations). Hyperedges only address the structural aspect of

consistency relation definitions. For this reason, each hyperedge is labeled with its corresponding predicate function.

The need for hypergraphs arises from the fact that predicates can relate more than two properties: the predicate in the relation CR_{PE} , which ensures equality of an employee's name and the concatenation of first and last name of a person, contains three properties. The first phase of the procedure is to set up such a structure using QVT-R transformations.

Traversal Order Among Transformations In order to build the property graph, each transformation must be processed to retrieve relations it contains. The decomposition procedure processes one transformation at a time. For compatibility checking purposes, transformations are independent of each other. They can be processed separately and in any order. The reason behind this is that QVT-R relations are not executed on models but only read. Therefore, they are side-effect free.

Traversal Order Inside Transformations A transformation is a set of QVT-R relations. In general, each relationship deals with the consistency of only a small part of each metamodel, for example the consistency between two classes. Unlike QVT-R transformations, QVT-R relations cannot be processed in any order. There are two types of relations: top-level relations and non-top-level relations. Top-level relations are always invoked, whereas non-top-level relations are only invoked in `where` or `when` clauses of other relations through a mechanism similar to a function call. Only relations that would be invoked during the execution of transformations have to be processed for the decomposition, as non-invoked relations cannot cause incompatibilities. To determine a relevant processing order and retain only QVT-R relations that can be invoked, one solution is to create the call graph of the transformation. We impose a restriction on the specification to make this representation easier: relations may only be invoked in `where` clauses. Starting from top-level relations, relations are visited using a depth-first traversal. A relation R_2 can be visited from a relation R_1 if R_1 's `when` clause invokes R_2 . As a result, a relation is only visited if it is top-level or if a relation invoking it was itself visited before.

5. Proving Compatibility of Consistency Relations

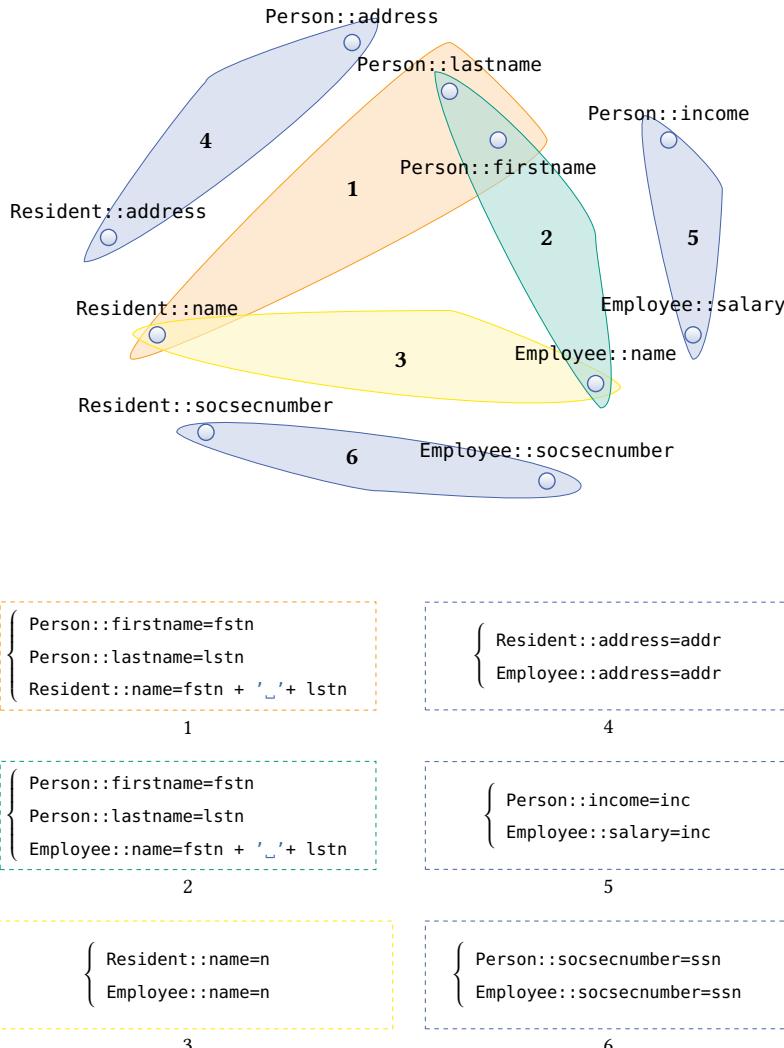


Figure 5.17.: Property graph for the QVT-R example in Figure 5.16 based on the relations in Figure 5.1. Taken from [Kla+20a].

```

relation R {
    [variable declarations]

    domain MM1 a : A {PA,1=eA,1, PA,2=eA,2}
    domain MM2 b : B {PB,1=eB,1}
    [when { PRE }] [where { INV }]
}

```

Listing 5.3.: Structure of a QVT-R relation with property template items. Taken from [Kla+20a].

From QVT-R Relation to Hyperedge At the beginning, the property graph is an empty object. Each time a QVT-R relation is processed, the property graph may get new vertices and a new hyperedge. In accordance with Definition 5.16, hyperedges can be generated from predicates. Therefore, it is relevant to translate each QVT-R relation into a set of predicates. Listing 5.3 depicts the structure of an abstract QVT-R relation between two metamodels MM_1 and MM_2 . Defining a predicate from a QVT-R relation amounts to find important properties for each metamodel and definitions that bind them. Class tuples are composed of classes that occur in each domain, i.e., $\mathfrak{C}_{MM_1} = \langle A \rangle$ and $\mathfrak{C}_{MM_2} = \langle B \rangle$. Each class in each class tuple is associated with a set of property template items. Important properties for the consistency specification are in the left-hand side of each property template item. For example, the property template item $P_{A,1} = e_{A,1}$ indicates that the property $P_{A,1}$ must match the OCL expression $e_{A,1}$ in which there are QVT-R variables. Not all properties are related to each other within the same QVT-R relation. For example, constraints on `Employee.salary` and `Employee.name` are independent, because consistency of one does not depend on consistency of the other. There is a simple criterion in QVT-R to identify interrelated properties. Pattern matching indicates which properties have to be grouped together to build a predicate. If two properties depend on the same QVT-R variable, they are interrelated, because a value assignment must satisfy both property template items. Predicates can then be generated from sets of interrelated properties. OCL expressions can also occur in `when` and `where` clauses. As with relation invocations, we focus on invariants (`where`), which we limit to the manipulation of QVT-R variables, given that properties can be limited to domain patterns without loss of generality. The processing of an invariant is similar to that of property template items: properties that depend on QVT-R variables occurring in the same invariant have to be grouped together.

Algorithm 2. Merge of properties to predicates. Adapted from [Kla+20a].

```
1: procedure MERGECONSISTENCYVARIABLES( $\{\langle\{p\}, V_{\{p\}}\rangle\}$ )
2:   stopMerge  $\leftarrow$  TRUE
3:   entries  $\leftarrow$  [ $\{\langle\{p\}, V_{\{p\}}\rangle\}$ ]
4:   do
5:     stopMerge  $\leftarrow$  TRUE
6:     results  $\leftarrow$  {}
7:     while entries  $\neq []$  do
8:       ref  $= \langle P_{\text{ref}}, V_{P_{\text{ref}}} \rangle \leftarrow \text{entries}[0]$ 
9:       others  $\leftarrow \text{entries}[1 :]$ 
10:      entries  $\leftarrow []$ 
11:      for  $\langle P, V_P \rangle \in \text{others}$  do
12:        if  $V_P \cap V_{P_{\text{ref}}} = \emptyset$  then
13:          entries  $\leftarrow \text{entries} + \langle P, V_P \rangle$ 
14:        else
15:          stopMerge  $\leftarrow$  FALSE
16:          ref  $\leftarrow \langle P \cup P_{\text{ref}}, V_P \cup V_{P_{\text{ref}}} \rangle$ 
17:        end if
18:      end for
19:      results  $\leftarrow \text{results} \cup \{\text{ref}\}$ 
20:    end while
21:    entries  $\leftarrow \text{results}$ 
22:    while  $\neg \text{stopMerge}$ 
23:      return set(entries)
24: end procedure
```

Algorithm 2 formalizes the way properties are grouped to form predicates. At the beginning of the algorithm, each property is associated with QVT-R variables that occur in the corresponding property template item. This association, called an *entry*, is a couple $(\{p\}, V_{\{p\}})$ where $\{p\}$ is a singleton containing the property p and $V_{\{p\}}$ a set of QVT-R variables. The entry of an invariant is composed of variables in it and all properties associated with these variables through property template items. At each iteration, the algorithm chooses a reference entry and merges all other entries with it if

the intersection of their sets of QVT-R variables is nonempty. The algorithm stops when all sets of QVT-R variables are pairwise disjoint.

Example 5.7. *There are five properties in the relation PE of the QVT-R transformation PersonEmployee in Figure 5.16, which can be described with the following entries:*

$$\begin{aligned} & (\{\text{firstname}\}, \{fstn\}), (\{\text{lastname}\}, \{lstn\}), \\ & (\{\text{income}\}, \{inc\}), (\{\text{name}\}, \{fstn, lstn\}), \\ & (\{\text{salary}\}, \{inc\}) \end{aligned}$$

After the execution of the algorithm, properties are merged as follows:

$$\begin{aligned} & (\{\text{firstname, lastname, name}\}, \{fstn, lstn\}), \\ & (\{\text{income, salary}\}, \{inc\}) \end{aligned}$$

This results in two sets of properties.

At the end of the algorithm, each entry can be transformed into a hyperedge. To do so, properties of the entry are assigned to the classes out of which they originate to form property sets. These property sets are grouped into two tuples. The predicate function is the conjunction of all OCL expressions associated with properties of the entry. When all transformations and all QVT-R relations have been processed, the property graph is correctly initialized. It is now invariable, in the sense that the procedure cannot add new vertices or hyperedges to the graph. Only hyperedges identified as redundant can then be removed. Moreover, all the information needed to assess compatibility in the consistency specification is translated into the property graph. There is no need to query metamodels or QVT-R transformations anymore.

5.4.3. Decomposition of Consistency Relations

Given a property graph $\mathcal{M} = (\mathcal{H}, I)$, decomposition is accomplished by removing redundant consistency relations (hyperedges of \mathcal{H}) until all relations have been tested once or until the property graph is only composed of trees. The hypergraph \mathcal{H} provides valuable information about the nature of the consistency specification. First, a necessary condition for a consistency relation between two metamodels M_1 and M_2 to be redundant according to

Definition 5.9 is the existence of an alternative concatenation of relations that links M_1 and M_2 too. In terms of graph, there must exist a path between M_1 and M_2 . Second, consistency relation definitions can be independent from each other, in the sense that they share no properties. Following Theorem 5.5, the union of two independent and compatible consistency relations is also compatible. Independency in the hypergraph is given by connected components. An important aspect of the decomposition procedure is to find consistency relation definitions that can be tested for redundancy. Taking advantage of the structure of the graph is useful for listing these relations.

Consistency Relation Preprocessing As a special kind of hypergraph, a property graph has the advantage of being expressive when it comes to model consistency relation definitions involving multiple properties. The downside is that common graph algorithms (such as graph traversal) become harder to define and to apply. The choice between graphs and hypergraphs is a balance between abstraction and usability. For purposes of implementation, the property graph is replaced by its dual, i.e., a simple graph that is equivalent to it. The dual of the property graph is the graph whose vertices are hyperedges of the property graph. If hyperedges of the property graph share at least one property, their corresponding vertices in the dual are linked. An example for the dual of a property graph is given in Figure 5.18.

Definition 5.17 (Dual of a Property Graph)

Let $\mathcal{M} = (\mathcal{H}, l)$ be a property graph. The dual of the property graph \mathcal{M} , denoted \mathcal{M}^* is a tuple (\mathcal{G}, v, l) with a simple graph \mathcal{G} and two functions v and l such that:

- $V_{\mathcal{G}} = E_{\mathcal{H}}$
- $E_{\mathcal{G}} = \{\{E_1, E_2\} \mid \forall (E_1, E_2) \in E_{\mathcal{H}}^2 : E_1 \cap E_2 \neq \emptyset\}$
- $\forall (E_1, E_2) \in E_{\mathcal{G}} : v(\{E_1, E_2\}) = E_1 \cap E_2$

Each edge $\{E_1, E_2\}$ in the dual is labelled with the set of properties that occur both in E_1 and E_2 . The dual contains all the information necessary to build the property graph again. Given a dual $\mathcal{M}^* = (\mathcal{G}, v, l)$, the property graph $\mathcal{M} = (\mathcal{H}, l)$ can be built by defining $V_{\mathcal{H}} = \bigcup_{V \in V_{\mathcal{G}}} V$ and $E_{\mathcal{H}} = V_{\mathcal{G}}$.

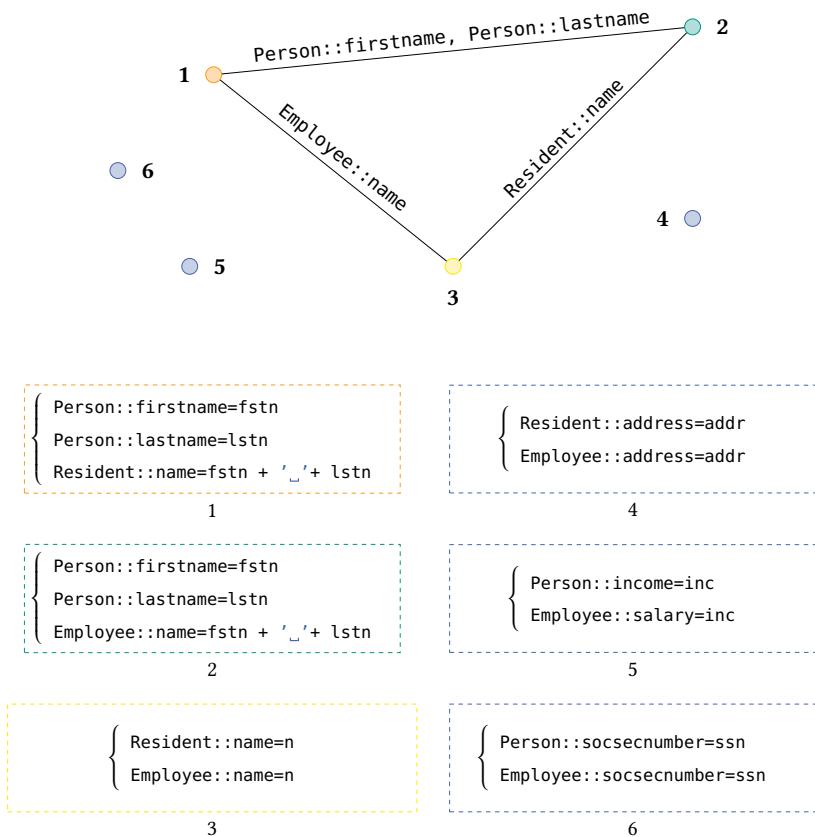


Figure 5.18.: Dual of the property graph for the QVT-R example in Figure 5.16 based on the relations in Figure 5.1. Taken from [Kla+20a].

Independent Subsets of Consistency Relations In a consistency specification, consistency relations can form independent sets, in the sense that the consistency of one set can be checked and repaired independently without affecting the consistency of the other set. This occurs at two levels. First, at the metamodel level, when there exist two sets of metamodels such that no metamodel of one set is bound to a metamodel of the other set through a consistency relation. Second, at the metamodel element level, when two consistency relation sets of the same metamodel relate objects that are inde-

pendent of each other. In terms of consistency relations and predicates, such sets are made up of relations that do not share any property. Independence is characterized in the same way for both levels in the property graph. This results in two subhypergraphs² such that there is no path (i.e., sequence of incident hyperedges) from one to the other. In the dual of the property graph, this results in two subgraphs that are not connected to each other as well. Otherwise, there would exist two consistency relations (one from each subgraph) that share at least one property, thus contradicting the hypothesis of independence. Independent subsets of hyperedges in the property graph can be processed independently, since one's compatibility has no influence on compatibility of the others.

Once the property graph is converted to its dual, the decomposition procedure computes independent subsets of relations. This can be achieved by computing connected components in the dual. Connected components are maximal subgraphs such that there exists a path between any two vertices in it, which conform the notion of independence in our formalization given in Definition 5.5. We use Tarjan's algorithm to compute them in linear time [Tar72]. Incompatibilities can then only occur within a connected component. Therefore, we prove that a consistency relation set is compatible by proving compatibility of every connected component of the dual of the property graph.

Generation of Candidate Relations For a connected component to be compatible, there must be no redundant predicate in it. According to our definition of redundancy (Definition 5.9), we consider a predicate and its representing hyperedge, respectively, redundant if it is subsumed by another concatenation of predicates that considers the same classes at the left side of the relation. In our operationalization, we, in fact, only consider the case in which exactly the same classes are related, thus also the classes at the right side must be equal. In the property graph, this requires the existence of an alternative sequence of hyperedges that relates the same properties as the possibly redundant hyperedge. Note that the existence of an alternative path is a necessary but not a sufficient condition. For instance, a predicate ensuring that two `String` attributes are equal could not be replaced by a

² A subhypergraph of a hypergraph $\mathcal{H} = (V_{\mathcal{H}}, E_{\mathcal{H}})$ is a hypergraph $\mathcal{S} = (V_{\mathcal{S}}, E_{\mathcal{S}})$ such that $E_{\mathcal{S}} \subseteq E_{\mathcal{H}}$ and $V_{\mathcal{S}} = \bigcup_{E \in E_{\mathcal{S}}} E$

sequence of predicates only ensuring that these strings have the same length. That is why the possibly redundant predicate and the alternative path of predicates are subject to a redundancy test, which we explain later (see Subsection 5.4.4).

Given that connected components are independent, a predicate can only be replaced by other predicates in the same component. Moreover, Theorem 5.6 also applies to the dual of the property graph: once the component is a tree, i.e., there are no two paths relating the same properties or classs, respectively, it is inherently compatible. As a result, the dual proves compatibility of the consistency specification if it is only composed of independent trees. Such a graph is called a *forest*.

In the property graph, an alternative path for a hyperedge E (i.e., a predicate) is a sequence of pairwise incident hyperedges such that the first and the last are also incident to E . Hyperedges of the property graph become vertices in the dual. Therefore, an alternative path for a predicate E in the dual is characterized by a cycle that contains E . If the vertex sequence of such a cycle is $\langle E, E_1, \dots, E_n, E \rangle$, the alternative path is $\langle E_1, \dots, E_n \rangle$. Ultimately, the generation of candidate relations for a redundancy test amounts to the enumeration of pairs $(E, \langle E_i \rangle)$, where E is a possibly redundant predicate (i.e., a hyperedge in the property graph and a vertex in its dual) and $\langle E_i \rangle$ is an alternative sequence of predicates that may replace E . There may be multiple alternative paths for a given predicate in the dual of the property graph, hence the need to find multiple cycles. The problem of finding all simple cycles in an undirected graph is called *cycle enumeration*.

The cycle enumeration algorithm used in the decomposition procedure relies on a *cycle basis*. In an undirected graph, a cycle basis is a set of simple cycles that can be combined to generate all other simple cycles of the graph. The cycle basis is first computed using Paton's algorithm [Pat69]. For a given predicate, the enumeration processes each cycle from the cycle basis and merges it with all yet processed cycles. In the context of the decomposition procedure, a cycle must also go through the predicate to analyze. Algorithm 3 is a slightly modified version of Gibb's algorithm to enumerate simple cycles in an undirected graph using a cycle basis [Gib69]. In this algorithm, 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 were yet processed. Merged with cycles of the basis $base_1, \dots, base_n$, these

Algorithm 3. Enumeration of alternative paths. Adapted from [Kla+20a].

```
1: procedure ENUMERATEFOUNDCYCLES(Dual  $\mathcal{M}^*$ ,  $pred \in V_{\mathcal{M}^*}$ )
2:    $\{base_1, \dots, base_n\} \leftarrow PATONALGORITHM(\mathcal{M}^*)$ 
3:    $foundCycles \leftarrow \{base_1\}$ 
4:    $currentCycles \leftarrow \emptyset$ ,  $currentCycles^* \leftarrow \emptyset$ 
5:   for  $base \in \{base_2, \dots, base_n\}$  do
6:     for  $foundCycle \in foundCycles$  do
7:        $newCycle \leftarrow foundCycle \oplus base$ 
8:       if  $foundCycle \cap base \neq \emptyset$  then
9:          $currentCycles \leftarrow currentCycles \cup \{newCycle\}$ 
10:      else
11:         $currentCycles^* \leftarrow currentCycles^* \cup \{newCycle\}$ 
12:      end if
13:    end for
14:    // Remove non-simple cycles from  $currentCycles$ 
15:    for  $cycle_1, cycle_2 \in currentCycles$  do
16:      if  $cycle_2 \subset cycle_1$  then
17:         $currentCycles \leftarrow currentCycles \setminus \{cycle_1\}$ 
18:         $currentCycles^* \leftarrow currentCycles^* \cup \{cycle_1\}$ 
19:      end if
20:    end for
21:    // New valid cycles are in  $currentCycles \cup \{base\}$ 
22:    for  $cand \in currentCycles \cup \{base\}$  do
23:      if  $pred \in cand \wedge REPLACEHYPEREDGE(pred, cand)$  then
24:        remove  $pred$  and its incident edges from  $\mathcal{M}^*$ 
25:        break
26:      end if
27:    end for
28:     $foundCycles \leftarrow foundCycles \cup currentCycles$ 
29:     $foundCycles \leftarrow foundCycles \cup currentCycles^* \cup \{base\}$ 
30:     $currentCycles \leftarrow \emptyset$ ,  $currentCycles^* \leftarrow \emptyset$ 
31:  end for
32: end procedure
```

linear combinations are used to merge more than two cycles of the basis. At each iteration of Algorithm 3, processing a new cycle from the cycle basis *base*, new simple cycles are in *currentCycles* $\cup \{\text{base}\}$. Edge-disjoint or non-simple cycles are stored in *currentCycles*^{*}. Note that redundancy tests can be performed as new cycles are generated, as shown on line 20. For the given predicate *pred*, it is checked whether one of the newly generated cycles is redundant, i.e., it contains the predicate *pred* and *pred* can be replaced by the concatenation of the other edges. By doing so, it is not necessary to store all cycles and wait for the end of the algorithm before starting redundancy tests. More interestingly, if the redundancy test is positive for one alternative sequence of predicates, there is no need to test others. The initial predicate can be removed and the algorithm can be used with another possibly redundant predicate.

Stopping Criterion The decomposition procedure stops when each predicate has been tested once. Note that if the connected component becomes a tree after a few removals of predicates, then the last tests of remaining predicates are trivial. As there are no more cycles in the dual of the connected component, no redundancy test is performed.

In this subsection, we have presented an algorithm for proving compatibility of relations in a consistency specification written in QVT-R. We defined property graphs and their dual as a representation of consistency relations and explained how they can be derived from a specification in QVT-R. We discussed how a consistency relation tree manifests in such a representation and how candidates for redundancies in connected components of such a representation can be found by computing a cycle basis. Based on Theorem 5.5 and Theorem 5.6, as well as Corollary 5.12, this algorithm is able to prove compatibility by removing redundant relations, such that the resulting network is a composition of independent trees. However, we still need to discuss how redundancy of relations, in terms of redundant predicates in the property graph, can be identified, according to the REPLACE-HYPEREDGE method in Algorithm 3. We will discuss that in the subsequent section.

5.4.4. Redundancy Check for Consistency Relations

In the decomposition procedure, enumerating possibly redundant predicates and proving that such predicates are redundant are uncoupled tasks. The

latter task can be regarded as a black box embedded in the former one: only the result of the redundancy test matters. As a consequence, the decomposition procedure allows the use of various strategies to prove that a predicate is redundant. This is the reason why we have provided the decomposition on its own in the previous subsections and used REPLACE-HYPEREDGE as a black-box method in the Algorithm 3 for decomposition.

Due to the limitations on the decidability of OCL expressions used in QVT-R, there is no perfect strategy to validate redundancy. In this thesis, we opt for a strategy that allows the decomposition procedure to be fully automated. We first discuss how predicates can be compared to prove redundancy. Then, an approach that translates OCL constraints (in predicates) to first-order logic formulae and uses an automated theorem prover is introduced. Finally, the translation rules from OCL to first-order logic are presented along with their limitations.

5.4.4.1. Intensional Comparison of Predicates

Whatever the strategy, the *redundancy test* takes a couple $(E, \langle E_1, \dots, E_n \rangle)$ as an input and returns TRUE if the predicate E was proven redundant because of the sequence of predicates $\langle E_1, \dots, E_n \rangle$, FALSE otherwise. As stated in Definition 5.9, a consistency relation is considered left-equal redundant if its removal from the set of consistency relations leads to an equivalent set and relates the same kinds of elements at the left side. For the relation to be redundant, there must be an alternative sequence of relations that already fulfills the role of the initial relation. This also applies to the property graph: for a predicate to be removed, there must exist another sequence of predicates relating the same properties that is strict enough not to weaken the consistency specification. Weakening the consistency specification means allowing models that would have been considered non-consistent before the removal of the predicate. Since we only consider predicates that relate the same properties the additional requirement of left-equal redundancy in comparison to general redundancy in Definition 5.8 is always fulfilled. In the following, we thus only discuss redundancy rather than left-equal redundancy, as it is always given by construction. As illustrated in Figure 5.19, a predicate E can only be removed if all instances matching the predicate also match predicates $\langle E_1, \dots, E_n \rangle$.

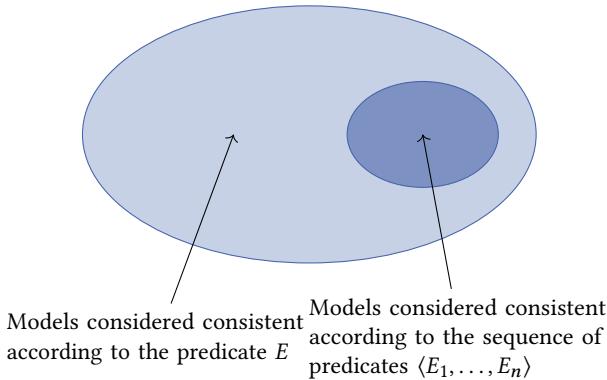


Figure 5.19.: Redundancy of a hyperedge: the hyperedge E is redundant, because all instances valid according to E are already valid according to the alternative sequence E_1, \dots, E_n of hyperedges. Taken from [Kla+20a].

Therefore, a redundancy test is equivalent to the comparison of two sets of instances. However, a predicate may be fulfilled by infinitely many model elements. For example, the predicate ensuring that the income of a person and the salary of an employee are equal is valid for infinitely many integer pairs. It is impossible to compare these sets element per element (i.e., extensionally). Since consistency relations in the decomposition procedure are defined intensionally, by means of predicates, anyway, the redundancy test compares sets in their intensional specification. As a result, the redundancy test uses the description of the possibly redundant predicate and the candidate alternative sequence of predicates to decide whether the predicate is redundant. In QVT-R, predicates are expressed as OCL constraints. As part of the construction of the property graph, these constraints are already represented as hyperedge labels. That is, comparing predicate definitions in the decomposition procedure amounts to perform a static analysis of these labels and QVT-R relation conditions (when and where clauses). In order to prove redundancy, the static analysis has to rely on a rigorous framework to reason about OCL constraints. This is provided by various formal methods in the field of software verification.

5.4.4.2. Encoding Redundancy in First-Order Formulae

In this thesis, the strategy we use to prove redundancy is based on first-order logic, a well-suited and expressive mathematical language for decision procedures. The idea is to set up a first-order formula that is valid (i.e., true under every interpretation) if and only if the redundancy test is positive. To this end, the formula embeds OCL expressions (translated into first-order logic as well) contained in predicates. Then, a theorem prover evaluates the validity of the formula.

The choice of first-order logic is motivated by the nature of OCL: there exists a general translation of OCL into first-order logic [BKS02]. This result was later refined in [BCG05] in order to show that OCL formulae are essentially full first-order formulae. What this means is that OCL does not form a fragment of first-order logic and needs all its expressiveness. First-order logic being undecidable in general, this also implies that not all redundant formulae can be proven valid. Therefore, the results obtained by the decomposition procedure are highly dependent on the performance of the theorem prover. Even with quantifiers and variables, the gap between programming languages and first-order sentences remains significant. OCL is composed of arithmetic operations, strings, arrays, etc. The meaning of language constructs must also be integrated into formulas. To this effect, it is possible to replace binary variables in first-order formulae with a Boolean sentence expressed in a given theory. Theories use all kinds of objects such as strings, floats, sequences, etc. With theories, the satisfiability problem equates to assign values to variables in first-order sentences such that the evaluation of sentences makes the whole formula TRUE. For instance, the formula $(a \times b = 10) \wedge (a + b > 0)$ is satisfiable given the assignment $\{a = 2, b = 5\}$. This extension is known as *satisfiability modulo theories* (SMT). First-order formulae for the SMT problem are called *SMT instances*. Some theorem provers come with built-in theories, they are thus called *theory-based theorem provers*. In the context of the decomposition procedure, we translate constructs in OCL constraints with corresponding constraints into built-in theories of the prover. By doing so, the mapping between OCL and first-order logic is easier to achieve.

Modeling as a Horn Clause For any two models, consistency depends on condition elements in predicate-based consistency relations, which themselves depend entirely on property values for which predicate functions

evaluate to TRUE. As a result, redundancy can be tested by comparing descriptions of predicate functions. This information is contained in the input of the redundancy test. Let $\pi = (\mathfrak{P}_{\mathfrak{C}_l}, \mathfrak{P}_{\mathfrak{C}_r}, f_\pi)$ 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_π .

In terms of predicate functions, the predicate E can be replaced by a sequence of predicates $\langle E_1, \dots, E_n \rangle$ under the following condition: for any set of models, f_E evaluates to TRUE whenever $f_{E_1} \wedge \dots \wedge f_{E_n}$ evaluates to TRUE. If this condition is met, the consistency specification is neither strengthened nor weakened after the removal of E . The specification is not strengthened because the removal of a predicate can only allow more sets of models to be consistent. It is not weakened either because all property values that match $\wedge f_{E_i}$ also match E . As a consequence, E is redundant. That is, solutions of $\wedge f_{E_i}$ form a subset of solutions of E . This is consistent with Figure 5.19. The redundancy test can be encoded as a formula in the following way:

$$(f_{E_1} \wedge \dots \wedge f_{E_n}) \Rightarrow f_E$$

The formula above is called a *Horn clause*. Horn clauses form an important fragment of logic in the field of automated reasoning. Terms on the left-hand side of the clause are called *facts*, whereas the term on the right-hand side is called *goal*. The implication represents the deduction of the goal from the facts. 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, then the alternative sequence of predicates can replace the initial predicate whatever the instantiation of metamodel elements (i.e., whatever the models).

One last detail is to be taken into consideration for the translation of predicate functions. Horn clauses are usually described without quantifiers. In a Horn clause, all variables are implicitly universally quantified. Given that predicate functions are made up of OCL expressions, they contain local QVT-R variables. Consistency depends upon pattern matching, i.e., the existence of a valid assignment of variables. Therefore, QVT-R variables in the goal clause have to be existentially quantified.

Example 5.8. Figure 5.18 depicts the dual of the property graph derived from the motivational example in Figure 5.1. The dual contains four connected

5. Proving Compatibility of Consistency Relations

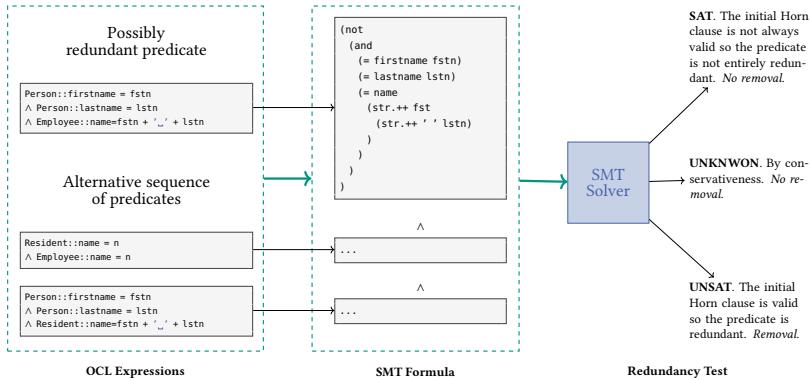


Figure 5.20.: Overview of the redundancy test, from OCL expressions to the SMT solver. Taken from [Kla+20a].

components, including three with one predicate only. Compatibility is already proven for these three components because they are trivial trees. The other component is made up 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 in which 3 is the goal and 1 and 2 are the facts:

$$\begin{aligned}
 & [(Person::firstname = f1) \wedge (Person::lastname = l1) \\
 & \quad \wedge (Resident::name = f1 + " " + l1)] \\
 & \wedge [(Person::firstname = f2) \wedge (Person::lastname = l2) \\
 & \quad \wedge (Employee::name = f2 + " " + l2)] \\
 \Rightarrow & (\exists n : (Resident::name = n) \wedge (Employee::name = n))
 \end{aligned}$$

QVT-R variables have been renamed to avoid conflicts. This is necessary because they are no longer isolated as they were before in two distinct QVT-R relations. According to the SMT solver, the formula above is valid. Therefore, predicate 3 can be removed from the property graph and its dual. There are then only two predicates left in this component. It is inherently compatible. As all independent subsets of predicates are compatible, the consistency specification is compatible.

Redundancy Test Redundancy can be proven by checking that the Horn clause derived from predicate functions is valid (i.e., true under every interpretation). Because the whole decomposition procedure is automated, the theorem prover, namely an SMT solver, embedded in the procedure is also automated. The solver takes an SMT instance as an input and answers whether it is satisfiable insofar as possible. Proving that a Horn clause H is valid is actually equivalent to proving that its negation $\neg H$ is unsatisfiable. Therefore, we prove that the SMT instance $f_{E_1} \wedge \dots \wedge f_{E_n} \wedge \neg f_E$ is unsatisfiable. The SMT solver can provide three possible outcomes:

Satisfiable. If $\neg H$ is satisfiable, then H is not valid. This means that 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, then H is valid. When the alternative sequence is fulfilled, so is the predicate. It is redundant and can be removed.

Unknown. First-order logic being undecidable, not all formulae can be proven valid. When a theorem prover is unable to evaluate the satisfiability of a formula, it returns *Unknown*. By application of the conservativeness principle, the redundancy test is considered negative. As a result, the predicate is not removed.

5.4.4.3. Translation

Translation refers to the process of mapping OCL expressions of QVT-R relations to SMT instances. In the context of the decomposition procedure, this is facilitated by the fact that QVT-R uses a subset of OCL called *EssentialOCL* [Obj16a], a side-effect free sublanguage that provides primitives data types, data structures and operations to express constraints on models. Many constructs of OCL have a direct equivalent in theories of the theorem prover. More complex constructs can often be mapped through the combination of primitive constructs. Note that there also exist constructs that cannot be translated. Language constructs of SMT solvers are described using the SMT-LIB specification, a standard that provides among others an input language for solvers [BFT17]. This language uses a syntax similar to that of Common Lisp. The translation is recursive: each OCL expression depends

OCL Data Type	Ecore Data Type	SMT Data Type
Integer	EInt	IntSort
Real	EDouble	RealSort
Boolean	EBoolean	BoolSort
String	Estring	StringSort
UnlimitedNatural	EInt	IntSort (<i>without infinity</i>)

Table 5.1.: Mapping between primitive types representations. Taken from [Kla+20a].

on the translation of its subexpressions. A complete reference of translated constructs has been developed in the master’s thesis of Pepin [Pep19].

Primitive Data Types OCL defines five primary data types: integers, reals, booleans, strings and unlimited naturals. These data types are mapped with Ecore when parsing QVT-R transformations. The mapping between Ecore data types and Z3 data types (called *sorts*) is described in Table 5.1. It is straightforward, except for *UnlimitedNatural*, a data type to represent multiplicities. *UnlimitedNatural* and *Integer* are different in that the former can take an infinite value whereas the latter cannot. IntSort in Z3 cannot be infinite. In this case, a workaround is to represent an *UnlimitedNatural* as a couple (IntSort, BoolSort) where the value equals ∞ if the Boolean is TRUE.

Data Structures Primitive data structures in OCL are called *collections*. There are four types of collections based on the combination of two features, the first defining whether elements are ordered and the second whether duplicate elements are allowed. Among those four collection types, two are currently supported: sequences (ordered, duplicates allowed) and sets (non-ordered, no duplicates). In QVT-R, collections are used either as literals or as types for a special kind of properties: role names.

Collection Literals Collection literals are OCL expressions that represent data structures with constant and predefined values (e.g., Sequence{1, 4, 9} or Set{2, 5}). In SMT solving, the fundamental theory to represent a collection of values is the theory of *arrays*. Arrays are maps that relate a set

of indexes (domain) and a set of values (codomain). They are immutable, purely functional data structures. Unlike OCL data structures, there is no notion of size in arrays. To overcome this limitation, we translate collections to algebraic data types in the SMT input language. Data types are composed of an array (collection values) and an integer storing the collection size. It is noteworthy that collection literals rarely occur in consistency relations. In general, collections are groups of objects resulting from references in metamodels.

Collections from Role Names In QVT-R property template items, properties are either attributes (like `Person::name`) or role names. A role name is an alias for objects at the end of the reference owned by the class of the pattern. If the upper bound of the reference multiplicity is greater than one (e.g., `0..*`), then the role name may represent a collection of objects. The nature of the collection depends on whether the end is ordered or unique or both. Even if the content of the collection is unknown, it is possible to reason about role names by means of the theory of *uninterpreted functions* (UF). A role name r of a class c can be represented as a function of c (e.g., $r(c)$). By abstracting the semantics of functions, uninterpreted functions help to reason about model elements without knowing all their details. For example, two role names are equal if both belong to objects that have been proven to be equal themselves.

Operations OCL also provides many operations on primitive data types and data structures, such as arithmetic operations or string operations. Following the object-oriented structure of OCL, every operation has a source and zero or more arguments. For example, the `+` operation denotes an addition when the source is an integer but a concatenation when the source is a string. We translated operations regarding arithmetics, booleans, conversion operators, equality operators, order relations, collections and strings [Pep19].

Some OCL operations are said to be *untranslatable*, because it is impossible to find a mapping between them and features of state-of-the-art SMT solvers. As a result, there are QVT-R relations that cannot be processed by the decomposition procedure. For instance, the string operations `toLower` and `toUpper` cannot be easily translated without numerous user-defined axioms in current SMT solvers. Although decision procedures for such a case exist [Vea+12], they are not yet integrated into solvers.

Summary

In this subsection, we have presented an approach to evaluate redundancy of a predicate in a property graph (respectively its dual) for the decomposition procedure, also depicted in Figure 5.20. The approach translates the OCL expressions of predicates into logic formulae and generates Horn clauses for a potentially redundant predicate and its alternative predicates. If an SMT solver proves unsatisfiability of that clause, the checked predicate is redundant and can be removed.

5.5. Summary

In this chapter, we have discussed the challenge regarding compatibility of consistency relations, which are encoded in transformations. We have derived a well-founded notion of compatibility, precisely formalized this notion and presented a formal approach that is able to validate compatibility of given relations. The approach is proven correct. Based on the formal approach, we developed a practical approach for QVT-R, which is able to validate compatibility of consistency relations defined in QVT-R and OCL. We conclude this section with the following central insight.

Insight 2 (Compatibility)

Transformations that are supposed to preserve contradictory consistency relations easily lead to problems when combining them to a network, because (some of) 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 virtually 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 only analyzed for a given transformation network.

6. Constructing Synchronizing Transformations

Transformations are the central artifacts of which a transformation network is composed. We have introduced them as *synchronizing transformations* in Definition 4.6, which are combinations of consistency relations with a consistency preservation rules 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 (cf. Definition 4.7). 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. Even if transformation languages allow bidirectional specifications, they still derive unidirectional consistency preservation rules from such a specification, such as forward and backward transformation (which may be incremental or not) derived from TGGs 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 transformation, 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.

Ordinary transformations as synchronizing ones

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 be used in a transformation network. This chapter thus 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 to ordinary transformations to be synchronizing; 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?

Benefits due to reusability

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 reduces the complexity, because the definition of two unidirectional consistency preservation rules is 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.

Publication of contributions

We have already published parts of the contributions in this chapter in [Kla18] and [Kla+19]. The identification of essential issues when constructing synchronizing transformations from ordinary transformations defined in existing transformation languages has already been discussed in [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 published that work in [Kla+19] and present the results especially in Section 6.4.

6.1. Deriving the Gap to Ordinary Transformation

We have already introduced that there is both a formal and 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 most 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 5.2.

Gap
between
synchroniz-
ing and
ordinary
transfor-
mations

6.1.1. Behavior of Ordinary Transformations in Networks

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 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 new class creation of the transformation from PCM to Java.

Failure in
motiva-
tional
example

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. If we now consider ordinary transformations with unidirectional consistency preservation rules, they 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

Failure in
running
example

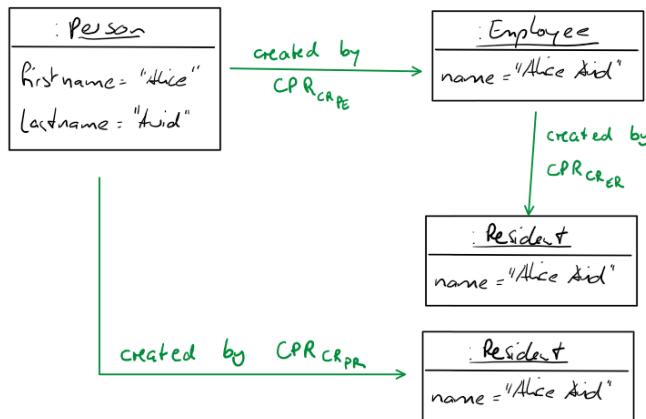


Figure 6.1.: Duplicate creation of a resident by two sequences of consistency preservation rules.

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 transformations 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 both CPR_{CRER} as well as CPR_{CRPR} 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.

Avoidance
of failures

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 look for the existence of elements to create first. This can either be done by using a trace model, which most existing transformation language use to store corresponding elements, or by searching for an appropriate element in the other model. Using a trace model, however, has some drawbacks and pitfalls, which we will investigate later.

6.1.2. 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 4.4, a unidirectional consistency preservation rule does only receive changes made to one of the two models and returns changes to the other model instead of receiving and returning changes to both.

Unidirectional
preservation
rules

Definition 6.1 (Unidirectional Consistency Preservation Rule)

Let M_1, M_2 be two metamodels and let \mathbb{CR} be a set of consistency relations between elements of those metamodels. A *unidirectional consistency preservation rule* $CPR_{\mathbb{CR}}$ for the relation set \mathbb{CR} is a (usually partial) function:

$$CPR_{\mathbb{CR}} : (I_{M_1}, I_{M_2}, \Delta_{M_1}) \rightarrow \Delta_{M_2} \cup \{\perp\}$$

This is how the consistency preservation rules defined in or derived from 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. If the rule is not able to handle the given changes, it may return \perp .

Preserva-
tion rules in
transfor-
mation
languages

In addition, they usually assume the input models to be consistent and then ensure that applying the input and the output changes to the models, the resulting models are consistent again. Since they are defined for consistent input models, their behavior in case of inconsistent models is undefined, either returning \perp or a change, which does not necessarily guarantee that the models when applying the input and output changes are consistent. 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 in [Ste10]. This is even compliant to the correctness notion that we defined for synchronizing consistency preservation rules in Definition 4.5. Thus, we define correctness of such a unidirectional consistency preservation rule as follows.

Correctness
of unidirec-
tional
preserva-
tion
rules

Definition 6.2 (Unidirectional Preservation Rule Correctness)

Let CPR_{CR} be a unidirectional consistency preservation rule. We call CPR_{CR} *correct* if the resulting models when applying the input and output changes are consistent to CR again:

$$\begin{aligned} \text{CPR}_{\text{CR}} \text{ correct} :&\Leftrightarrow \forall m_1 \in I_{M_1}, m_2 \in I_{M_2}, \delta_{M_1} \in \Delta_{M_1} : \\ &(\langle m_1, m_2 \rangle \text{ consistent to } \text{CR} \\ &\wedge \exists \delta_{M_2} \in \Delta_{M_2} : \delta_{M_2} = \text{CPR}_{\text{CR}}(m_1, m_2, \delta_{M_1})) \\ &\Rightarrow \langle \delta_{M_1}(m_1), \delta_{M_2}(m_2) \rangle \text{ consistent to } \text{CR} \end{aligned}$$

Optional
partiality of
synchroniz-
ing
preserva-
tion
rules

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 4.4, because there may be changes to both models which cannot be processed reasonably as one the changes may need to reverted to achieve consistency. Ignoring that practical requirement, it is theoretically possible to always return changes that, if applied to the input models, produce consistent models. Those returned changes may perform arbitrarily unreasonable modifications, but still restore consistency.

Mandatory
partiality of
unidirec-
tional
preserva-
tion
rules

In general, unidirectional consistency preservation rules need to be partial. This is due to the reason that there can be models for which no other models can be generated such that they are consistent with respect to a set of consistency relations. Consider the consistency relations $CR = \{\langle a, z \rangle, \langle b, z \rangle\}$ 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 the consistency relation CR requires for a and b an element z to exist in the other model. The consistency relation 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 4.18). In consequence, a unidirectional consistency preservation rule cannot produce a result for such an input without violating the correctness definition and thus be partial. For that reason, unidirectional consistency preservation rules necessarily need to be partial, whereas this requirement was optional for synchronizing consistency preservation rules.

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, m_2 and 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 $\text{C}\mathbb{R}$ for a set of consistency relations $\text{C}\mathbb{R}$. 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 that restores consistency can be theoretically 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.

Undefined behavior

6.1.3. Unidirectional Relations and Preservation

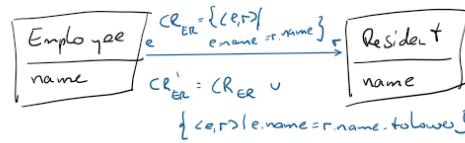
Defining unidirectional consistency preservation rules based on a unidirectional notion of consistency relations imposes the idea of having one unidirectional consistency preservation rule associated with one unidirectional consistency relation, or at least a set of unidirectional relations between the same two metamodels. In consequence, we would have two sets of unidirectional consistency relations between two metamodels and a consistency preservation rule for each of them.

Unidirectional preservation rules for unidirectional relations

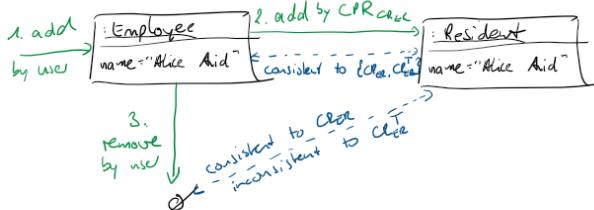
It is, however, easy to see that a unidirectional consistency preservation rule cannot only consider one direction of consistency relations, but needs to consider both. Consider the example in Figure 6.2, which contains an extract of the consistency relations of the running example. We assume the consistency 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 $\text{CPR}_{CR_{ER}}$ 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 show that this is not the case. While for the scenario of adding an employee the rule properly propagates the change by adding a resident and thus restores consistency, removing an employee leads to a violation of consistency. Removing an employee does not require the consistency preservation rule to perform any changes in the resident model, because CR_{ER} only requires a unique resident

Both relation directions for unidirectional preservation rule

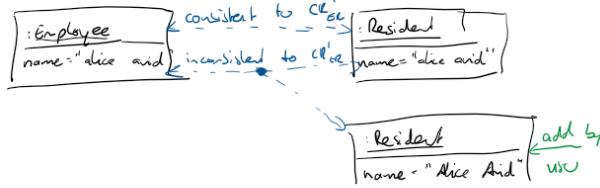
6. Constructing Synchronizing Transformations



1. Removing Employee with CR'_{ER} only for CR_{ER}



2. Adding a Resident, effect to CR'_{ER}



3. Removing an Employee, effect to CR'_{ER}

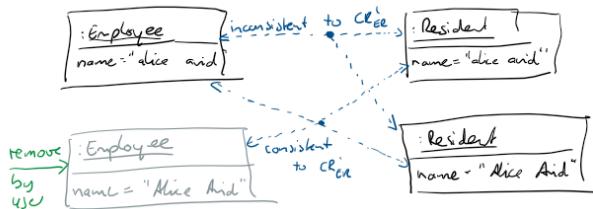


Figure 6.2.: Nonalignment of unidirectional relations and preservation rules.

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 .

The given scenario exemplifies the general case that a 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 according 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 relation 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.

We can see that any 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 the other 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 the other 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 name “alice avid” and a corresponding resident with the same name, the models are consistent to that relation. Adding a resident with name “Alice Avid” violates CR'_{ER} , because the employee “alice avid” 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

Both
relation
directions
affected by
one change

Change
type
relevance
for relation
direction

another element that occurs in a consistency relation pairs 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 avid” and “Alice Avid”, the models are consistent because each employee has a corresponding resident and vice versa. If we remove the employee “Alice Avid”, 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 Avid” and “Bob Bright”. 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 synchronizing 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, such that 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 will therefore discuss relaxed requirements to unidirectional consistency preservation rules to be able to act like a synchronizing transformation.

6.1.4. Bidirectional Transformations

A unidirectional consistency preservation rule does usually not appear on its own but in combination with another rule for the opposite direction. We have already 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. In practice, many transformation languages, especially relational ones such as QVT-R or TGG tools, allow the specification of *bidirectional transformations*, which means that they define or derive unidirectional consistency preservation rules for both directions.

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 model can be updated to restore consistency. A synchronizing transformation according to Definition 4.6 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.

Unidirectional
preservation
rules
for both
directions

Bidirectional
transformations

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 $CPR_{\mathbb{CR}}^{\rightarrow}$ and $CPR_{\mathbb{CR}}^{\leftarrow}$ be unidirectional consistency preservation rules with:

$$CPR_{\mathbb{CR}}^{\rightarrow} : (I_{M_1}, I_{M_2}, \Delta_{M_1}) \rightarrow \Delta_{M_2} \cup \{\perp\}$$

$$CPR_{\mathbb{CR}}^{\leftarrow} : (I_{M_2}, I_{M_1}, \Delta_{M_2}) \rightarrow \Delta_{M_1} \cup \{\perp\}$$

A *bidirectional transformation* is a triple $t = \langle \mathbb{CR}, CPR_{\mathbb{CR}}^{\rightarrow}, CPR_{\mathbb{CR}}^{\leftarrow} \rangle$.

We call such a bidirectional transformation correct if both consistency preservation rules are correct, i.e., they both preserve consistency according to the underlying consistency relation set.

Definition 6.4 (Bidirectional Transformation Correctness)

Let $t = \langle \mathbb{C}\mathbb{R}, \text{CPR}_{\mathbb{C}\mathbb{R}}^{\rightarrow}, \text{CPR}_{\mathbb{C}\mathbb{R}}^{\leftarrow} \rangle$ be a bidirectional transformation. We call t correct if, and only if, $\text{CPR}_{\mathbb{C}\mathbb{R}}^{\rightarrow}$ and $\text{CPR}_{\mathbb{C}\mathbb{R}}^{\leftarrow}$ are both correct according to Definition 6.2.

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, if possible. 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

Some existing work already proposed strategies to synchronize concurrent changes between two models. For example, some work proposes techniques for processing concurrent changes with TGGs [Her+12; OPN20], whereas others define specific algorithms for a general notion of synchronizing transformations according to our given definition [Xio+13; Xio+09]. All these approaches, however, deal with the more general case that any changes may have been made to the models. That especially includes conflicting updates by one or more user, which need to be resolved. Such a resolution may lead to the necessity of reverting one of the changes.

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 being able to find a consistent result for specific changes. This assumption 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 Sub-section 6.1.3 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. There are, however, different options how to apply the rules, each having different benefits and drawbacks.

Conflicting concurrent changes

Specific concurrent changes in transformation networks

Assumed compatibility of relations

Execution of both preservation rules

Independent execution and merge

We have already 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 to them δ_{M_1} and δ_{M_2} . If we apply the two unidirectional consistency preservation rules independently, we get $\delta'_{M_2} = \text{CPR}_{\text{CR}}^{\rightarrow}(m_1, m_2, \delta_{M_1})$ and $\delta'_{M_1} = \text{CPR}_{\text{CR}}^{\leftarrow}(m_2, m_1, \delta_{M_2})$. 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 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 rather than $\delta_{M_1}(m_1)$. An example may be that δ_{M_1} removes an element from m_1 , which δ'_{M_1} changes. Even if the change is still defined for that changed model, the result may not be consistent because the necessary change produced by $\text{CPR}_{\text{CR}}^{\rightarrow}$ cannot be applied anymore. Thus simply merging the individually produced changes does not necessarily lead to a consistent result.

Sequential execution

Another option is to sequence the execution, thus first generating $\delta'_{M_2} = \text{CPR}_{\text{CR}}^{\rightarrow}(m_1, m_2, \delta_{M_1})$ as before. $\langle \delta_{M_1}(m_1), \delta'_{M_2}(m_2) \rangle$ is consistent due to correctness of $\text{CPR}_{\text{CR}}^{\rightarrow}$. Afterwards, we apply the second consistency preservation rule to the the new consistent models and the original change to m_2 , thus $\delta'_{M_1} = \text{CPR}_{\text{CR}}^{\leftarrow}(\delta'_{M_2}(m_2), \delta_{M_1}(m_1), \delta_{M_2})$. 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 CR. This means that δ_{M_2} is not applied to m_2 anymore, to which the changes were made originally, but needs to be applied to $\delta'_{M_2}(m_2)$. It is again unclear whether the change can still be applied to that state, i.e., whether δ_{M_2} is defined for $\delta'_{M_2}(m_2)$, like in the merge example above. If, however, the changes are applicable, then we have the guarantee that all original changes are reflected, as they were applied to the models, and that the resulting models are consistent, because of the correctness of the consistency preservation rules.

Sequential execution with less drawbacks

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 give a guarantee that if changes can be concatenated the resulting models are consistent. Thus, we do not pursue that approach anymore. In the following, we further discuss the second option of sequencing the execution of the consistency preservation rules.

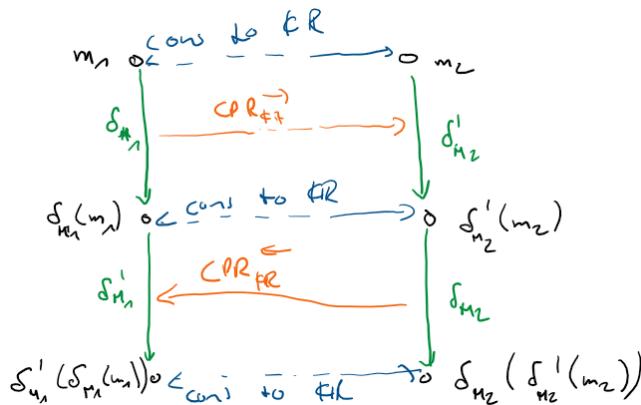


Figure 6.3.: Schema for sequencing unidirectional consistency preservation rules after concurrent changes.

6.2.2. Sequencing of Consistency Preservation Rules

The sequential application of original changes and execution of consistency preservation rules as explained in the previous section 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 $\text{CPR}_{\text{CCR}}^{\rightarrow}$ produces changes for the second model that affect a disjoint set of elements in CCR 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 even not make a difference in which order they are applied then.

Changes affecting disjoint element sets

Unfortunately, the changes δ'_{M_2} produced by $\text{CPR}_{\text{CCR}}^{\rightarrow}$ and the original ones produced by other transformations δ_{M_2} do not necessarily affect disjoint sets of elements. In that case, two problems can occur:

Issues where sequencing preservation rule application

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 $\text{CPR}_{\text{CCR}}^{\rightarrow}$. This can, for example, happen when δ'_{M_2} removes an element that is affected by

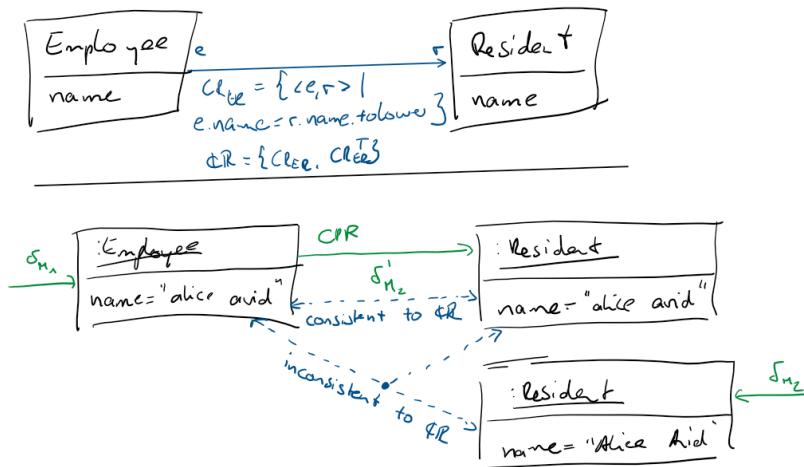


Figure 6.4.: Example for non-transformability when sequencing the application of unidirectional consistency preservation rules and concurrent changes.

δ_{M_2} . Due to the change of the element in δ_{M_2} , it is part of a condition element in another transformation that was executed before. Since CPR_{CR} removed that element, the condition element exists no longer anyway, thus this removal has to be propagated back by the transformation that originally introduced the change in 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, in fact, all changes in δ_{M_2} can be 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 changes δ_{M_2} can be applied to $\delta'_{M_2}(m_2)$, this does still not guarantee that CPR_{CR} is able to process the given changes. In fact, this requirement applies to all changes, thus even original user changes, but there are special circumstances in this situation that make it 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 $\text{CR} = \{\text{CR}_{ER}, \text{CR}_{ER}^T\}$. An employee "alice avid" is added by the original change to m_1 . The con-

sistency preservation rule then generates an appropriate resident with the same name to fulfill the consistency relation. Applying the original change to m_2 then leads to the addition of resident “Alice Avid”, which was generated by another transformation, e.g., the one that created an appropriate person and changed the capitalization of the name. Now it is impossible for $\text{CPR}_{\text{CR}}^{\leftarrow}$ 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 simply performing no additional changes, as the originally added resident is already consistent to the originally added employee. In consequence, if the unidirectional consistency preservation rule would have 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 consistency preservation cannot find a result, whereas 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 already was 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 second model was already modified, such that the given models are not initially consistent, to reflect the changes that already have been made and integrate them into consistency preservation. This means that 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 assume the consistency preservation rules to produce results for inputs that are not consistent. We have already given examples where it is not possible to restore consistency by one unidirectional consistency preservation rule after changes in both models.

Non-transformal
changes of
users un-
avoidable

Necessity to
process in-
consistent
inputs

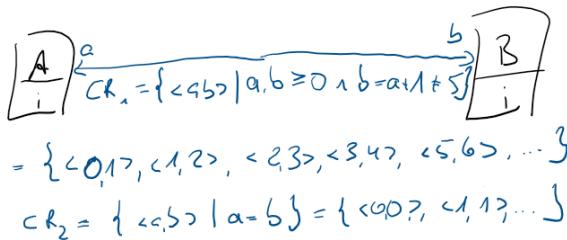


Figure 6.5.: A consistency relation requiring multiple executions of unidirectional consistency preservation rules to find a consistent result.

Relaxed
notion
affecting
number of
executions

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 that directly affects the requirements we have to define.

6.2.3. Execution Bounds

Unidirectional
preservation
rules
cannot
always be
correct

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, which 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 $CPR_{\mathbb{CR}}^{\rightarrow}$ for propagating changes in models with instances of A to one with instances of B and $CPR_{\mathbb{CR}}^{\leftarrow}$ to propagate changes in the opposite direction.

Example
scenario

Consider the following scenario: If an A with $i = 0$ is added to an empty model, $CPR_{\mathbb{CR}}^{\rightarrow}$ 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 $CPR_{\text{CR}}^{\rightarrow}$ cannot modify the model with instances of A, it is impossible for $CPR_{\text{CR}}^{\rightarrow}$ 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 $CPR_{\text{CR}}^{\leftarrow}$ adds an A with $i = 1$ in response to the previous execution of $CPR_{\text{CR}}^{\rightarrow}$, all consistency relations except CR_1 are fulfilled. $CPR_{\text{CR}}^{\rightarrow}$ can then create a B with $i = 2$, which is iteratively processed by $CPR_{\text{CR}}^{\leftarrow}$. This process terminates as soon as $CPR_{\text{CR}}^{\rightarrow}$ adds a B with $i = 5$, as then CR_1^T is also fulfilled, because it does not require a corresponding A for a B with $i = 5$.

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 execution of consistency preservation rules can be arbitrarily high. The value of 5 in CR_1 of the example can be exchanged by an arbitrary high value requiring an arbitrary high number of executions. This may only be circumvented if we required that $CPR_{\text{CR}}^{\rightarrow}$ must perform changes such that $CPR_{\text{CR}}^{\leftarrow}$ can then restore consistency with a single execution. In our scenario this would mean that $CPR_{\text{CR}}^{\rightarrow}$ adds all B with $i \leq 5$. Anyway, such a behavior requires a relaxation of the correctness requirement for consistency preservation rules, because $CPR_{\text{CR}}^{\rightarrow}$ 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 yet. In that case, both the partial condition element has to be completed in one model and the corresponding condition element in the other model have to be created. Thus changes in both models are required, which can only be achieved by executing both consistency preservation rules and accepting that the first executed one does not return consistent models. An example for such a scenario could be the consistency relation between components in PCM and their realization as a package and component in Java. It may be desired that as soon as a package at a specific place, e.g., a “components” package, or with a specific name, e.g., containing “Component”, is created in the Java code, this is identified as a component, thus creating the component in the PCM model as well as 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

Multiple executions leading to consistent result

Arbitrary high number of necessary executions

Preservation rules complete partial condition elements

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, which is then corrected by the second consistency preservation rule.

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, 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 sections, 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 it can be required that consistency preservation rules need to be executed multiple times. This is obviously necessary to make bidirectional transformation synchronizing, thus we will especially consider

Necessity of
termination
guarantee

Processing
inconsis-
tent input
models

Guarantee-
ing
termination

how to achieve execution bounds, such that 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 such that they also accept models that are not initially consistent. We can then not require them to restore consistency between the models 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 partial consistency. We derive requirements to the transformations and finally show that transformations fulfilling these requirements terminate consistently.

Consistency preservation rules for inconsistent models

6.3.1. Partial Consistency of Models

Given two models m_1 and m_2 and a change δ_{M_1} and δ_{M_2} to each of them, a unidirectional consistency preservation rule $CPR_{\mathbb{C}\mathbb{R}}^{\rightarrow}$ 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 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, $CPR_{\mathbb{C}\mathbb{R}}^{\rightarrow}$, even if correct according to Definition 6.2, does not guarantee that applying the delivered change delivers 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 fulfill some kind of consistency notion.

Changed models partially consistent

Such a notion of partial consistency may be defined in two different ways. First, we may say that two models only fulfill a extract of the consistency relations. Second, we may say that only extracts of two models fulfill the consistency relations.

Options for partial consistency definition

Fulfilling
subsets of
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. To circumvent that, we would need to define a notion of partial consistency relations, which allows us to define that models are consistent to parts of consistency relations. Such a notion would have to be defined at the level of consistency relation pairs and their condition elements within the consistency relations. It would, however, not make sense to consider subsets of consistency relations, i.e., only a subset of their consistency relation pairs, because when analyzing consistency of two models those consistency relation pairs are not independent. This is because 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 consistent 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. These interdependencies of consistency relation pairs are the reason why consistency to partial consistency relations does not provide insights on the reasons for models being inconsistent, which is why we do not consider this as our notion for partial consistency.

Parts of
models
fulfilling
relations

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 of them 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. It is based on the idea to find 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 $m_2 \in I_{M_2}$, we define their *maximal consistent subsets* $m_1^p \subseteq m_1$ and $m_2^p \subseteq m_2$ with regards to \mathbb{CR} as the subsets of m_1 and m_2 that are consistent and larger than all other consistent subsets:

$$\begin{aligned} \langle m_1^p, m_2^p \rangle &\text{ consistent to } \mathbb{CR} \wedge m_1^p \subseteq m_1 \wedge m_2^p \subseteq m_2 \\ &\wedge (\forall m_1^{p'} \in \mathcal{P}(m_1), m_2^{p'} \in \mathcal{P}(m_2) : \\ &\quad \langle m_1^{p'}, m_2^{p'} \rangle \text{ consistent to } \mathbb{CR} \Rightarrow |m_1^{p'}| + |m_2^{p'}| \leq |m_1^p| + |m_2^p|) \end{aligned}$$

We define partial consistency of two models with respect to \mathbb{CR} as the ratio between the size of the maximal consistent subsets and the size of the models in $\text{CONS}_{\mathbb{CR}}$:

$$\begin{aligned} \text{CONS}_{\mathbb{CR}} : (I_{M_1}, I_{M_2}) &\rightarrow [0, 1] \\ (m_1, m_2) &\mapsto \frac{|m_1^p| + |m_2^p|}{|m_1| + |m_2|} \end{aligned}$$

Such maximal consistent subsets do always exist. In the extreme case, when models are not consistent in any way, it is $m_1^p = m_2^p = \emptyset$, because empty models are always consistent by definition. In that case, partial consistency of the models is 0. In cases when models are actually consistent, the maximal consistent subsets are the models themselves, which is why partial consistency is 1.

Extremes of
partial
consistency

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

Resolving
partial
inconsisten-
cies

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 $t = \langle \mathbb{C}\mathbb{R}, \text{CPR}_{\mathbb{C}\mathbb{R}}^{\rightarrow}, \text{CPR}_{\mathbb{C}\mathbb{R}}^{\leftarrow} \rangle$ be a bidirectional transformation for metamodels M_1 and M_2 . An *execution step* Ex_t^1 of t is a function:

$$\text{Ex}_t^1 : (I_{M_1}, I_{M_2}, \Delta_{M_1}) \rightarrow (I_{M_1}, I_{M_2}, \Delta_{M_1}) \cup \{\perp\}$$

$$(m_1, m_2, \delta_{M_1}) \mapsto \begin{cases} (m'_1, m'_2, \delta'_{M_1}) \\ \perp \end{cases}$$

with:

$$\delta'_{M_2} := \text{CPR}_{\mathbb{C}\mathbb{R}}^{\rightarrow}(m_1, m_2, \delta_{M_1}) \quad m'_1 := \delta_{M_1}(m_1)$$

$$\delta'_{M_1} := \text{CPR}_{\mathbb{C}\mathbb{R}}^{\leftarrow}(m_2, m'_1, \delta'_{M_2}) \quad m'_2 := \delta'_{M_2}(m_2)$$

If either consistency preservation rule is undefined for the input, i.e., $\text{CPR}_{\mathbb{C}\mathbb{R}}^{\rightarrow}(m_1, m_2, \delta_{M_1}) = \perp$ or $\text{CPR}_{\mathbb{C}\mathbb{R}}^{\leftarrow}(m_2, m'_1, \delta'_{M_2}) = \perp$, then the execution is undefined, i.e., $\text{Ex}_t^1(m_1, m_2, \delta_{M_1}) = \perp$.

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. To these resulting models and the resulting change the execution step can be reapplied.

Algorithm 4. Execution of a bidirectional transformation.

```

1: procedure EXECUTE( $t = \langle \text{CCR}, \text{CPR}_{\text{CR}}^{\rightarrow}, \text{CPR}_{\text{CR}}^{\leftarrow} \rangle, m_1, m_2, \delta_{M_1} \rangle$ )
2:   if  $\neg(\langle m_1, m_2 \rangle \text{ consistent to } \text{CCR})$  then
3:     return  $\perp$ 
4:   end if
5:   while  $\neg(\langle \delta_{M_1}(m_1), m_2 \rangle \text{ consistent to } \text{CCR})$  do
6:      $\text{executionResult} \leftarrow \text{Ex}_t^1(m_1, m_2, \delta_{M_1})$ 
7:     if  $\text{executionResult} = \perp$  then
8:       return  $\perp$ 
9:     else
10:       $(m_1, m_2, \delta_{M_1}) \leftarrow \text{executionResult}$ 
11:    end if
12:   end while
13:   return  $\langle \delta_{M_1}(m_1), m_2 \rangle$ 
14: end procedure

```

The execution of a bidirectional transformation then consists of the consecutive application of execution steps until the delivered models are consistent, as defined in Algorithm 4. Although we, theoretically, require the consistency preservation rules to be able to handle initial models that can be arbitrarily inconsistent, it will not be possible to define such rules in practice. Since we follow a delta-based notion of consistency preservation, we will therefore 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 and produce new changes that all introduced inconsistencies can be resolved. In contrast to our initial definition of consistency preservation rules, it is still not necessary that consistency is restored with a single execution of a consistency preservation rule. When finally coming to the synchronization scenario, in which both models have been modified, this will even not be possible anymore.

Without loss of generality, we defined bidirectional transformation execution and the individual 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, just by swapping $\text{CPR}_{\text{CR}}^{\rightarrow}$ and $\text{CPR}_{\text{CR}}^{\leftarrow}$. Since we finally consider the case that

Execution of bidirectional transformation

Direction of execution step

both models have been changed, it is not relevant for us which change to consider first anyway.

6.3.3. Transformation Execution Termination

It is obvious that the algorithm does only return \perp if an execution step of the transformation cannot be applied. Additionally, we can easily show that in all other cases, if the algorithm terminates, it returns consistent models.

Lemma 6.1 (Bidirectional Transformation Execution Consistency)

If Algorithm 4 terminates, it either returns \perp or a consistent model pair.

Proof. Algorithm 4 can terminate by one of its two return statements in Line 8 and Line 13. Line 8 returns \perp , which fulfills the lemma statement. Line 13 returns the model pair $\langle \delta_{M_1}(m_1), m_2 \rangle$. The only possibility to achieve this statement is the termination of the previous while loop. Since the loop can only terminate due to the other return statement or the non-fulfillment of the loop condition, we can only reach this statement by non-fulfillment of the loop condition. Since the negation of the loop condition is $\langle \delta_{M_1}(m_1), m_2 \rangle$ consistent to $\mathbb{C}\mathbb{R}$, the result fulfills the lemma statement. \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 either no further execution step can be applied, i.e., it returns \perp , or it delivers consistent models. 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 $t = \langle \text{CIR}, \text{CPR}_{\text{CIR}}^{\rightarrow}, \text{CPR}_{\text{CIR}}^{\leftarrow} \rangle$ be a bidirectional transformation for metamodels M_1 and M_2 . We say that t is *partial-consistency-improving* if, and only if, an execution step does always improve partial consistency by reducing the size of the models or improving the size of the maximal consistent subsets. So for all inputs, for which the execution step of t does not return \perp , i.e.,

$$(m'_1, m'_2, \delta'_{M_1}) := \text{Ex}_t^1(m_1, m_2, \delta_{M_1})$$

we denote $\delta_{M_1}(m_1)^p$ and m_2^p as the maximal consistent subsets of $\langle \delta_{M_1}(m_1), m_2 \rangle$ and $\delta'_{M_1}(\delta_{M_1}(m_1))^p$ and $\delta'_{M_2}(m_2)^p$ as the ones of $\langle \delta'_{M_1}(\delta_{M_1}(m_1)), \delta'_{M_2}(m_2) \rangle$ and 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):

$$\begin{aligned} & |\delta'_{M_1}(\delta_{M_1}(m_1))^p| + |\delta'_{M_2}(m_2)^p| - |\delta_{M_1}(m_1)^p| - |m_2^p| \\ & > |\delta'_{M_1}(\delta_{M_1}(m_1))| + |\delta'_{M_2}(m_2)| - |\delta_{M_1}(m_1)| - |m_2| \end{aligned}$$

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

Partial consistency improvement as intuitive expectation

Practical necessity of partial consistency improvement

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 requirements in Definition 6.7 to be appropriate for practical transformation definition. They even represent a weaker notion than what we want to achieve in practice, because the requirement is only based on the sizes of the models and their maximal consistent subsets but not the actual contents. In practice, the consistent subsets before transformation execution will be a subset of those after transformation execution, 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 be able to have a unique subset that can be calculate from each model state and 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 does also ensure that the execution steps are strongly monotone regarding partial consistency, it does not ensure that a partial consistency value 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, for example, be achieved by adding consistent elements in each step, which do not affect the existing elements. Then the size of both the maximal consistent subsets as well as the models themselves increases by the same value, thus partial consistency improves but never reaches 1.

Lemma 6.2 (Bidirectional Transformation Execution Termination)

Let $t = \langle \mathbb{C}\mathbb{R}, CPR_{\mathbb{C}\mathbb{R}}^{\rightarrow}, CPR_{\mathbb{C}\mathbb{R}}^{\leftarrow} \rangle$ be a partial-consistency-improving bidirectional transformation. Then Algorithm 4 does always terminate.

Proof. The while loop of the algorithm consecutively applies an execution step of the bidirectional transformation t . The algorithm terminates when at some point a return statement is executed, thus either an execution step cannot be executed and returns \perp , 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 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 $\mathbb{C}\mathbb{R}$ if, and only if, their partial consistency is 1, i.e., $\text{Cons}_{\mathbb{C}\mathbb{R}}(\delta_{M_1}(m_1), m_2) = 1$. 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. We start with the initial size difference:

$$\text{sizeDifference}_0 = |\delta_{M_1}(m_1)| + |m_2| - |\delta_{M_1}(m_1)^P| + |m_2^P|$$

It is $\text{sizeDifference}_0 \geq 0$, 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 models m_1^i , m_2^i and change $\delta_{M_1}^i$. Then we have the size differences before that iteration, i.e., the difference after iteration $i - 1$, and after that iteration, as:

$$\begin{aligned} \text{sizeDifference}_{i-1} &= |\delta_{M_1}^{i-1}(m_1^{i-1})| + |m_2^{i-1}| - |\delta_{M_1}^{i-1}(m_1^{i-1})^P| + |m_2^{i-1,P}| \\ \text{sizeDifference}_i &= |\delta_{M_1}^i(m_1^i)| + |m_2^i| - |\delta_{M_1}^i(m_1^i)^P| + |m_2^{i,P}| \end{aligned}$$

If we consider the reduction of the size difference in the i -th iteration, this is given by:

$$\begin{aligned} \text{sizeDifferenceReduction}_i &= \text{sizeDifference}_i - \text{sizeDifference}_{i-1} \\ &= |\delta_{M_1}^i(m_1^i)| + |m_2^i| - |\delta_{M_1}^i(m_1^i)^P| + |m_2^{i,P}| \\ &\quad - (|\delta_{M_1}^{i-1}(m_1^{i-1})| + |m_2^{i-1}| - |\delta_{M_1}^{i-1}(m_1^{i-1})^P| + |m_2^{i-1,P}|) \end{aligned}$$

We know that $\text{sizeDifferenceReduction}_i > 0$, because t is partial-consistency-improving. Because of the model sizes being natural numbers, we even know that:

$$\text{sizeDifferenceReduction}_i \geq 1$$

So we can calculate the remaining size difference in the i -th iteration by applying all size difference reductions starting from sizeDifference :

$$\begin{aligned}\text{sizeDifference}_i &= \text{sizeDifference}_0 - \sum_{k=1}^i \text{sizeDifferenceReduction}_k \\ &\leq \text{sizeDifference}_0 - \sum_{k=1}^i 1 = \text{sizeDifference}_0 - i\end{aligned}$$

This implies that:

$$i > \text{sizeDifference}_0 \Rightarrow \text{sizeDifference}_i \leq 0$$

In fact, we ignored for reasons of simplicity that as soon as $\text{sizeDifference}_i = 0$, then $\text{sizeDifferenceReduction}_k = 0$ ($k > i$), because sizeDifference_i cannot be less than 0. In consequence, we know that after at most sizeDifference_0 loop iterations, we have $\text{sizeDifference}_i = 0$ for all $i > \text{sizeDifference}_0$ and thus consistent models after that iteration. Since $0 \leq \text{sizeDifference}_0 < \infty$, the algorithm leaves the loop after a finite number of iterations. \square

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 transformation 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, and thus also one execution step of the transformation, 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 now define a process for integrating changes to both involved models to

Termination
of partial-
consistency-
improving
transforma-
tions

Partial
consistency
improve-
ment to
integrate
two
changes

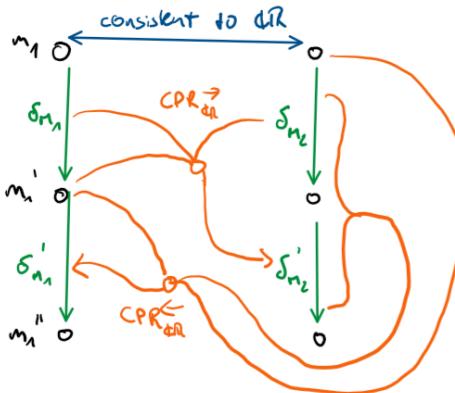


Figure 6.6.: Operation of a synchronizing bidirectional transformation execution step.

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 will discuss that 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 discussed approach is, theoretically, able to process changes to models that are initially arbitrarily inconsistent. For practical applicability, we restricted the approach to the case that the initial models are consistent and inconsistency is introduced by the given change to one of the models. The transformation than iteratively improves partial consistency until consistent models are delivered.

Since we want to consider the case that both of the 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, which is able to process changes to both models. The operation of that execution step is depicted in Figure 6.6. To this end, the first executed consistency preservation rule is applied to the first model

Transformation
property for
processing
inconsis-
tent
inputs

Synchroniz-
ing
transforma-
tion
execu-
tion
step

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 unmodified 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 way, we ensure that all inconsistencies are introduced by changes that are processed by the consistency preservation rules, which was our requirement for practical applicability due to the necessity only to react to changes instead of processing arbitrarily inconsistent models states.

Definition 6.8 (Synchronizing Bidirectional Execution Step)

Let $t = \langle \mathbb{C}\mathbb{R}, \text{CPR}_{\mathbb{C}\mathbb{R}}^{\rightarrow}, \text{CPR}_{\mathbb{C}\mathbb{R}}^{\leftarrow} \rangle$ be a bidirectional transformation for metamodels M_1 and M_2 . A *synchronizing execution step* SyncEx_t^1 of t is a function:

$$\begin{aligned} \text{SyncEx}_t^1 : (I_{M_1}, I_{M_2}, \Delta_{M_1}, \Delta_{M_2}) &\rightarrow (I_{M_1}, I_{M_2}, \Delta_{M_1}) \cup \{\perp\} \\ (m_1, m_2, \delta_{M_1}, \delta_{M_2}) &\mapsto \begin{cases} (m'_1, m'_2, \delta'_{M_1}) \\ \perp \end{cases} \end{aligned}$$

with:

$$\begin{aligned} \delta'_{M_2} &= \text{CPR}_{\mathbb{C}\mathbb{R}}^{\rightarrow}(m_1, \delta_{M_2}(m_2), \delta_{M_1}) & m'_1 &= \delta_{M_1}(m_1) \\ \delta'_{M_1} &= \text{CPR}_{\mathbb{C}\mathbb{R}}^{\leftarrow}(m_2, m'_1, \delta'_{M_2} \circ \delta_{M_2}) & m'_2 &= \delta'_{M_2}(\delta_{M_2}(m_2)) \end{aligned}$$

If either consistency preservation rule is undefined for the input, i.e., $\text{CPR}_{\mathbb{C}\mathbb{R}}^{\rightarrow}(m_1, \delta_{M_2}(m_2), \delta_{M_1}) = \perp$ or $\text{CPR}_{\mathbb{C}\mathbb{R}}^{\leftarrow}(m_2, m'_1, \delta'_{M_2} \circ \delta_{M_2}) = \perp$, then the execution is undefined, i.e., $\text{SyncEx}_t^1(m_1, m_2, \delta_{M_1}) = \perp$.

Single
execution
of synchro-
nization
step

The synchronizing bidirectional execution step is necessary to first integrate the changes made in both models. It is defined such that it only produces a change in the first model, such that afterwards ordinary transformation execution steps that only need to deal with a change to one model have to be applied. This leads to Algorithm 5 for the execution of a synchronizing bidirectional transformation.

Algorithm 5. Synchronizing execution of a bidirectional transformation.

```

1: procedure EXECUTESYNC( $t = \langle \text{CCR}, \text{CPR}_{\text{CCR}}^{\rightarrow}, \text{CPR}_{\text{CCR}}^{\leftarrow} \rangle, m_1, m_2, \delta_{M_1}, \delta_{M_2}$ )
2:   if  $\neg(\langle m_1, m_2 \rangle$  consistent to  $\text{CCR}$ ) then
3:     return  $\perp$ 
4:   end if
5:   if  $\neg(\langle \delta_{M_1}(m_1), m_2 \rangle$  consistent to  $\text{CCR}$ ) then
6:      $\text{executionResult} \leftarrow \text{SYNCEx}_t^1(m_1, m_2, \delta_{M_1}, \delta_{M_2})$ 
7:     if  $\text{executionResult} = \perp$  then
8:       return  $\perp$ 
9:     else
10:       $(m_1, m_2, \delta_{M_1}) \leftarrow \text{executionResult}$ 
11:    end if
12:   end if
13:   while  $\neg(\langle \delta_{M_1}(m_1), m_2 \rangle$  consistent to  $\text{CCR}$ ) do
14:      $\text{executionResult} \leftarrow \text{Ex}_t^1(m_1, m_2, \delta_{M_1})$ 
15:     if  $\text{executionResult} = \perp$  then
16:       return  $\perp$ 
17:     else
18:        $(m_1, m_2, \delta_{M_1}) \leftarrow \text{executionResult}$ 
19:     end if
20:   end while
21:   return  $\langle \delta_{M_1}(m_1), m_2 \rangle$ 
22: end procedure

```

This algorithm has the same properties as the one for the non-synchronizing case given in Algorithm 4. It does always terminate and either returns \perp or a consistent pair of models.

Theorem 6.3 (Synchronizing Transformation Termination)

Let $t = \langle \text{CCR}, \text{CPR}_{\text{CCR}}^{\rightarrow}, \text{CPR}_{\text{CCR}}^{\leftarrow} \rangle$ be a partial-consistency-improving bidirectional transformation. Then Algorithm 5 does always terminate and either returns \perp or a consistent model pair.

Consistent
termination
of synchro-
nizing
transfor-
mation
execu-
tion

Proof. The algorithm is identical to Algorithm 4, except for Lines 5–12, which add the initial synchronization step. These lines add a single return statement

that can return \perp . Since the return statement not returning \perp is still preceded by the while loop having the loop condition that the model pair needs to be inconsistent, the argument of the proof for Lemma 6.1 regarding non-synchronizing bidirectional transformations ensuring that only consistent models are returned still applies. Thus, we know the algorithm either returns \perp or a consistent model pair.

Termination of the algorithm is guaranteed for the same reasons as for non-synchronizing bidirectional transformations as proven in Lemma 6.2. Although the additional execution of SYNCEx 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, finally, the model pair becomes consistent. \square

We have proven that we can execute a bidirectional transformation that is partial-consistency-improving for two given models and changes to both of them, such that consistent models are delivered, as long as the transformation is able to process the changes. In fact, we have already restricted the algorithm such that it must not be able to deal with arbitrarily inconsistent models, but only with models that are initially consistent, such that only the given changes introduce inconsistencies. This is supposed to make it easier to define transformation in practice that fulfill the property of being partial-consistency-improving, 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 can define *synchronizing bidirectional transformations* as bidirectional transformations with the property of being partial-consistency-improving.

Definition 6.9 (Synchronizing Bidirectional Transformation)

Let t be a partial-consistency-improving bidirectional transformation.
Then we call t a *synchronizing bidirectional transformation*.

As we have already discussed at the end of Subsection 6.3.2, we defined bidirectional transformation execution steps starting with $CPR_{\mathbb{C}\mathbb{R}}^{\rightarrow}$, although

inconsisten-
cies only
introduced
by changes

Synchroniz-
ing
bidirec-
tional
transfor-
mations

Order of
consistency
preserva-
tion
rules

it may also be necessary to start with $\text{CPR}_{\mathbb{C}\mathbb{R}}^{\leftarrow}$ if a change was performed in the second model. We discussed that our definitions are without loss of generality and can be directly transferred by swapping the rules. For the synchronization case, in which both models have been modified, it does, theoretically, not even make a difference which of the consistency preservation rules is executed first, because changes to both models are present anyway. From a practical perspective, it can, however, make sense to define one of the consistency preservation rules as the one to always be executed 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, 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 (cf. Definition 4.6), whose single consistency preservation rule accepts two consistent models and a change to each of them and return two changes that, if applied to the models, result in consistent models again. Synchronizing bidirectional transformations, i.e., the yet defined transformations, which are composed of two 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 simply concatenating all changes calculated by the transformation execution steps. For reasons of simplicity, we omitted that in the formal description.

Although synchronizing transformations and synchronizing bidirectional transformations have the same requirements to their inputs and provide the same guarantees regarding consistency for their output, both may also return \perp to indicate that they were not able to calculate changes that lead to consistent models. While a synchronizing transformation can be defined such that it never returns \perp by defining a consistency preservation rule that is a total function, 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

Signatures
of synchronizing
and synchronizing
bidirectional
transformations

Expressiveness
of synchronizing
and synchronizing
bidirectional
transformations

equal expressiveness, i.e., they can always return the same results for the same inputs.

Theorem 6.4 (Synchronizing Transformation Expressiveness)

Synchronizing bidirectional transformations have equal expressiveness than synchronizing transformations, 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 t be a synchronizing bidirectional transformation with:

$$\text{EXECUTESYNC}(t, m_1, m_2, \delta_{M_2}) = (m'_1, m'_2)$$

Then we define the consistency preservation rule CPR of a synchronizing transformation as:

$$\begin{aligned} \text{CPR}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) &= (m_1, m_2, \delta'_{M_1}, \delta'_{M_2}) \\ \text{with } \delta'_{M_1}(\delta_{M_1}(m_1)) &:= m'_1 \wedge \delta'_{M_2}(\delta_{M_2}(m_2)) := m'_2 \end{aligned}$$

Per definition, applying the resulting changes to the input models, the synchronizing transformation delivers for every possible input the same result by applying 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 CPR be the consistency preservation rule of a synchronizing transformation with:

$$\text{CPR}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) = (m_1, m_2, \delta'_{M_1}, \delta'_{M_2})$$

Then we can define the two unidirectional consistency preservation rules of the synchronizing transformation t as follows.

$$\begin{aligned} \text{CPR}^{\rightarrow}(m_1, \delta_{M_2}(m_2), \delta_{M_1}) &= \delta_{M_2}^b \text{ with } \delta_{M_2}^b(\delta_{M_2}(m_2)) := \delta'_{M_2}(m_2) \\ \text{CPR}^{\leftarrow}(m_2, \delta_{M_1}(m_1), \delta_{M_2}^b \circ \delta_{M_2}) &= \delta_{M_1}^b \text{ with } \delta_{M_1}^b(\delta_{M_1}(m_1)) := \delta'_{M_1}(m_1) \end{aligned}$$

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 SYNCEx_t^1 according to Definition 6.8 those changes that are necessary to produce exactly the results of the consistency preservation rule CPR of the synchronizing transformation. Then according to the behavior of SYNCEx_t^1 , we have:

$$\begin{aligned} \text{SYNCEx}_t^1(m_1, m_2, \delta_{M_1}, \delta_{M_2}) &= (m_1^s, m_2^s, \delta_{M_1}^s) \text{ with} \\ \delta_{M_1}^s(m_1^s) &= \delta_{M_1}^s(\delta_{M_1}(m_1)) = \text{CPR}^{\leftarrow}(m_2, \delta_{M_1}(m_1), \delta_{M_2}^b \circ \delta_{M_2})(\delta_{M_1}(m_1)) \\ &= \delta_{M_1}^b(\delta_{M_1}(m_1)) = \delta'_{M_1}(m_1) \\ \wedge m_2^s &= \text{CPR}^{\rightarrow}(m_1, \delta_{M_2}(m_2), \delta_{M_1})(\delta_{M_2}(m_2)) \\ &= \delta_{M_2}^b(\delta_{M_2}(m_2)) = \delta'_{M_2}(m_2) \end{aligned}$$

So SYNCEx_t^1 produces m_1^s , m_2^s and $\delta_{M_1}^s$ for which we know that $\delta_{M_1}^s(m_1^s)$ and m_2^s 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 t according to Algorithm 5 terminates after the conditional statement in Line 5 with the same consistent models that are delivered the applying the changes calculated by the consistency preservation rule CPR of the assumed synchronizing transformation.

With these construction approaches in both directions, 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 approach in the proof does not reflect a practical construction approach for the unidirectional consistency preservation rules of a synchronizing bidirectional transformation. In prac-

tice, 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 in theory it would possible.

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 imposes 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 actual transformation languages. In contrast to only being correct, as it is commonly required from 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. Although 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 that 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 transformation 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 networks 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 most relevant problem scenario that transformation developers have to deal with when developing synchronizing bidirectional transformations.

Systematic avoidance of synchronization problems

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, 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, removals or changes, helps to identify different relevant scenarios, it is easy to see that actually the modification of condition elements of the consistency relations rather than individual elements is relevant.

Inconsistency introduced by changes

If we process a change δ_{M_1} to model m_1 , and m_2 was changed by δ_{M_2} as well, a consistency preservation rule CPR^{\rightarrow} from M_1 to M_2 of a synchronizing bidirectional transformation t produces a change δ'_{M_2} in the execution of the synchronizing execution step $SYNCEx_t^1$. If we assume that δ_{M_1} performs a change that introduces a new condition element, 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 be restored to 1. 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 rather than 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

No case distinction by models changes types

reference. In fact, it must only be considered that a condition element was added, but not which kind of change introduced it.

We already discussed in Subsection 6.1.3 that the addition, removal and change of condition elements are the relevant scenarios that can lead to the violation of consistency. 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 now represent another condition element of the same condition and, thus, are still part of the same consistency relation. We then usually require the consistency preservation rule to update the corresponding condition element appropriately.

That behavior is what consistency preservation rules are actually supposed to implement. A bidirectional transformation with such consistency preservation rules is inherently supposed to fulfill the property of being partial-consistency-improving, because the elements, which have no corresponding elements due to the modification, 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, such that the size of maximal consistent subsets improves.

What may prevent a transformation from being partial-consistency-improving is that, in addition to the above considerations, the addition or removal of a condition element to improve consistency affects further condition elements. This may be due to the reason that 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 elements of the added condition element together

Case
distinction
fpr
condition
element
change
types

Intuitive
behavior
ensuring
partial
consistency
improve-
ment

Interference
of condition
elements

with other existing elements form a condition element of any consistency relation, this may introduce an inconsistency if no corresponding element already exists, 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. It, however, 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, then 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 or 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 defined for identifying consistency. If elements get 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.

Conflicts
between
condition
element
changes

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 already have 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, the corresponding element must necessarily exist and thus can be removed. If the corresponding condition element is not present because it was removed from the other model already, the element can 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 now 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 is removed, the other consistency preservation rule will remove the changed condition element anyway to restore consistency. Thus nothing has to be done and the consistency preservation rule must only consider that the corresponding element may have been removed. If the corresponding element was changed, which is identified by the trace model that still contains a trace link to a changed element, the corresponding element must be adapted such that it reflects the given change. The modification to the corresponding element will then be propagated back by the consistency preservation rule in the opposite direction.

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 condition elements are not present anymore, the first scenario requires an approach to find out whether corresponding elements already exist or not. While existing corresponding elements can be retrieved from the trace model, there are no trace links for these elements yet. In the following, we discuss an approach how to find such corresponding elements.

Problematic scenario is duplicate addition of condition 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 that model may not have been modified as well, these elements cannot already exist. In the synchronization case, however, the change to the other model may have already introduced those elements, thus it is necessary to find them to avoid a duplicate creation.

In [Kla+19], we have proposed a strategy to identify such corresponding elements. Transformation languages usually use trace models to store the information 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 might 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, for example, a further Java class. Resolving the existing trace links transitively is, however, not a solution. In this case, there exists a unique one-to-one mapping 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 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 of transformation and modular reuse. Even worse, it could also be the case that another third class is required by the consistency

Identification of corresponding elements

Trace links

Transitive trace links

Semantics of trace links

information
for
identifying
correspond-
ing
elements is
given in
consistency
relation

Correspond-
ing element
identifica-
tion
level

Explicit and
unique
identifica-
tion by
trace links

Implicit and
unique
identifica-
tion by key
information

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 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 must be used, e.g. based on ambiguous information or transitive resolution of indirect trace links.

In the best case, a trace link already exists between the corresponding elements. This can be due to the reason that 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 [Obj16a, p. 7.10.2.]. We call this information *implicit unique*, because elements can be unambiguously identified but rely on implicit information within the models rather than explicit traces. Note that in case that multiple corresponding elements are found by matching 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 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 element and thus no direct trace link exists, it is impossible to identify that these elements are to 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 or not.

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, which is an extended version of [Sağ20, Algorithm 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 receives the consistency relation for which corresponding elements shall be found, the condition element c_l of the condition $\mathbb{C}_{l,CR}$ that

Non-unique
identification by
heuristics

Extension of
ordinary
transformations

Algorithm
for finding
corresponding
elements

Algorithm 6. Retrieval of corresponding elements.

```
1: procedure FINDCORRESPONDING( $CR, c_l, m_2, m_{traces}$ )
2:    $tracedElements \leftarrow \{c_r \mid \langle c_l, c_r \rangle \in m_{traces}\}$ 
3:   for  $c_r \in tracedElements$  do
4:     if  $\langle c_l, c_r \rangle \in CR$  then
5:       return  $c_r$ 
6:     end if
7:   end for
8:   for  $c_r \in \mathcal{P}(m_2)$  do
9:     if  $\langle c_l, c_r \rangle \in CR$  then
10:       $m_{traces} \leftarrow m_{traces} \cup \{\langle c_l, c_r \rangle\}$ 
11:      return  $c_r$ 
12:    end if
13:   end for
14:   return  $\perp$ 
15: end procedure
```

was added to model m_1 for which corresponding elements shall be found or created, the second model m_2 in which the corresponding elements shall be searched, and the trace model $traces \subseteq \mathcal{P}(m_1) \times \mathcal{P}(m_2)$ 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 the loop in Line 3 checks whether any of the corresponding elements according to the trace model is a corresponding element in the consistency relation CR . If this is the case, a corresponding element c_r is found and the procedure returns 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 $\langle c_l, c_r \rangle$ is added to the trace model $trace$ as an appropriate trace link and the procedure returns the found element 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 the model elements by their class. Depending on the implementation of the modeling framework, different possibilities to efficiently find specific elements can be used. The implementation of 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 because a change adds 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. Thus, 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 begin partial-consistency-improving. Although we derived the insights from thorough argumentation, we also validate them in the evaluation in Chapter 9.

Application situation of algorithm

Partial consistency improvement not proven

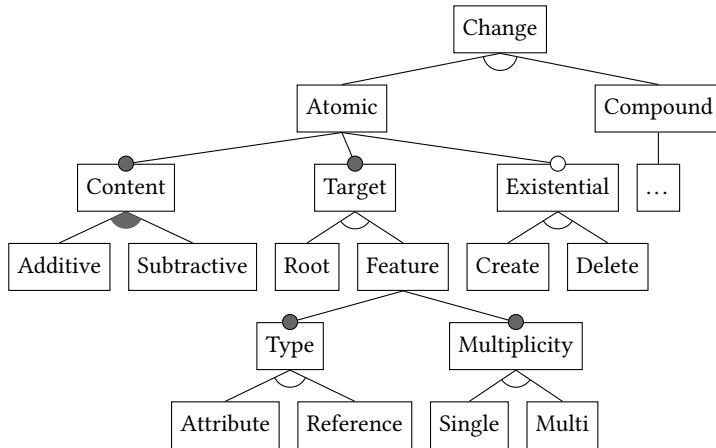
6.4.3. Model Changes To Condition Element Changes

The previous discussions were based on the distinction of different change scenarios for condition elements, as those are relevant when considering the synchronization case of bidirectional transformations. An actual 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 developers needs to decide which model changes introduce which modifications of condition elements to determine the appropriate behavior of the consistency preservation rules.

Condition element changes induced by model element changes

The possible types of model changes are induced by the used modeling formalism, as the meta-metamodel defines which changes can be performed in models. The EMOF is the most common standard describing a modeling formalism, on which Ecore, the meta-metamodel of the EMF, is based. Our

EMOF- and Ecore-based models

**Constraints:**

1. $\text{Single} \Rightarrow (\text{Additive} \wedge \text{Subtractive})$
2. $\text{Multi} \Rightarrow (\text{Additive} \oplus \text{Subtractive})$
3. $\text{Root} \Rightarrow (\text{Additive} \oplus \text{Subtractive})$
4. $\text{Existential} \Rightarrow (\text{Root} \oplus \text{Reference})$
5. $\text{Create} \Rightarrow \text{Additive}$
6. $\text{Delete} \Rightarrow \text{Subtractive}$

Figure 6.7.: Feature model for changes in Ecore-based models, adapted from [Kra17, Fig. 5.3].

modeling formalism introduced in Section 3.3 is conforming to EMOF, which is why we consider changes in EMOF- and Ecore-based models.

Kramer [Kra17] proposes feature models that express all kinds of possible changes in these types of models. The feature model for changes in EMOF-based models is given at [Kra17, Fig. 5.2] and the one for changes in Ecore-based models is given at [Kra17, Fig. 5.3]. Since EMOF and Ecore are rather similar (cf. Subsection 2.2.3), 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 at [Kra17, Fig. 5.3], we 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.

Possible
change
types

Corrections
of existing
feature
model

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 sequence and is comparable to moving an element from one reference to another, which is also modelled as a compound change.

Mandatory content: We made the *Content* feature mandatory, as due to removal of the permutation every change is either additive or subtractive.

Constraints reduction: We reduced the constraints to those that are still relevant after performing the previously discussed changes.

Error corrections: 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 types of changes in Ecore-based models can be derived 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.

Case
distinction
of model
changes

Additive root change (possibility create): Adding a root element can lead to the addition of a condition element, which consists only of this root element. It may neither induce a change or removal of a condition element.

Subtractive root change (possibility delete): Removing a root element can lead to the removal of a condition element, which involves the root element. Since it completely removes an element, it can neither lead to a change or 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, for example, by renaming an element. Finally, it can also lead to an element that is not present in a condition anymore. This does apply 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 an attribute 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 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 an attribute 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 (possibility create and/or delete): Changing 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 (possibility create): Adding a reference 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 did already exist before.

Subtractive multi-valued reference change (possibility delete): Removing a reference from a multi-valued reference 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 did already exist 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. This is due to the fact that 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.

Relevant changes restriction depending on consistency relation

Application cases for corresponding elements identification

6.5. Summary

In this chapter, we have discussed how synchronizing transformations, as required in transformation networks, can be defined using existing transformation languages. To this end, we defined the specific notion of synchronizing bidirectional transformations as an extension of bidirectional transformations defined in transformation languages. We have formally proven that these transformations always terminate consistently and that they have equal expressiveness than synchronizing transformations. Finally, we have discussed how that property can be achieved in transformation languages. We close this chapter with the following central insight:

Insight 3 (Synchronization)

Synchronizing transformations, as required in transformation networks, process models which 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 developers needs to consider only few situations explicitly to make a bidirectional transformation partial-consistency-improving. The most important is that the transformation creates elements that already exist, because another transformation 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 only fulfilling an additional property for which we provided 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 able to be properly used in a transformation network. In the following, 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 4.12, we have shortly discussed that we cannot require correctness for such a function in the sense that it always yields consistent models for any 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 will 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, i.e., 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. That algorithm is supposed to improve the ability of a transformation developer to identify

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 yielding 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 topologies of transformation networks, our approach is supposed to not restrict the topology of transformations in any way. Existing work proposes, for example, to define an execution order explicitly [Pil+08; Van+07] or to derive a topologic order [Ste20], which restricts the 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, come up with an algorithm that is able to process transformation networks of arbitrary topology, which follows a specific orchestration strategy that does not necessarily find an execution order that yields consistent models whenever it exists, but instead is defined in way that it supports the transformation developer or user in finding the reason for the inability to find such an execution 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 a concrete algorithm for the orchestration to be readily applied.

Undecidability of
orchestra-
tion
problem,
practical
algorithm

Benefits:
systematic
knowl-
edge and
concrete
algorithm

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 been published in [GKB20].

Publication
of contribu-
tions

7.1. Orchestration Goals and Problem Statement

To recapitulate, an application function $\text{APP}_{\text{ORC}_t}$ for transformation networks, as defined in Definition 4.12, 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 t of the network in an order determined by the orchestration function ORC_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 4.13 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 further discussion of the alternatives, which we provide in the following.

Application
function
results

In fact, 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 specifically 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.

Depen-
dence on
orches-
tra-
tion
func-
tion

Definition 7.1 (Orchestration)

Let \mathbb{t} be a set of transformations. We call a sequence of these transformations $[t_1, t_2, \dots] \in \mathbb{t}^{<\mathbb{N}} := \emptyset \cup \mathbb{t}^1 \cup \mathbb{t}^2 \cup \dots$ an *orchestration* of them.

For models $m \in \mathfrak{M}$ and changes $\delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}}$, we say that an orchestration $[t_1, \dots, t_n]$ is *consistent* if, and only if, the subsequent application of the transformations to m and $\delta_{\mathfrak{M}}$ is consistent, i.e.,

$$\exists \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \text{GEN}_{\mathfrak{M}, t_n} \circ \dots \circ \text{GEN}_{\mathfrak{M}, t_1}(m, \delta_{\mathfrak{M}}) = (m, \delta'_{\mathfrak{M}}) \\ \wedge \delta'_{\mathfrak{M}}(m) \text{ consistent to } \mathbb{t}$$

Length of
orchestra-
tions

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 without discussing such restriction. In the following, we show the necessity of having this unrestricted notion rather than allowing each transformation to be executed only once, as proposed by existing work such as [Ste20]. 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

Ranges for
executions
times

The possible number 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.

Spanning
trees
insufficient

The even stronger restriction to spanning trees is obviously not sufficient. Consider the following consistency relations. For reasons of simplicity, we

use model-level consistency relations according to Definition 4.1 instead of fine-grained ones:

$$CR_{12} = \{\langle m_1, m_2 \rangle, \langle m_1, m'_2 \rangle, \langle m'_1, m'_2 \rangle, \langle m'_1, m''_2 \rangle\}$$

$$CR_{13} = \{\langle m_1, m_3 \rangle, \langle m_1, m''_3 \rangle, \langle m'_1, m_3 \rangle, \langle m'_1, m'_3 \rangle\}$$

$$CR_{23} = \{\langle m_2, m_3 \rangle, \langle m'_2, m'_3 \rangle, \langle m'_2, m''_3 \rangle, \langle m''_2, m_3 \rangle\}$$

That set of relations 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 possibly spanning trees of transformations that try to restore consistency, which we denote as t_{12}, t_{13}, t_{23} for the according consistency relations. Each tree consists of two transformations:

t_{12}, t_{13} : t_{12} may change m_2 to m'_2 . t_{13} does nothing, because m'_1 and m_3 are already consistent to CR_{13} . m'_2 and m_3 are, however, not consistent to CR_{23} .

t_{12}, 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} .

t_{13}, 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 and thus 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 [GKB20], which used a different scenario context.

Necessity to
execute
each trans-
formation

Consider the example in Figure 7.1, which describes the example depicted in Figure 1.4 within the introduction more precisely. In the example, interfaces in UML and Java are related to architectural interfaces in a PCM model. PCM components are realized by equally named classes in UML and Java. Additionally, when a PCM component requires an interface, this is realized by a field with the interface type in the component realization class in UML and Java and an appropriate constructor argument. Consistency is defined

Example
scenario for
PCM, UML
and Java

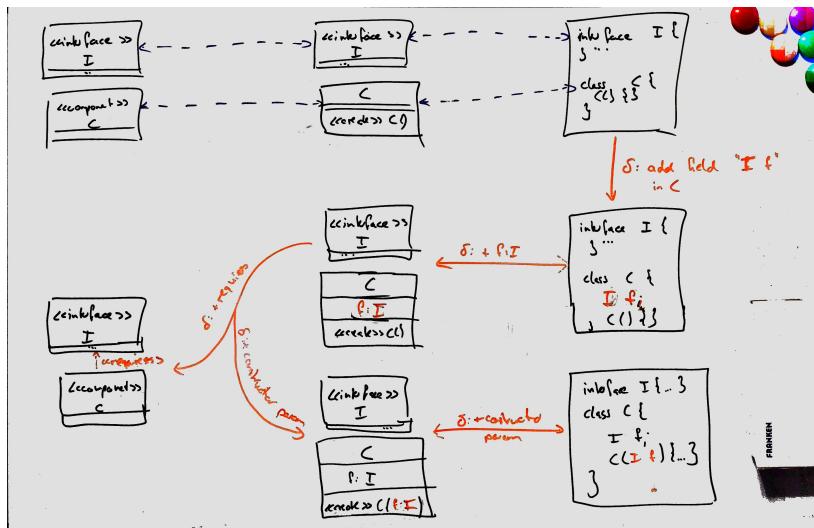


Figure 7.1.: Necessity of executing a transformation multiple times.

by transformations between PCM and UML, as well as between UML and Java.

Duplicate execution of transformation

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 UML and Java. A user then introduces a change of the Java code, in which he adds a field of the interface type to the component realization class in Java. 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 UML, as required by the consistency relations. 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.

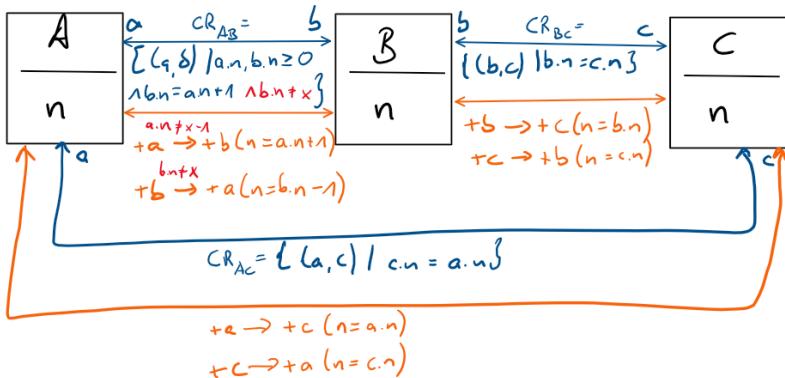


Figure 7.2.: Example for an arbitrary bound of necessary transformation execution depending on value of x .

We simplified the example to the necessary core, although in practice a further transformation between PCM and Java would be required, for example, 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 could resolve the problem, as the necessary second execution of the transformation between UML and Java is not necessary if the information is propagated from PCM to Java. This is, however, only true if exactly that order of transformations is chosen for execution and if the transformation between PCM and Java does not introduce further information in the Java model that then needs to be propagated to UML.

Cycles in transformation networks

In general, it is always possible that transformations need to react to the changes performed by others, if they are not in some way aligned with each other. This is due to the fact that a synchronizing transformation may change both models. Thus, if one transformation restores consistency between two models and another transformation reacts to that 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.

Synchronizing transformations change already processed models

We can generalize the previous example to the one depicted in 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

Example generalization

multiple times. This also applies to the case in which multiple synchronizing transformations are combined. The depicted relations and the informally defined 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 $n = x - 1$. In consequence, for an A with $n = i$, all A, B and C with $i \leq n < x$ need to 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 variable value of x . Let A, B, C be the classes depicted in Figure 7.1.

$$I_{M_1} := \mathcal{P}(I_A), I_{M_2} := \mathcal{P}(I_B), I_{M_3} := \mathcal{P}(I_C)$$

$$CR_{12} := \{\langle a, b \rangle \in I_A \times I_B \mid b.n = a.n + 1 \neq x\}, \mathbb{C}\mathbb{R}_{12} := \{CR_{12}, CR_{12}^T\}$$

$$\text{CPR}_{\mathbb{C}\mathbb{R}_{12}}^{\rightarrow}(m_1, m_2, \delta_{M_1}) = \delta_{M_2}$$

$$\text{with } \delta_{M_2}(m_2) := \{b \in I_B \mid \exists a \in \delta_{M_1}(m_1) : b.n = a.n + 1 \neq x\}$$

$$\text{CPR}_{\mathbb{C}\mathbb{R}_{12}}^{\leftarrow}(m_2, m_1, \delta_{M_2}) = \delta_{M_1}$$

$$\text{with } \delta_{M_1}(m_1) := \{a \in I_A \mid \exists b \in \delta_{M_2}(m_2) : b.n = a.n + 1 \neq x \wedge a \geq 0\}$$

$$t_{12} := \langle \mathbb{C}\mathbb{R}_{12}, \text{CPR}_{\mathbb{C}\mathbb{R}_{12}}^{\rightarrow}, \text{CPR}_{\mathbb{C}\mathbb{R}_{12}}^{\leftarrow} \rangle$$

$$CR_{13} := \{\langle a, c \rangle \in I_A \times I_C \mid c.n = a.n\}, \mathbb{C}\mathbb{R}_{13} := \{CR_{13}, CR_{13}^T\}$$

$$\text{CPR}_{\mathbb{C}\mathbb{R}_{13}}^{\rightarrow}(m_1, m_3, \delta_{M_1}) = \delta_{M_3}$$

$$\text{with } \delta_{M_3}(m_3) := \{c \in I_C \mid \exists a \in \delta_{M_1}(m_1) : c.n = a.n\}$$

$$\text{CPR}_{\mathbb{C}\mathbb{R}_{13}}^{\leftarrow}(m_3, m_1, \delta_{M_3}) = \delta_{M_1}$$

$$\text{with } \delta_{M_1}(m_1) := \{a \in I_A \mid \exists c \in \delta_{M_3}(m_3) : c.n = a.n\}$$

$$t_{13} := \langle \mathbb{C}\mathbb{R}_{13}, \text{CPR}_{\mathbb{C}\mathbb{R}_{13}}^{\rightarrow}, \text{CPR}_{\mathbb{C}\mathbb{R}_{13}}^{\leftarrow} \rangle$$

$$CR_{23} := \{\langle b, c \rangle \in I_B \times I_C \mid c.n = b.n\}, \mathbb{C}\mathbb{R}_{23} := \{CR_{23}, CR_{23}^T\}$$

$$\text{CPR}_{\mathbb{C}\mathbb{R}_{23}}^{\rightarrow}, \text{CPR}_{\mathbb{C}\mathbb{R}_{23}}^{\leftarrow} \text{ and } t_{23} \text{ accordingly}$$

$$\mathbb{C}\mathbb{R} := \mathbb{C}\mathbb{R}_{12} \cup \mathbb{C}\mathbb{R}_{13} \cup \mathbb{C}\mathbb{R}_{23}$$

$$t_{inc} := \{t_{12}, t_{13}, t_{23}\}$$

For these transformations, we can show that the transformation 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 arbitrary high number of repeated executions by proper selection of x .

Minimal
number of
executions
in example

Lemma 7.1 (Minimal Number of Transformation Executions)

Let \mathbb{t}_{inc} be the previously defined set of transformations and let $m_1 = m_2 = m_3 = \emptyset$ be empty models and $\delta_{M_1} \in \Delta_{M_1}$ a change with $\delta_{M_1}(m_1) = \{a \in I_A \mid a.n = 0\}$. Then the result of every orchestration function $ORC_{\mathbb{t}_{inc}}$ with $APP_{ORC_{\mathbb{t}_{inc}}}(\langle m_1, m_2, m_3 \rangle, \langle \delta_{M_1}, \delta_{id}, \delta_{id} \rangle)$ consistent to CR contains t_{12} at least $x - 1$ times.

Proof. $APP_{ORC_{\mathbb{t}_{inc}}}$ can only return consistent models when it applies the transformations in the order delivered by $ORC_{\mathbb{t}_{inc}}$ by definition in Definition 4.12. We thus consider every order of transformations, as delivered by any orchestration function, to show that it contains t_{12} at least $x - 1$ times to deliver consistent models.

Let $max_n(m_1, m_2, m_3) := \max\{e.n \mid e \in m_1 \cup m_2 \cup m_3\}$ 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 if the concrete models are currently not relevant.

Executing t_{13} and t_{23} an arbitrary number of times does not increase max_n :

The transformations do 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.

A single execution of t_{12} increases max_n by at most one: There is no A or B with $n > max_n$. For every A with $n < max_n$, 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 consistency relations. Thus, max_n is, at most, increased by 1.

When $\max_n(m_1, m_2, m_3) < x - 1$, **then** m_1, m_2, m_3 **are not consistent to** CR :

There is at least one element within the models with $n = \max_n$. If the element with $n = \max_n$ is an A, then there must be a B with value $n + 1$, because of CR_{12} and because $n < x - 1$. But since $n = \max_n$, 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, then 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 CR_{23} a C with the same value must exist and then the same argument as before leads to a contradiction.

In summary, we have shown that models m_1, m_2, m_3 are only consistent to CR when $\max_n(m_1, m_2, m_3) \geq x - 1$. Additionally, only 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 t_{12} in an arbitrary sequence of the transformations in \mathbb{t}_{inc} to achieve consistent models. \square

Arbitrary number of necessary executions

We have proven that arbitrary 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 if a consistent orchestration of transformations exists, the approach delivers it.

Theorem 7.2 (Orchestration with Single Execution)

For a set of transformations \mathbb{t} , there can be models \mathbf{m} and changes δ to them for which each possible orchestration function $ORC_{\mathbb{t}}$ with whom $APP_{ORC_{\mathbb{t}}}(\mathbf{m}, \delta)$ is consistent, delivers a sequence as $ORC_{\mathbb{t}}(\mathbf{m}, \delta)$ that contains at least one transformation twice.

Proof. We know from Lemma 7.1 that \mathbb{t}_{inc} requires at least 2 executions of t_{12} for the inputs defined in Lemma 7.1 when selecting $x \geq 3$. This proves the theorem by example. \square

In fact, for a concrete 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. We know, however, from Theorem 7.2 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 τ_{inc} . Thus, we can conclude that the insight is potentially relevant for realistic scenarios. We should not restrict orchestration to execute each transformation only once, as there can be realistic scenarios in which multiple executions are necessary to find consistent models. In the following, we thus, for first, 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 in [Ste20]. 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 to this end is not a proper solution as shown in the examples, which is why we do not consider a notion of authoritative models.

Single execution insufficient

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 models that are consistent. We did, however, not yet define 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 .

Returning \perp

Reasons for
not finding
consistent
orchestra-
tion

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 different reasons why the orchestration function may not deliver such a sequence:

Relations are incompatible: 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 such that the resulting models are consistent and still reflect the given user change.

No consistent orchestration exists: Even if the relations are compatible, transformations may be defined in a way that they make contradictory decisions for locally consistent solutions. Thus, for a given a change the consistency relations allow different ways to store consistency, of which the transformations always select a way that is not consistent to one of the other relations. Then no order of the transformations can restore consistency, although models exist that fulfill consistency for the given change.

No consistent orchestration found: Finally, although an order of transformations for given changes exists that delivers consistent models, the orchestration function may not deliver it.

These reasons can be considered to reside at different levels, because each of them induces the next, i.e., 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 initially given intuitive requirement that the orchestration function delivers a consistent orchestration whenever it exists would thus ensure the third level and needs to assume the first two levels to be fulfilled to avoid situations in which no consistent orchestration is found. While we can assume compatibility of the relations, as we discussed how to analyze it in Chapter 5, we cannot assume that an orchestration does always exist, as we will see in the following.

Implication
hierarchy of
reasons

Assumption
of
consistent
orchestra-
tion
existence

Compatibility not ensuring consistent orchestration

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., 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 consistency preservation rules of 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 those models are uniquely indexed by i and that we defined the following model-level consistency relations:

$$\begin{aligned} CR_{AB} &= \{\langle a_i, b_k \rangle \mid k = i\} \\ CR_{AC} &= \{\langle a_i, c_l \rangle \mid l = i \vee l = i + 1\} \\ CR_{BC} &= \{\langle b_k, c_l \rangle \mid l = k + 1 \vee l = k + 2\} \end{aligned}$$

Overlapping options in consistency relations

This induces the set of consistent 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. In the set of consistent models, however, only those models fulfilling the first of these two conditions are contained. If the consistency preservation rules do, however, always select the models that fulfill the second condition, the resulting models are locally consistent to its consistency relation, but will never become globally consistent to all three relations. More precisely, if the consistency preservation rules for CR_{AC} do always select c_i for a_i and vice versa, and if the rules for CR_{BC} do always 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.

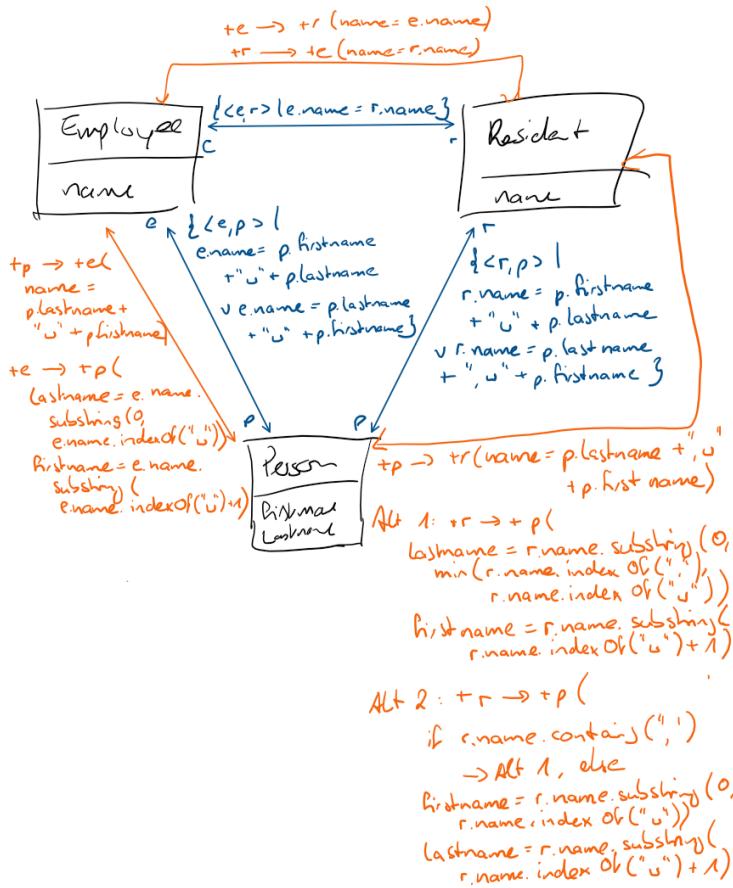


Figure 7.3.: Consistency relations with options for corresponding elements leading to consistency preservation rules for which no consistent orchestration exists.

Running example scenario

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 and 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 first and last name of a person, an employee is also allowed to have the inverse concatenation of last and first name, whereas a resident is also allowed to have this inverse concatenation, but with an additional separation of the last and first name 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 be encoded differently. This can be reasonable if the order of first and last name 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 first and last name. Nevertheless, these consistency relations are compatible, because for each possible condition element, i.e., for every possible employee, person and resident, there are consistent models that contain them.

Consistency preservation rules that preserve consistency to 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 name is the concatenation of the last and first name of a person, whereas the resident 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 first and last name 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 person and resident always introduces a comma in the resident name, which is then appended to the last name by the transformation between employee 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 first and last name that is represented as concatenation of first

Relations without
consistent
orchestra-
tion

and last name in both an employee and resident. This is a concrete example for the previously discussed abstract situation that of different options in consistency relations always the non-overlapping ones are chosen by the consistency preservation rules.

Relations with consistent orchestration

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 name. If a comma is present, the name relation containing a comma is used, and otherwise the simple concatenation of first and last name 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 first and last name of a person according to the relation that is also fulfilled between person and employee, because the name does not contain a comma. After adding a person, first executing the transformation from persons to employees and then following the process above also ensures consistency. 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.

Only specific orchestrations are consistent

Although there are consistent orchestrations of the transformations with the consistency preservation rule defined as alternative 2, 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 first. If that transformation is first executed after the addition of a person, then a comma is added, which leads to the subsequent application of the same consistency preservation rules as with alternative 1, meaning that no further orchestration yields consistent models.

Necessity to find consistent orchestrations

No matter whether exactly those 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 [Ste20], we call a given tuple of models and changes *resolvable* by a transformation network, if a consistent orchestration exists. In contrast to existing work, we do, however, not restrict ourselves to a single execution of each transformation, as we have motivated before.

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 any changes 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 do not focus on that problem, but accept that it can occur 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 $\text{GEN}_{\mathfrak{M}, t}$ is the generalization function that applies a transformation, which is only defined for two models, to a model tuple that instantiate all metamodels in \mathfrak{M} .

Resolvability

Necessity to deal with unresolvability

Optimality property of orchestration function

Optimal orchestration function

Definition 7.2 (Optimal Orchestration Function)

Let \mathbf{t} be a set of transformations for a tuple of metamodels \mathfrak{M} . We say that an orchestration function $\text{ORC}_{\mathbf{t}}$ for these transformations is *optimal* if, and only if, it returns a consistent orchestration whenever it exists, i.e.,

$$\begin{aligned} \forall \mathfrak{m} \in \{ \mathfrak{m}' \in I_{\mathfrak{M}} \mid \mathfrak{m}' \text{ consistent to } \mathbb{CR} \} : \forall \delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \\ (\exists t_1, \dots, t_i \in \mathbf{t} : \exists \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \delta'_{\mathfrak{M}}(\mathfrak{m}) \text{ consistent to } \mathbb{CR} \\ \wedge \text{GEN}_{\mathfrak{M}, t_i} \circ \dots \circ \text{GEN}_{\mathfrak{M}, t_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}}) \\ \Rightarrow \exists t'_1, \dots, t'_k \in \mathbf{t} : \exists \delta''_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \delta''_{\mathfrak{M}}(\mathfrak{m}) \text{ consistent to } \mathbb{CR} \\ \wedge \text{GEN}_{\mathfrak{M}, t'_k} \circ \dots \circ \text{GEN}_{\mathfrak{M}, t'_1}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta''_{\mathfrak{M}}) \\ \wedge \text{ORC}_{\mathbf{t}}(\mathfrak{m}, \delta_{\mathfrak{M}}) = [t'_1, \dots, t'_k]) \end{aligned}$$

Behavior
without
consistent
orchestra-
tion
existence

Note that we allow an optimal orchestration function to return a sequence when there is no consistent orchestration. This is reasonable, because an application function may be defined to return consistent models whenever there is a consistent orchestration, but to also support the process of identifying why there is none by delivering a sequence of transformations that leads to a failure, as we will discuss later.

Application
function
optimality

Finally, the result of the application function is what is relevant in the process of consistency preservation in a transformation network. 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 τ be a set of transformations for a tuple of metamodels \mathfrak{M} . We say that an application function APP_{ORC_τ} for these transformations is *optimal* if, and only if, it returns models that are consistent whenever there is a consistent orchestration of the transformations, i.e.,

$$\begin{aligned} \forall m \in \{m' \in I_{\mathfrak{M}} \mid m' \text{ consistent to } \mathbb{CR}\} : \forall \delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \\ (\exists t_1, \dots, t_m \in \tau : \exists \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \delta'_{\mathfrak{M}}(m) \text{ consistent to } \mathbb{CR} \\ \wedge GEN_{\mathfrak{M}, t_1} \circ \dots \circ GEN_{\mathfrak{M}, t_m}(m, \delta_{\mathfrak{M}}) = (m, \delta'_{\mathfrak{M}}) \\ \Rightarrow APP_{ORC_\tau}(m, \delta_{\mathfrak{M}}) \text{ consistent to } \mathbb{CR}) \end{aligned}$$

According to the defined behavior of an application function, an optimal application function requires an optimal orchestration function.

Optimality dependency

Lemma 7.3 (Application / Orchestration Function Optimality)

An application function APP_{ORC_τ} can only be optimal if ORC_τ is optimal.

Proof. Let us assume that the complete 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 the given models and changes. Then to be optimal, the application function needs to return models that are consistent. According to the definition of an application function (see Definition 4.12), the sequence of transformations delivered by ORC_τ for that input must yield the same model tuple as APP_{ORC_τ} . Thus, the orchestration function must deliver a sequence for such inputs that yields consistent models, which is equivalent to ORC_τ being optimal. \square

7.1.4. The Orchestration Problem

The problem to find a consistent orchestration whenever it 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

Orchestrati-
on problem

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 for us.

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 we know that such an orchestration exists, this test must terminate, even if it may take an impractically long time. \square

Since the orchestration problem is derived from the goal of finding an optimal application function, it is obviously equivalent to find an optimal applica-

tion function or to solve the orchestration and the orchestration existence problem.

Theorem 7.5 (Optimality / Orchestration Problem Equivalence)

An optimal application function APP_{ORC_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 APP_{ORC_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.

$$\text{EXISTSORC}(t, m, \delta_M) := \begin{cases} \text{TRUE}, & APP_{ORC_t}(m, \delta_M) \text{ consistent to } t \\ \text{FALSE}, & \text{otherwise} \end{cases}$$

EXISTSORC returns TRUE if, and only if, a consistent orchestration exists. APP_{ORC_t} does, per definition, only return consistent models when there is an orchestration that yields them. Since it is optimal, it does always return consistent models when an orchestration that yields them exists. \square

7.2. Limitations of Orchestration Decidability

We 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 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 the transformations that ensures that executing the transformations in any order

often enough, they yield consistent models if $S_i \neq \emptyset$, i.e., for any given input i there is an $n \in \mathbb{N}$ such that $\forall s \in P : |s| > n \Rightarrow s \in S_i$.

In the latter case, the orchestration function may return any order of the transformations, as long as the sequence is long enough, to be optimal. This means, performing an iterative execution of the transformations leads to a consistent result, comparable to a fixed-point iteration. Since optimality is a property of an orchestration function with respect to a set of transformations, defining a *well-behavedness* property as a restriction 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 rather than being an ordinary mathematical function. 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 yet discussed the orchestration and application functions as purely mathematical functions. In practice, however, they need to be implemented in terms of algorithms. In Algorithm 7, we propose an algorithm that realizes an application function. It also encodes the orchestration function, because in contrast to the mathematical definition, an algorithm for the orchestration function will not determine a complete sequence of transformations for given models and changes, but dynamically select the next transformation to execute. As soon as no further transformations are determined for execution by the orchestration, it returns the resulting models if they are consistent or otherwise returns \perp .

Algorithm 7. Application function implementation.

```

1: procedure APPLY( $\mathbf{t}$ ,  $\mathbf{m}$ ,  $\delta_{\mathfrak{M}}$ )
2:    $isConsistent \leftarrow \text{CHECKCONSISTENCY}(\mathbf{t}, \mathbf{m})$ 
3:   if  $\neg isConsistent$  then
4:     return  $\perp$ 
5:   end if
6:    $\mathbf{t}_{executed}[] \leftarrow []$ 
7:    $\delta_{\mathfrak{M}, generated}[] \leftarrow []$ 
8:    $\mathbf{t}_{next} \leftarrow \text{ORCHESTRATE}_{\mathbf{t}}(\mathbf{m}, \delta_{\mathfrak{M}}, \mathbf{t}_{executed}[], \delta_{\mathfrak{M}, generated}[])$ 
9:   while  $\mathbf{t}_{next} \neq \perp$  do
10:     $(\mathbf{m}, \delta_{\mathfrak{M}}) \leftarrow \text{GEN}_{\mathfrak{M}, \mathbf{t}_{next}}(\mathbf{m}, \delta_{\mathfrak{M}})$ 
11:     $\mathbf{t}_{executed}[] \leftarrow \mathbf{t}_{executed}[] + \mathbf{t}_{next}$ 
12:     $\delta_{\mathfrak{M}, generated}[] \leftarrow \delta_{\mathfrak{M}, generated}[] + \delta_{\mathfrak{M}}$ 
13:     $\mathbf{t}_{next} \leftarrow \text{ORCHESTRATE}_{\mathbf{t}}(\mathbf{m}, \delta_{\mathfrak{M}}, \mathbf{t}_{executed}[], \delta_{\mathfrak{M}, generated}[])$ 
14:   end while
15:    $\mathbf{m}_{res} \leftarrow \delta_{\mathfrak{M}}(\mathfrak{M})$ 
16:    $isConsistent \leftarrow \text{CHECKCONSISTENCY}(\mathbf{t}, \mathbf{m}_{res})$ 
17:   if  $\neg isConsistent$  then
18:     return  $\perp$ 
19:   end if
20:   return  $\mathbf{m}_{res}$ 
21: end procedure

```

An application function according to Definition 4.12 is parametrized by an orchestration function, which, in turn, is parametrized by the set of transformations \mathbf{t} that it is supposed to be executed on. A transformation network according to Definition 4.14 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 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 concrete transformations and, instead, accepts them as a parameter. We do, however, focus on a general algorithm and ORCHESTRATE function that can be applied to any set of transformations. In that case, the algorithm does not realize a single

Independent from
concrete
transformations

application function but actually a family of application functions for all possible transformation sets τ .

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 does only return a transformation that can be applied to the models and current changes, because otherwise its application by the GEN function in Line 10 would fail. The complete logic of the orchestration function is combined with the application of the delivered sequence in Lines 8–14. 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 8–14 leads to a redundancy by applying the transformations both in the separate orchestration algorithm as well as in the given 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 for the ORCHESTRATE function after its first execution anymore. In a practical implementation of that function, the history may, however, not be considered or truncated, depending on the information necessary for the concrete implemented orchestration strategy.

The ORCHESTRATE function may implement different strategies for selecting the next transformation, which we will later discuss in more detail. One of the most simple strategies 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 transformation and after executing one transformation to enqueue all transformations that are adjacent to the metamodels of the two models that were modified by the transformation if they are not yet enqueued. This ensures that those transformation are executed next which can process changes that have just been produced by another transformation. Both these

Dynamic selection of next transformation

History of changes and transformations

Strategies for selecting next transformation

strategies are independent from the concrete transformations and could thus be implemented in a function that can be used for any set of transformations t . In Section 7.4, we will discuss a specific orchestration strategy. Until then, the concrete 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 4.11. The `CHECKCONSISTENCY` function checks whether the given models are consistent to the set of transformations, according to Definition 4.9. 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* [Obj16a, 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 4.8, 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 implicitly defined as the image 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.

Assumed additional functions

7.2.2. Correctness and Termination of the Algorithm

Algorithm 7 is constructed to implement an application function according to Definition 4.12. It is designed to be correct, i.e., it returns models only when they are consistent. We show that the algorithm fulfills these properties in the following theorem.

Algorithm as correct application function

Theorem 7.6 (Apply Algorithm Correctness)

The APPLY function in Algorithm 7 fulfills the functional behavior of an application function as defined in Definition 4.12 and is correct according to Definition 4.13.

Proof. The APPLY function fulfills the input and output requirements of an application function according to Definition 4.12. It returns a model tuple only in Line 20, which is achieved by applying the changes delivered by the sequence of transformations delivered by the orchestration function realized as a repeated call of the ORCHESTRATE function in Lines 8–14. Thus, APPLY fulfills the definition of an application function.

Correctness of an application function according to Definition 4.13 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 20. These models are always consistent to the consistency relations of all transformations, because Lines 16–19 ensure this and otherwise return \perp before. \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 sequence of transformations is executed and leads to the same changes, which means that the algorithm only performs an infinite alternation. We do, however, need to ensure that the ORCHESTRATE function returns \perp after a finite number of calls.

If we assume that we can achieve optimality for the orchestration function, we would have the guarantee that if a consistent orchestration exists, the

No
termination
guarantee

Termination
guarantee
options

function will find it. There is, however, no restriction to what the orchestration function may return when there is no consistent orchestration at all. Thus, we have 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, such that if more transformations were 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 will, however, prove in the following that there is no such upper bound. Afterwards, we will show that identifying whether a consistent orchestration exists is not possible either. This will lead to the insight that we cannot guarantee termination of the algorithm with an optimal orchestration function.

Termination by upper bound

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 in Lemma 7.1 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, not sufficient, we can also use it to show the more general statement that we cannot find a maximal length for the orchestration of transformation networks of specific size.

No general upper bound

Theorem 7.7 (Upper Bound for Shortest Consistent Orchestration)

For every $t_{size} \geq 3$ and every $n \geq 0$, there is a set of transformations t with $|t| > t_{size}$ such that there are m and changes δ to them for which each possible orchestration function ORC_t , with whom $APP_{ORC_t}(m, \delta)$ is consistent, delivers a transformation sequence with $|ORC_t(m, \delta)| > n$.

Proof. We know from Lemma 7.1 that t_{inc} requires at least $x - 1$ executions of 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 $x - 1 = n + 1$ executions of t_{12} are necessary. Let m and δ be the inputs defined in Lemma 7.1. Then for any orchestration function $ORC_{t_{inc}}$ that delivers a consistent orchestration for these inputs, we know that $|ORC_{t_{inc}}(m, \delta)| \geq x - 1 = n + 1 > n$. Since $|t_{inc}| = 3$ and adding arbitrary transformations whose consistency preservation rules implement the identity function leads to transformation sets of arbitrary size ≥ 3 with the same behavior, this proves the theorem by example. \square

In consequence, it is not possible to find a fixed value or a value only 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 no upper bound for the shortest consistent orchestration. Thus, even if we are able to ensure optimality of the orchestration realized by the **APPLY** and **ORCHESTRATE** functions, there is no upper bound for the number of executed transformations in a consistent orchestration. We cannot abort the execution 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 will show that this orchestration existence problem is undecidable. To do so, 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 modelling formalism, in [GKB20].

Given a Turing machine TM over some alphabet Σ , we construct metamodels \mathfrak{M}_{TM} and a transformation network with a set of transformations t_{TM} , as well as initial models $m_{TM,x} \in I_{\mathfrak{M}_{TM}}$ and changes $\delta_{\mathfrak{M}_{TM},x}$ 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

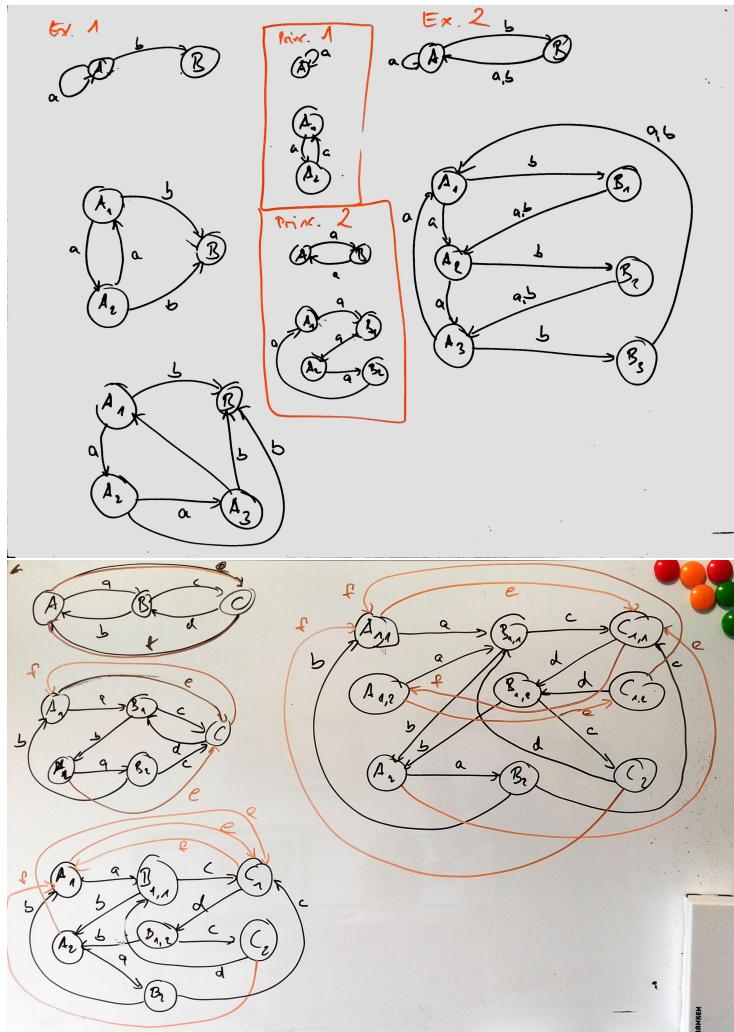


Figure 7.4.: Principles to eliminate cycles of length ≤ 2 in transition function of a Turing machine and two application examples.

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 duplicating 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 let all ingoing 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 let all ingoing 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 and the application to a scenario with self-loops as well as three states with pairwise cycles of length 2 are depicted in Figure 7.4.

Models representing Turing machine states

We construct models that consist of a timestamp, the tape content and the tape position. With our formalism, we can, for example, encode this into a metamodel M_{TM} having one class with exactly these contents and models that do only contain one instance of that class. For reasons of simplicity, we do not explicitly denote the according metamodel and denote a model m as $m := \langle \text{time}, \text{cont}, \text{pos} \rangle \in \mathbb{N}_0 \times \Sigma^* \times \mathbb{N}_0$, which simply represents a tuple of timestamp, tape content and tape position. We use one model for each state of the Turing machine and one transformation between each pair of models whose represented states have a transition between them. To be able to identify the state of the Turing machine that a model represents by its metamodel, we assume one metamodel for each of the states, although they all look equal. Thus, we have $\mathfrak{M}_{\text{TM}} = \langle M_{1,\text{TM}}, \dots, M_{n,\text{TM}} \rangle$ with $n = |Q_{\text{TM}}|$ if we assume $Q_{\text{TM}} = \{q_1, \dots, q_n\}$ to be the set of states of TM . The instances of each of the metamodels $M_{i,\text{TM}}$ reflect the possible models combining timestamp, tape content and tape position, i.e., $I_{M_{i,\text{TM}}} = \mathbb{N}_0 \times \Sigma^* \times \mathbb{N}_0$. We define the following function that returns the state of the Turing machine represented by a metamodel:

$$Q : M_{i,\text{TM}} \mapsto q_i$$

Transformations representing Turing machine transitions

The transformations increment the timestamp, change the tape content and update the tape position according to the transition of TM if, and only if, the timestamp of one model is higher than the one of the other. More formally, let $\text{Tr}(q_1, q_2) \subseteq \Sigma \times \{-1, 0, 1\} \times \Sigma$ 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”). We define a consistency preservation rule for the transformation between metamodels $M_{i,\text{TM}}$ and $M_{k,\text{TM}}$ realizing the transition between the represented states of TM as follows:

$$\text{CPR}_{i,k}(m_i, m_k, \delta_{M_{i,\text{TM}}}, \delta_{M_{k,\text{TM}}}) = (\delta'_{M_{i,\text{TM}}}, \delta'_{M_{k,\text{TM}}})$$

with

$$m'_i := \langle \text{time}_{m'_i}, \text{cont}_{m'_i}[], \text{pos}_{m'_i} \rangle := \delta_{M_{i,\text{TM}}}(m_i)$$

$$m'_k := \langle \text{time}_{m'_k}, \text{cont}_{m'_k}[], \text{pos}_{m'_k} \rangle := \delta_{M_{k,\text{TM}}}(m_k)$$

$$\delta'_{M_{i,\text{TM}}}(m_i) := \begin{cases} \langle \text{time}_{m'_k} + 1, \text{cont}_{m'_k}|_{\text{pos}_{m'_k} \leftarrow \text{repl}}, \text{pos}_{m'_k} + \text{dir} \rangle, \\ \text{if } \text{time}_{m'_k} > \text{time}_{m'_i} \wedge \\ \exists \langle \text{cont}_{m'_k}[\text{pos}_{m'_k}], \text{dir}, \text{repl} \rangle \in \text{Tr}(Q(M_{i,\text{TM}}), Q(M_{k,\text{TM}})) \\ \delta_{M_{i,\text{TM}}}(m'_i), \quad \text{else} \end{cases}$$

$$\delta'_{M_{k,\text{TM}}}(m_k) := \begin{cases} \langle \text{time}_{m'_i} + 1, \text{cont}_{m'_i}|_{\text{pos}_{m'_i} \leftarrow \text{repl}}, \text{pos}_{m'_i} + \text{dir} \rangle, \\ \text{if } \text{time}_{m'_i} > \text{time}_{m'_k} \wedge \\ \exists \langle \text{cont}_{m'_i}[\text{pos}_{m'_i}], \text{dir}, \text{repl} \rangle \in \text{Tr}(Q(M_{i,\text{TM}}), Q(M_{k,\text{TM}})) \\ \delta_{M_{k,\text{TM}}}(m'_k), \quad \text{else} \end{cases}$$

where $\text{cont}|_{\text{pos} \leftarrow \text{repl}} := \text{cont}[0 .. \text{pos} - 1] \cdot \text{repl} \cdot \text{cont}[\text{pos} + 1 .. |\text{cont}| - 1]$.

The model-level consistency relations are implicitly given by the fixed points of the consistency preservation rules. For a consistency preservation rule $\text{CPR}_{i,k}$, we define:

$$\text{CR}_{i,k} = \{ \langle m_i, m_k \rangle \in I_{M_{i,\text{TM}}} \times I_{M_{k,\text{TM}}} \mid \exists m'_i, m'_k, \delta_{M_{i,\text{TM}}}, \delta_{M_{k,\text{TM}}} : \\ \text{CPR}_{i,k}(m'_i, m'_k, \delta_{M_{i,\text{TM}}}, \delta_{M_{k,\text{TM}}})(m'_i, m'_k) = \langle m_i, m_k \rangle \}$$

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, 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.

Implicit
consistency
relations

Transformations only
for transitions

Representation of
Turing
machine
start state

Reduction
of halting
problem to
orchestra-
tion
problem

We denote the set of all transformations realizing the transitions of TM as \mathbb{t}_{TM} , containing transformations $t_{i,k} = \langle CR_{i,k}, \text{CPR}_{i,k} \rangle$ for all metamodel pairs $\langle M_{i,\text{TM}}, M_{k,\text{TM}} \rangle$, for which a transition between the represented states in Q_{TM} exists, i.e., $\text{Tr}(Q(M_{i,\text{TM}}), Q(M_{k,\text{TM}})) \neq \emptyset$.

Let $s \in Q_{\text{TM}}$ be the initial state of TM. We set

$$\begin{aligned}\mathfrak{m}_{\text{TM},x} &:= \langle m_{1,\text{TM},x}, \dots, m_{n,\text{TM},x} \rangle \\ \text{with } m_{i,\text{TM},x} &:= \langle 0, \varepsilon, 0 \rangle \\ \delta_{\mathfrak{M},\text{TM},x} &:= \langle \delta_{M_1}, \dots, \delta_{M_n} \rangle \\ \text{with } \delta_{M_i,\text{TM},x}(m_i) &:= \begin{cases} \langle 1, x, 0 \rangle, & \text{if } Q(M_i) = s \\ m_i, & \text{else} \end{cases}\end{aligned}$$

We can show that for every Turing machine, this construction of a transformation network out of it solves the halting problem by construction 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 $\mathfrak{m}_{\text{TM},x}$ and changes $\delta_{\mathfrak{M},\text{TM},x}$ 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 $\mathfrak{m}_f = \delta_{\mathfrak{M},f}(\mathfrak{m}_{\text{TM},x})$ with the highest timestamp contains $\text{tm}(x)$ as tape content.

Proof. Let $\delta_s, s \in \mathbb{N}_0$ be the sequence 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 $t_{i,k} \in \mathbb{t}_{\text{TM}}$, such that $\langle m_{i,s}, m_{k,s} \rangle$ is not consistent to $t_{i,k}$, 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 $\mathbf{m}_{\text{TM},x}$. 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 of 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 $(time_{h,s}, cont_{h,s}, pos_{h,s}) := m_{h,s} \in \mathbf{m}_s$ that has the highest timestamp $time_{i,s}$ of all models in \mathbf{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 thus can 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 either be $m_{i,s}$ or $m_{k,s}$. We assume without loss of generality $m_{h,s} = m_{i,s}$.
3. If a $t_{i,k}$ exists to which $\langle m_{i,s}, m_{k,s} \rangle$ is not consistent, then $m_{k,s+1}$ will contain 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 $cont_s$ and tape position pos_s . Additionally, $m_{k,s+1}$ will be the model with the highest timestamp of all models in \mathbf{m}_{s+1} .
4. \mathbf{m}_s is consistent to \mathbb{T}_{TM} and thus no further transformation can produce changes if, and only if, TM would halt in state $m_{i,s}$ with tape content $cont_{i,s}$ and tape position $pos_{i,s}$. 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. \square

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 construction 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. \square

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 APP_{ORC_t} be an application function. Then APP_{ORC_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. \square

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.

Algorithm
cannot
terminate
and be
optimal

Corollary 7.11 (Apply Algorithm Non-Optimality)

APPLY according to Algorithm 7 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. \square

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 does never return consistent models although there is no orchestration that yields them, but may not return consistent models in some cases in which actually an orchestration that yields them existed.

Transforma-
tion
restriction
vs. conser-
vativeness

Finally, just because the orchestration problem is undecidable does not mean that this must be an essential problem for executing practical transformation networks. Most programming languages are Turing-complete and are still 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 must not mean that we cannot practically apply those networks. We will thus especially focus on how to deal with undecidability and conservatively approximate the problem.

Practical
relevance o
f both
options

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 above discussed problem. Afterwards, we discuss how we can

Discussion
of both
options

realize APPLY in a way that it always terminates and produces reasonable outputs.

7.2.4. Restriction of Transformation Networks

We have discussed that it is necessary to restrict the transformations as an input of the application function to avoid that it is undecidable whether a consistent orchestration of them exists for given models and changes. Such restrictions can be defined at two levels:

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, i.e., the combination of transformations. Only a set of transformations can have 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 will see 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

Transformation-
local vs.
network
restrictions

Impractical-
ity of
restrictions

Selection of
contradic-
tory
options

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 does always exist would be to restrict consistency relations, such 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 relations are compatible, all consistency relations relate elements in an equal way, thus the transformations must find exactly those elements.

Although that approach will at least reduce the number of cases in which no consistent orchestration is found in our algorithm, there are still inputs for which no consistent orchestration exists. Since we do not restrict the transformations in what they 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, which are consistent to that transformation but not to any models that can be delivered by the other transformations.

Let A, B, C be three classes, each with one attribute n storing a number. We define the following metamodels, each only consisting of one of those classes, and consistency relations between them that define that for each element in one model a corresponding one in the others with the same value of n has to exist. These consistency relations are obviously compatible. This is a further simplification of our running example that requires persons, residents and employees with the same names. Additionally, we define consistency preservation rules, each of them delivering changes that always, i.e., for every input, yield the same models that only consist of one element with a specific value. The resulting models are chosen in a way such that they

Restrict
options in
consistency
relations

Restriction
not solving
the
problem

Example
with unsat-
isfactory
restriction

are consistent to the according consistency relation, but not to any of the others.

$$I_{M_1} := \mathcal{P}(I_A), I_{M_2} := \mathcal{P}(I_B), I_{M_3} := \mathcal{P}(I_C)$$

$$CR_{12} := \{\langle a, b \rangle \in I_A \times I_B \mid a.n = b.n\}, \mathbb{C}\mathbb{R}_{12} := \{CR_{12}, CR_{12}^T\}$$

$$\text{CPR}_{\mathbb{C}\mathbb{R}_{12}}(m_1, m_2, \delta_{M_1}, \delta_{M_2}) := (\delta'_{M_1}, \delta'_{M_2})$$

$$\text{with } \delta'_{M_1}(m_1) := \{a \in I_A \mid a.n = 1\} \wedge \delta'_{M_2}(m_2) := \{b \in I_B \mid b.n = 1\}$$

$$CR_{13} := \{\langle a, c \rangle \in I_A \times I_C \mid a.n = c.n\}, \mathbb{C}\mathbb{R}_{13} := \{CR_{13}, CR_{13}^T\}$$

$$\text{CPR}_{\mathbb{C}\mathbb{R}_{13}}(m_1, m_3, \delta_{M_1}, \delta_{M_3}) := (\delta'_{M_1}, \delta'_{M_3})$$

$$\text{with } \delta'_{M_1}(m_1) := \{a \in I_A \mid a.n = 2\} \wedge \delta'_{M_3}(m_3) := \{c \in I_C \mid c.n = 2\}$$

$$CR_{23} := \{\langle b, c \rangle \in I_B \times I_C \mid b.n = c.n\}, \mathbb{C}\mathbb{R}_{23} := \{CR_{23}, CR_{23}^T\}$$

$$\text{CPR}_{\mathbb{C}\mathbb{R}_{23}}(m_2, m_3, \delta_{M_2}, \delta_{M_3}) := (\delta'_{M_2}, \delta'_{M_3})$$

$$\text{with } \delta'_{M_2}(m_2) := \{b \in I_B \mid b.n = 3\} \wedge \delta'_{M_3}(m_3) := \{c \in I_C \mid c.n = 3\}$$

The given consistency relations are compatible, they contain each condition element only in one consistency relation pair, and the consistency preservation rules are correct, as their result is consistent to the consistency relation. Still, there is no consistent orchestration of the transformation for any input that is not yet consistent. This is because the consistency preservation rules always produce models that are not consistent to the consistency relations between the other models.

One might argue that the defined consistency preservation rules are highly unreasonable and will never occur in that way in practice. We would probably assume the 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 do, however, not yet have an appropriate notion for that. Some work on transformations, such as [Che+17; MC16], proposes a notion of *least change* to ensure that transformations do not perform arbitrary unrelated changes, which could exclude that situations.

Although the given example is rather artificial and although there might be the additional property of least change, which 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 that there is exactly one Java class that is considered consistent to a UML class is obviously not applicable in practice, thus this 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.

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 possible every input, even if the examples that we have used to show that on are rather artificial. Although this does not serve as a proof for 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, if even impractical restrictions do not solve it.

Least
change
property
not solving
problem

Unlikely to
find
practical
restrictions

7.2.5. Confluence in Transformation Networks

Confluence is an even stronger requirement than the existence of an optimal orchestration. In literature [Ste20], 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.

Confluence
definition

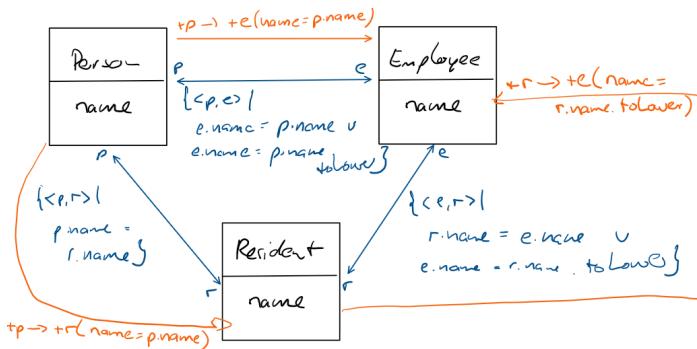


Figure 7.5.: Counterexample for the practicality of confluence.

Counterexample for confluence

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 lower case. 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 lower case 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. Second, the transformation between persons and residents can be executed, followed by the one between residents and employees. Then the employee has the name in lower case, but still this is consistent to the relation between persons and employees.

Impracticality of confluence

Apart from that artificial example, such a situation can always occur if transformations have different options for elements to be consistent. If there is not a single element that is in the overlap of consistent elements between all transformations, 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 prevent it by requiring confluence is not practical. Finally, Stevens [Ste20, p. 14] also

states explicitly that a network will only be confluent under very specific circumstances.

7.3. Conservative Approximation of the Orchestration Problem

In the preceding section, we have proven that the orchestration problem is undecidable and we have discussed that it is unlikely to restrict the problem in a way 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, does 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 even further from finding consistent orchestrations.

Based on those previous 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 found 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.

Optimal orchestration unachievable

Gradual optimality notion

7.3.1. Systematic Improvement of Optimality

Since no optimal application function can be achieved, we can at least define a gradual notion of optimality. It indicates for how many inputs of models and changes the application function is able to return consistent models in comparison to the number of cases in which a consistent orchestration exists at all. This can be seen as a fitness function for the optimality OPT of an application function:

$$\begin{aligned} \text{OPT}(\text{APP}_{\text{ORC}_t}) \\ := \frac{|\{\langle m, \delta_M \rangle \mid \text{APP}_{\text{ORC}_t}(m, \delta_M) \text{ is consistent to } t\}|}{|\{\langle m, \delta_M \rangle \mid \text{consistent orchestration of } t \text{ exists for } \langle m, \delta_M \rangle\}|} \end{aligned}$$

In fact, both the numerator as well as the denominator will usually 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 be able 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 $\text{APP}_{\text{ORC}_t}$ 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_{m, δ_M} :

$$E_{m, \delta_M} : \text{APP}_{\text{ORC}_t}(m, \delta_M) \text{ is consistent to } t$$

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:

$$P(E_{m, \delta_M}) = \frac{|\{t[] \in t^{<\mathbb{N}} \mid t[] \text{ is consistent orchestration for } \langle m, \delta_M \rangle\}|}{|t^{<\mathbb{N}}|}$$

Here, the denominator is the size of what we previously introduced as the problem space $P = |\mathbf{t}^{<\mathbb{N}}|$ containing all possible orchestrations, and the numerator is the size of what we previously introduced as the solution space $S_i = |\{\mathbf{t}[] \in \mathbf{t}^{<\mathbb{N}} \mid \mathbf{t}[] \text{ is consistent orchestration for } \langle \mathbf{m}, \delta_{\mathbf{M}} \rangle\}|$, which contains all consistent orchestrations for an input of models and changes.

We can introduce a stochastic variable $AppCons_{\mathbf{m}, \delta_{\mathbf{M}}}$, which assigns the values 0 and 1 to the events $E_{\mathbf{m}, \delta_{\mathbf{M}}}$ and its complementary:

$$AppCons_{\mathbf{m}, \delta_{\mathbf{M}}}(\omega) := \begin{cases} 0, & \omega = APP_{ORC_t}(\mathbf{m}, \delta_{\mathbf{M}}) \text{ is not consistent to } \mathbf{t} \\ 1, & \omega = APP_{ORC_t}(\mathbf{m}, \delta_{\mathbf{M}}) \text{ is consistent to } \mathbf{t} \end{cases}$$

The expected value of this variable is then equal to the probability of the event $E_{\mathbf{m}, \delta_{\mathbf{M}}}$ to occur:

$$\mu(AppCons_{\mathbf{m}, \delta_{\mathbf{M}}}) = P(AppCons_{\mathbf{m}, \delta_{\mathbf{M}}} = 1) = P(E_{\mathbf{m}, \delta_{\mathbf{M}}})$$

For an application function that chooses a random orchestration, we can thus express the numerator of $\text{OPT}(APP_{ORC_t})$ as the sum of expected values of the stochastic variables for all possible inputs.

$$\begin{aligned} |\{\langle \mathbf{m}, \delta_{\mathbf{M}} \rangle \mid APP_{ORC_t}(\mathbf{m}, \delta_{\mathbf{M}}) \text{ is consistent to } \mathbf{t}\}| &= \sum_{\mathbf{m}, \delta_{\mathbf{M}}} \mu(AppCons_{\mathbf{m}, \delta_{\mathbf{M}}}) \\ &= \frac{\sum_{\mathbf{m}, \delta_{\mathbf{M}}} |\{\mathbf{t}[] \in \mathbf{t}^{<\mathbb{N}} \mid \mathbf{t}[] \text{ is consistent orchestration for } \langle \mathbf{m}, \delta_{\mathbf{M}} \rangle\}|}{|\mathbf{t}^{<\mathbb{N}}|} \end{aligned}$$

Thus, if we can improve $P(E_{\mathbf{m}, \delta_{\mathbf{M}}})$, we also improve optimality, even if orchestrations are chosen randomly. We can improve that 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 this 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.

Expected value for finding consistent orchestration

Number of found consistent orchestrations

Improvement by reducing considered orchestrations

7. Orchestrating Transformation Networks

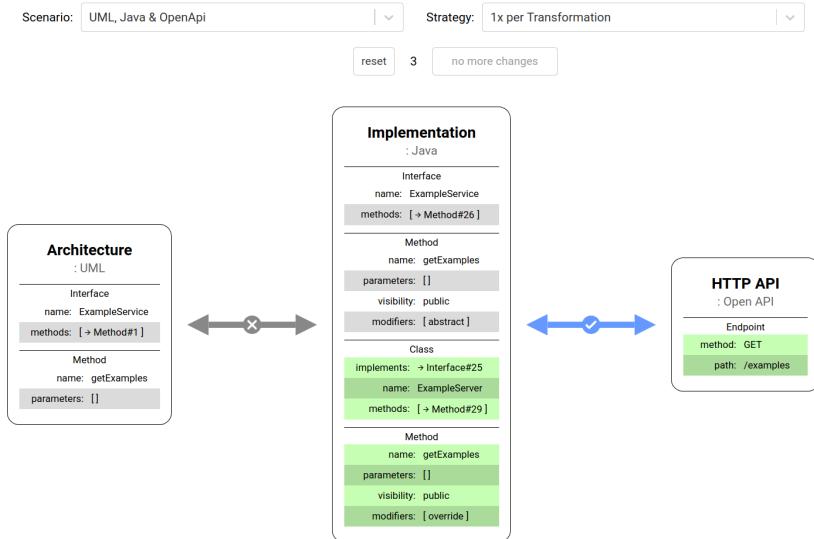


Figure 7.6.: Screenshot of a network of for an architecture specification, its implementation and an API specification in OpenAPI, which requires multiple execution of the same transformation, developed in the transformation network simulator [Gle20].

The application function can, of course, contain more intelligent logic for determining an orchestration beyond the random selection of transformations to improve the number of cases in which it finds a consistent orchestration. By implementing further mechanisms to make a reasonable selection, the possibility to find a consistent orchestration may be further improved. 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 developed specifically for that purpose, which is available at GitHub [Gle20]. An example of a scenario showing the necessity to execute transformation more than once that we implemented in the simulator is depicted in Figure 7.6 For each strategy, however, we found categories of transformation networks for which it performed worse than another 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

No systematic improvement

Backtracking to explore problem space

example, be achieved by performing backtracking. Algorithm 7 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 transformation are tried out, the algorithm recursively steps back the iterations of the loop. While this approach, in theory, allows 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, for example, those with a limited length.

Since it was impossible to find a strategy that is, in general, superior to other investigated strategies, we gave up that direction and 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 improving optimality, by both its dynamic detection as well as its avoidance. Additionally, we focused on finding a strategy that improves the ability to deal with the fact that an optimality of 1 can never be achieved by orchestrating the transformations such that the process of finding the reason for not finding a consistent orchestration is supported.

Avoiding
alternation

7.3.2. Dynamic Detection of Alternation

The proposed algorithm, 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

Orchestra-
tion
requires
abortion
criterion

could yield consistent models, 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 10 of Algorithm 7 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 Algorithmus)

During an execution of Algorithm 7, let there be a number n of executions of the transformation execution loop in Lines 8–14 of Algorithm 7, such that for all numbers of loop executions $> n$ there is a sequence of executed transformations and generated changes that occur at least two times subsequently at the end of the current states of $t_{executed}[]$ and $\delta_{M,generated}[]$. 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 yet generated changes and is thus able to identify the situation that the same sequence of transformations was already executed and produced equal changes in each application. This enables the implementation of 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.

Alternation produces orchestrations that can never yield consistent models, thus they are part of the problem space $P = t^{<\mathbb{N}}$ of finding an orchestration for a given input i of models and changes but can never be part of the

Definition
of
alternation

Orchestrate
function
detecting
alternation

Avoiding
alternation
improves
orchestra-
tion

solution space $S_i = |\{t[] \in t^{<\mathbb{N}} \mid t[] \text{ is consistent orchestration for } \langle m, \delta_M \rangle\}|$ containing the consistent orchestrations. Avoiding such alternations thus reduces the problem space without affecting the solution space 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, as a specific kind of non-termination scenario, can be avoided by construction of the orchestration function or at least can be detected by the `APPLY` algorithm. Instead of detecting alternation during orchestration, 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 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 because of two reasons. First, the notion of partial consistency is not applicable for transformation networks, because each transformation needs to restore consistency between two models completely. 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 a different notion of monotony for transformations as follows.

Non-alternation by construction

Monotony notion from synchronizing transformations

Definition 7.7 (Monotone Synchronizing Transformation)

Let $\mathfrak{M} = \langle M_1, \dots, M_n \rangle$ be metamodels and let t be a synchronizing transformation. We call t monotone if, and only if, it does not change elements that were already changed, i.e.,

$$\begin{aligned} \forall \mathfrak{m} = \langle m_1, \dots, m_n \rangle \in \mathfrak{M}, \delta_{\mathfrak{M}} = \langle \delta_{M_1}, \dots, \delta_{M_n} \rangle \in \Delta_{\mathfrak{M}} : \\ (\exists \delta'_{\mathfrak{M}} = \langle \delta'_{M_1}, \dots, \delta'_{M_n} \rangle \in \Delta_{\mathfrak{M}} : \text{GEN}_{\mathfrak{M}, t}(\mathfrak{m}, \delta_{\mathfrak{M}}) = (\mathfrak{m}, \delta'_{\mathfrak{M}}) \\ \Rightarrow \forall i \in \{1, \dots, n\} : (\delta_{M_i}(m_i) \setminus m_i) \subseteq \delta'_{M_i}(m_i) \\ \wedge (m_i \setminus \delta_{M_i}(m_i)) \cap \delta'_{M_i}(m_i) = \emptyset) \end{aligned}$$

No
repeated
change of
same
elements

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, which leads to the model relations in the definition.

Property of
monotone
transforma-
tion
orchestra-
tions

Having only monotone transformations ensures that each orchestration, which does not apply a transformation to already consistent models, yields a sequence of pairwise different model states, if the transformations are sequentially applied.

Lemma 7.12 (Monotone Transformation Orchestration Prefixes)

Let \mathbf{t} be a set of correct monotone synchronizing transformations for a tuple of metamodels \mathfrak{M} . Then for all models and changes, as well as any orchestration $[t_1, \dots, t_m] \in \mathbf{t}^{<\mathbb{N}}$ that does not contain a transformation to be executed when its models are already consistent, the prefixes of that orchestration only yield the same models if those prefixes are consistent orchestrations, i.e.,

$$\begin{aligned} & \forall m \in I_{\mathfrak{M}}, \delta_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : \forall i, k \in \{1, \dots, m\} : \\ & GEN_{\mathfrak{M}, t_i} \circ \dots \circ GEN_{\mathfrak{M}, t_1}(m, \delta_{\mathfrak{M}}) = GEN_{\mathfrak{M}, t_k} \circ \dots \circ GEN_{\mathfrak{M}, t_1}(m, \delta_{\mathfrak{M}}) \\ & \Rightarrow \exists \delta'_{\mathfrak{M}} \in \Delta_{\mathfrak{M}} : GEN_{\mathfrak{M}, t_k} \circ \dots \circ GEN_{\mathfrak{M}, t_1}(m, \delta_{\mathfrak{M}}) = (\mathbf{m}, \delta'_{\mathfrak{M}}) \\ & \quad \wedge \delta'_{\mathfrak{M}}(m) \text{ consistent to } \mathbf{t} \end{aligned}$$

Proof. Assume that there are two prefixes $[t_1, \dots, t_i]$ and $[t_1, \dots, t_k]$ of an orchestration, $i < k$ without loss of generality, such that they yield the same inconsistent models, i.e., $GEN_{\mathfrak{M}, t_i} \circ \dots \circ GEN_{\mathfrak{M}, t_1}(m, \delta_{\mathfrak{M}}) = GEN_{\mathfrak{M}, t_k} \circ \dots \circ GEN_{\mathfrak{M}, t_1}(m, \delta_{\mathfrak{M}})$ although $GEN_{\mathfrak{M}, t_k} \circ \dots \circ GEN_{\mathfrak{M}, t_1}(m, \delta_{\mathfrak{M}})$ is not consistent to \mathbf{t} . We denote the change tuple delivered by any prefixes of length l as $\delta_{\mathfrak{M}, l} = \langle \delta_{M_1, l}, \dots, \delta_{M_n, l} \rangle$ with $\langle m, \delta_{\mathfrak{M}, l} \rangle = GEN_{\mathfrak{M}, t_l} \circ \dots \circ GEN_{\mathfrak{M}, t_1}(m, \delta_{\mathfrak{M}})$. We know that the sequence of changes between the two prefixes does not perform any changes and thus acts like the identity function, i.e., $GEN_{\mathfrak{M}, t_k} \circ \dots \circ GEN_{\mathfrak{M}, t_{i+1}}(m, \delta_{\mathfrak{M}, i}) = \langle m, \delta_{id}(m, \delta_{\mathfrak{M}, i}) \rangle$ and thus $\delta_{\mathfrak{M}, i}(m) = \delta_{\mathfrak{M}, k}(m)$. We also know that all the transformations between the prefixes, i.e., all transformations t_l for each l with $i < l \leq k$, do not act like the identity function for their inputs, i.e., $GEN_{\mathfrak{M}, t_l}(m, \delta_{\mathfrak{M}, l-1}) \neq \langle m, \delta_{id}(m, \delta_{\mathfrak{M}, l-1}) \rangle$. 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, or they would not be consistent afterwards, which conflicts with the assumed correctness of the transformations.

Thus, each transformation t_l ($i < l \leq k$) 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 (analogous argumentation for the removal). Then there is a model that contains the element after applying the change generated by the transformation, i.e., $\exists s \in \{1, \dots, n\} : \exists e : e \in \delta_{M_s, i+1}(m_s) \setminus \delta_{M_s, i}(m_s)$. 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_l with $i + 1 < l \leq k$ which, in turn, removes this element, i.e., $e \in \delta_{M_s,l-1}(m_s) \setminus \delta_{M_s,l}(m_s)$. Then $e \in \delta_{M_s,l-1}(m_s) \setminus m_s$ and thus $\delta_{M_s,l-1}(m_s) \setminus m_s \not\subseteq \delta_{M_s,l}(m_s)$, which conflicts the definition of monotone transformations for t_l .

In consequence, each transformation t_l ($i < l \leq k$) can neither add nor remove an element, thus our assumption that two prefixes that yield the same inconsistent models does not hold, which proves the lemma. \square

With that insight, it is easy to see that given only monotone transformations, no alternation can occur in our algorithm Algorithm 7.

Monotone transformation prevent alternation

Theorem 7.13 (Monotone Transformations Prevent Alternation)

Let \mathbb{t} be a set of correct, monotone synchronizing transformations. Then the execution of Algorithm 7 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 of $\delta_{\mathfrak{M},\text{generated}}[]$ in the transformation execution loop in Lines 8–14 of Algorithm 7 can never contain the same two changes. This would, however, be necessary to fulfill Definition 7.6 for alternation. \square

Stronger guarantee than non-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 give the guarantee of non-alternation, but monotony according to Definition 7.7 is not a property that we cannot 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.

Monotony
not
generally
assumable

Still, the given examples were rather artificial, so they cannot be seen as an indicator for monotony to be not practically achievable. It may, at least in some cases, be possible to specify transformations that are monotone. Even if only some of the transformations are monotone, or if 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.

Ensuring
monotony
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.

Combining
alternation
avoidance
and
detection

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.

Algorithm 7 may not terminate, because it generates an infinitely long orchestration, thus never leaving the loop in Lines 8–14. Thus, 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 for the number of necessary executions 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 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. That bound will be well-defined, such that we can reasonably assume that no more execution are practically necessary.

In the following, we propose design goals for a conservative application algorithm and the so called *provenance algorithm* itself and finally prove its correctness and termination properties. That algorithm was developed together with Joshua Gleitze in a scientific internship and published in [GKB20].

Always
terminating,
correct
algorithm

Artificial
execution
bound
drawback

High
number of
executed
transformation
unwanted

Design
goals of
orchestration

7.4.1. Design Goals

An adapted version of Algorithm 7 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 8–14, needs to be defined.

Degrees of freedom

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 for that in practice. 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.

Importance of execution order

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.” [GKB20]

Incremental consistency restoration

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 those transformations instead of new 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 improves identifying 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 $\mathbf{t}_{exec} \subseteq \mathbf{t}$, then

Principle benefits

there is some transformation $t \in \mathbb{t}_{exec}$ that was executed for its first time last. Thus, the algorithm found an orchestration of $\mathbb{t}_{exec} \setminus t$ such that the models were consistent to all those transformation, but it was not able to execute t and the transformations in \mathbb{t}_{exec} afterwards, such that the models become consistent to all these transformations. 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 $t \setminus \mathbb{t}_{exec}$, 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 t is somehow conflicting with the other transformations in \mathbb{t}_{exec} . This does not necessarily mean that there is something wrong with t , but only that also considering that transformation does either induce 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 $\mathbb{t}_{exec} \setminus t$ can be used as a starting point to either identify the occurring problem or to manually restore consistency of the models.

Principle
not
improving
optimality

If the algorithm operates according to the introduced principle and is not able to preserve consistency anymore after it considers an additional transformation t , 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 t earlier or integrating further transformations in t before ensuring consistency to all transformations in \mathbb{t}_{exec} can, of course, lead to the situation that the algorithm finds a consistent orchestration. This can reduce optimality of the realized orchestration function, but we claim the discussed benefits to outweigh that.

Transforma-
tions
reacting to
others

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 set of transformations. 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. It should, however, not be necessary that a transformation must be executed again after the other transformations reacted the execution of that transformation. 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.

Definition 7.8 (Reactive Converging Transformations)

A set of synchronizing transformations t is *reactive converging* with respect to models m and changes δ_m if in any orchestration of t in which the last transformation $t \in t$ of that sequence was executed after all other duplicate-free orders of transformations in that orchestration, the yielded models are consistent to t .

The property does not require that the other transformations were executed in each order consecutively, but only that the orchestration contains each order of those transformations, but potentially with other transformations in between. As an example, assume a set of transformations $\{t_1, t_2, t_3\}$, which is reactive converging for some input of models and changes. After executing them for these models and changes in the order $[t_1, t_2, t_3, t_3, t_1, t_2, t_3]$, 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 $[t_2, t_1]$.

Reactive
converging
transforma-
tions
example

7.4.2. The Provenance Algorithm

We propose an algorithm that realizes the discussed design goal with the function PROVENANCEAPPLY in Algorithm 8. The algorithm is a derivation of the general algorithm implementing an application function depicted in Algorithm 7. 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, the orchestration itself is implemented differently in a recursive rather than an iterative manner, which implicitly ensures termination.

Recursive
orchestra-
tion
algorithm

The function PROPAGATE implementing the orchestration in a recursive manner acts as follows: It selects one of the transformations as a candidate

Incremental
consistency
restoration
principle

Algorithm 8. The provenance algorithm. Adapted from [GKB20].

```
1: procedure PROVENANCEAPPLY( $\mathbb{t}, \mathbb{m}, \delta_{\mathbb{M}}$ )
2:    $isConsistent \leftarrow \text{CHECKCONSISTENCY}(\mathbb{t}, \mathbb{m})$ 
3:   if  $\neg isConsistent$  then
4:     return  $\perp$ 
5:   end if
6:    $\delta_{\mathbb{M},res} \leftarrow \text{PROPAGATE}(\mathbb{t}, \mathbb{m}, \delta_{\mathbb{M}})$ 
7:   if  $\delta_{\mathbb{M},res} = \perp$  then
8:     return  $\perp$ 
9:   end if
10:  return  $\delta_{\mathbb{M},res}(\mathbb{m})$ 
11: end procedure

12: procedure PROPAGATE( $\mathbb{t}, \mathbb{m}, \delta_{\mathbb{M}}$ )
13:    $\mathbb{t}_{executed} \leftarrow \emptyset$ 
14:   for  $\mathbb{t}_{candidate} \in \mathbb{t} \setminus \mathbb{t}_{executed} \mid \delta_{\mathbb{M}}.\text{affects}(\mathbb{t}_{candidate})$  do
15:      $appResult \leftarrow \text{GEN}_{\mathbb{M},\mathbb{t}_{candidate}}(\mathbb{m}, \delta_{\mathbb{M}})$ 
16:     if  $appResult = \perp$  then
17:       return  $\perp$ 
18:     end if
19:      $\langle \mathbb{m}, \delta_{\mathbb{M},candidate} \rangle \leftarrow appResult$ 
20:      $\delta_{\mathbb{M},propagation} \leftarrow \text{PROPAGATE}(\mathbb{t}_{executed}, \mathbb{m}, \delta_{\mathbb{M},candidate})$ 
21:     if  $\delta_{\mathbb{M},propagation} = \perp$  then
22:       return  $\perp$ 
23:     end if
24:      $\mathbb{m}_{propagation} \leftarrow \delta_{\mathbb{M},propagation}(\mathbb{m})$ 
25:      $isConsistent \leftarrow \text{CHECKCONSISTENCY}(\mathbb{t}_{candidate}, \mathbb{m}_{propagation})$ 
26:     if  $\neg isConsistent$  then
27:       return  $\perp$ 
28:     end if
29:      $\delta_{\mathbb{M}} \leftarrow \delta_{\mathbb{M},propagation}$ 
30:      $\mathbb{t}_{executed} \leftarrow \mathbb{t}_{executed} \cup \{\mathbb{t}_{candidate}\}$ 
31:   end for
32:   return  $\delta_{\mathbb{M}}$ 
33: end procedure
```

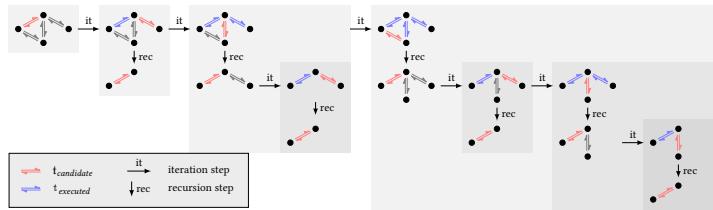


Figure 7.7.: Exemplary execution of the provenance algorithm for a change in the topmost model. The transformations present to the current execution of PROPAGATE, as well as the executed and candidate transformations $t_{executed}$ and $t_{candidate}$ are depicted for each iteration (horizontal) and recursion step (vertical).

to execute next. This selection ensures that a transformation is selected whose models are affected by any already performed change, such that the transformation needs to perform changes. Models are affected by a change if any of the two changes in δ_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 $t_{executed}$. After that recursive execution it is checked whether the models yielded by the resulting changes are still consistent to the candidate transformation. If this consistency check fails, the transformations do not fulfill the definition of being reactive converging according to Definition 7.8, as we will prove later. If the 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 transformations, the recursive execution ensures consistency to all yet executed transformations by applying all yet 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 $t_{executed}$ is considered. Thus, the set of transformations gets smaller in each recursive call of PROVENANCEAPPLY.

Exemplary
algorithm
execution

7.4.3. Correctness, Termination and Goal Fulfillment

The provenance algorithm was 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 does always terminate 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 4.12 and Definition 4.13.

Theorem 7.14 (Provenance Algorithm Termination)

Algorithm 8 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 14 and the recursive call in Line 20. 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 thus 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 precondition is not fulfilled. If the given set of transformations is empty, the loop is not entered and thus no recursive call is performed. In consequence, the recursion depth is never higher than $|\mathbb{T}|$. \square

Theorem 7.15 (Provenance Algorithm Correctness)

Algorithm 8 realizes a correct application function according to Definition 4.13.

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 τ , 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 τ . Thus, PROPAGATE implicitly implements an orchestration function according to Definition 4.10 and applies the transformations in the determined order to calculate the result delivered by PROVENANCEAPPLY. Thus PROVENANCEAPPLY fulfills Definition 4.12 for an application function.

Let us assume that Algorithm 8 does not realize a correct application function. PROVENANCEAPPLY may return \perp in Line 4 or Line 8, or it may return models in Line 10. Correctness requires the function to either return \perp or consistent models, which may only be violated by PROVENANCEAPPLY by returning models that are not consistent. This means that for some input models and changes, PROVENANCEAPPLY returns models m_{res} , such that there is a transformation $t \in \tau$ to which m_{res} , or more specifically two models m_i and m_k , whose metamodels are related by t , are not consistent. We distinguish three cases:

1. t was never executed by PROPAGATE. This means that the changes δ_{M_i} and δ_{M_k} in δ_M of the two models that are kept consistent by t were always empty, i.e., δ_{id} , because otherwise t would have been selected in the loop header. Since the initial models m_i and m_k were consistent to t , the returned models are still consistent, because only the identity function is applied to them.
2. t was executed and no other transformation that involves m_i or m_k was executed afterwards. Then the returned models are consistent by definition of correctness for t .
3. t was executed and another transformation $t' \in \tau$ that involves m_i or m_k was executed afterwards. Since t' was executed after t , t was in $\tau_{executed}$ when t' was the candidate transformation $t_{candidate}$. Thus, t is executed in the recursion after the first execution of t' , thus the result is consistent to t and because of the check in Line 26 after returning from the recursion also to t' . Thus, the returned models are consistent to both t and 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. \square

Algorithm complexity

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 8 executes transformation at most $O(2^{|\mathbb{T}|})$ 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 13) and grows by one transformation every iteration of the loop (Line 30). It follows that the recursive call in Line 20 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:

$$T(m) = m + \sum_{i=0}^{m-1} T(i) = 2 + 2 T(m-1) = 2 * (2^m - 1) \in O(2^m)$$

$$T(0) = 0$$

\square

Design principle fulfillment

Finally, the algorithm was constructed in order to implement the principle defined in Subsection 7.4.1 to ensure consistency among the transformations that have already been executed before executing a transformation that has not been executed yet.

Theorem 7.17 (Provenance Algorithm Design Principle)

Algorithm 8 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 20, the current model tuple $\mathbf{m}_{propagation}$ is consistent to all executed transformations in $\mathbb{T}_{executed}$, according to Theorem 7.15. \square

We have given Definition 7.8 for specifying 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 if all other transformation have been executed afterwards, as we assume this to be a reasonable property that induces an upper bound for the number of transformation executions. We have used that property as a design goal for the proposed algorithm and can now show that the algorithm actually always returns consistent models when the transformations fulfill that property, which means that the algorithm implements an optimal application function.

Theorem 7.18 (Provenance Algorithm Optimality)

If the transformation set τ passed to Algorithm 8 is reactive converging according to Definition 7.8 and if the consistency preservation rules of all transformations in τ are total functions, the algorithm implements an optimal application function.

Proof. We show that the algorithm does not return \perp 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 \perp when the input models are inconsistent, returning \perp in Line 4 is valid. The algorithm returns \perp in Line 8 if PROPAGATE returns \perp , thus we show that PROPAGATE does not return \perp . PROPAGATE returns \perp in Line 17 if the application of a selected transformation in Line 15 returns \perp . By assumption, all transformations are total, thus the application can never return \perp . PROPAGATE returns \perp in Line 22 if a recursive call returns \perp . If the loop in that recursive call is executed, the arguments for not returning \perp apply recursively. If the loop is not executed in the recursion than the input changes are returned, which do not yield \perp .

Finally, PROPAGATE returns \perp in Line 27 if the models yielded by the recursive call are not consistent with the transformation that is the candidate $t_{candidate}$ in that loop iteration. Since the transformation set is reactive converging, this can only be the case if not all orders of the transformations currently in $\tau_{executed}$ have been executed yet. We show that all transformations in

$t_{executed}$ have been executed by induction. In the first iteration of the loop only the candidate of that iteration is executed and $t_{executed}$ is empty, thus the statement is trivially true. Let us assume that in a loop iteration with $|t_{executed}| = i - 1$ all orchestrations of transformations in $t_{executed}$ have been executed in Line 27, but that in the following loop iteration with $|t_{executed}| = i$ this is not true. This means that there is an order $[t_1, \dots, t_i]$ of the transformations in $t_{executed}$, in which they have not been executed yet. Let t be the candidate $t_{candidate}$ of the last iteration with $|t_{executed}| = i - 1$. Let k be the index of t in that sequence, i.e., $t = t_k$. Then per induction assumption the sequence $[t_1, \dots, t_k]$ has been executed in one of the previous iterations of the loop. Afterwards t was executed in Line 15. Additionally, the sequence $[t_{k+1}, \dots, t_i]$ has been executed in the recursive call in Line 20 by induction assumption. Hence, the transformations have been executed in the order $[t_1, \dots, t_i]$, which is a contradiction to our assumption.

In consequence, PROPAGATE never returns \perp and thus PROVENANCEAPPLY does not return \perp , except for inconsistent input models. Since we have already proven that the algorithm terminates always and implements a correct application function, it implements an optimal application function. \square

Optimality assumptions fulfillment

Optimality can, however, only be guaranteed under rather specific conditions. Apart from the necessary fulfillment of the property to be reactive converging, the transformations need to be able to handle every input, thus 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 will usually 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.

Orchestration problem non-existence

Although this means that under that 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 for these assumptions.

Well-defined requirements

Finally, it is an open question how far we can assume sets of transformations to be reactive converging in practice. We did, however, not introduce this as a property that should be fulfilled by transformations, as it is obviously hard to ensure or even analyze that 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 8 does only return \perp in those situations and thus does 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 easily retrieved.

The algorithm may fail because it is, at some point, not able to execute a candidate transformation (Line 17), 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 that case, we at least know that after the previous loop iteration consistency to all yet executed transformations could be achieved.

Whenever the PROPAGATE function fails and returns \perp , we know that for the current transformations in $t_{executed}$ an orchestration exists that yields the current changes in δ_M , for which we know that applied to the original models the result $\delta_M(m)$ is at least consistent to $t_{executed}$. We also know that the algorithm was not able to ensure consistency to the current candidate transformation $t_{candidate}$. 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 $t_{candidate}$, $t_{executed}$ and the current changes δ_M reduces the necessary model states and transformations to consider for a transformation user or developer to identify why no consistent orchestration was found.

Information about failure state

Reasons for algorithm failing

Relevant failure state information

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 made until the algorithm fails, because less models may have been modified until the algorithm fails.

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 of such a function, which ensures that an application function always delivers consistent models if there is an order of the transformations that leads to those models. From that 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 has a well-defined order and bound for the transformation execution and, rather than improving optimality, ensures that in cases when no consistent models can be derived at least some information can be provided that helps developers or user of transformations to identify why no consistent models were found. We conclude this section with the following central insight.

Insight 4 (Orchestration)

We have proven that whether an orchestration of modular and independently developed transformations exists that restores consistency for given models and changes, denoted as the *orchestration problem*, is undecidable. We have shown that even impractical restrictions to the individual transformations do not make the problem decidable, such that we need to accept that the problem is undecidable. In consequence, any algorithm that realizes an application function for transformations can only implement a conservative approximation of the orchestration problem. Due to this conservativeness, any 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 yet delivered changes can help the transformation developer or user to identify why the algorithm failed. We found that this can be achieved with a strategy of iteratively restoring consistency for subsets of the transformations, such that always a subset of the transformation for which consistency could be restored as well as a transformation for which it could not be restored anymore can be given to improve reasoning about the cause for failing. We have proposed an algorithm that implements that strategy and is proven to fulfill that property.

Discuss error as the result of failing in achieving correctness

Proposal for specification levels and categorization of errors

Subordinate contributions

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 the transformation network, namely the consistency relations, the consistency preservation rules and the application function. It may, however, easily occur that a transformation developer defines 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 thus 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 the 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 approaches to achieve them that we have introduced so far.

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 yet 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 [Kla+19] and are part of the following discussion in revised form.

8.1. Knowledge Levels in Transformation Network Specification

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 concrete 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 at each of them specific kinds of mistakes can be made and that a formalism may ensure

Level	Name	Correctness	Knowledge
1	Transformation	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.

correctness with respect to one of those levels and the ones below, whereas 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, which we discuss in the following. At the *transformation level*, we consider the specific properties, especially synchronization, of a single transformation to be used in a, more precisely any, transformation network. 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. On the other hand, a transformation can be correct at the transformation level without being correct at the relation and network rule level.

Three distinguished levels

These levels are especially different 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 concrete 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.

Required knowledge per level

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 [Kla+19]. In that work, we have discussed the *global* level, which considers the global knowledge in terms of the overall, n -ary 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 make sense to explicitly consider correctness according to it.

Level 1 (*Transformation*): At the first level, we do 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 concrete 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 the consistency relations being compatible.

Level 3 (*Network Rule*): At the third level, we consider the knowledge about the complete transformations of a concrete network, thus especially also the consistency preservation rules that preserve consistency. We have discussed in Chapter 7 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 an option from different possibilities provided by a consistency relation to restore consistency.

Levels for
modular
consistency
notion

Knowledge
about
single trans-
formation

Knowledge
about
consistency
relations of
network

Knowledge
about
complete
transforma-
tions of
network

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 does 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. Rather declarative, usually bidirectional transformation languages already relieve the developer from specifying consistency preservation rules and lifts the abstraction to consistency rules, from which consistency preservation rules are automatically derived. Some of those language even relieve the developer from manually ensuring synchronization, for example, 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 must only deal with potential problems arising at the higher levels. Integrating an analysis for compatibility, such as the one proposed in Chapter 5, to QVT-R could thus also abstract from the network relation level.

Languages that ensure correctness at higher levels than the transformation level are currently unusual. This requires either 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 [TA15; TA16] or to Multi Graph Grammars (MGGs) [KS06] 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

Inherent correctness by formalism

Inherent correctness at transformation level

Formalisms for higher-level correctness

language [Gle17], which we present in detail in ??, also lifts the abstraction such that the relation and network rule level must not 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, i.e., theoretically the network relation level does not require correctness at the transformation level. It does, however, not make sense from a knowledge perspective to ensure compatibility of relations when the transformations that ensure them 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 already discussed in Chapter 7. We then consider the *mistakes* and the resulting *faults* in the transformation specifications, which a transformation developer can make. The mistakes are specific for the yet introduced conceptual specification levels, thus we derive them from those levels. We finally relate the mistakes to the failures that can occur when transformation networks containing faults that are 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 concrete transformation or the necessity to ensure certain properties. For example, the missing knowledge that transformations need to 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 the transformation network can lead to the mistake that incompatible transformations are realized. If a transformation language abstracts from a specific conceptual level and thus relieves the developer from ensuring that no mistakes at that level are made, a faulty implementation of the language can, of course, also have faulty behavior because of those mistakes, if they were made by the transformation language developer. We do, however, not consider that case explicitly.

Erroneous knowledge

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 match 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.

Erroneous implementation

Failure: A failure occurs at execution time of the transformation 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

Manifestation of fault during execution

every fault must lead to a failure. On the other hand, a fault can also lead to several failures, for example, 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

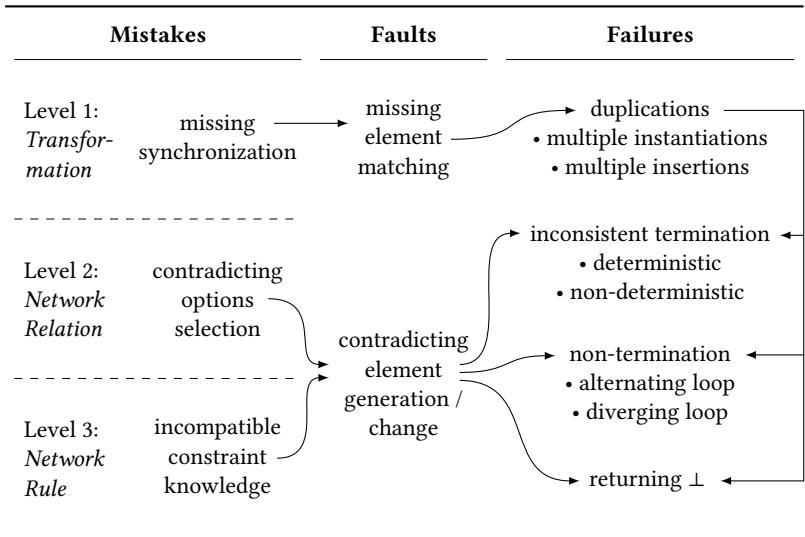
Failures are the manifestation of faults during transformation execution and thus the final result of mistakes made by a transformation developer. A failure means that the transformation network or more precisely the execution of the application algorithm reached an unwanted state. We have already discussed 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 by returning \perp . 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 actually contain faults that lead to the algorithm not being able to find 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 combination of them) exist for being consistent, i.e., whether corresponding condition elements are missing or whether there are too many condition elements for which no consistent models can be found by adding further ones. We did, however, find in [Sag20, Table 5.7] that the 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 lead to both the return of inconsistent models or non-termination,

Relation to
other error
notions

Essential
failure
types

Specializa-
tion options
for failure
types

**Figure 8.1.:** Categorization and dependencies of mistakes, faults and failures.

depending on whether the application algorithm aborts or not. Duplications of elements are of special 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 , it can, however, be an indicator whether the algorithm fails because an artificial execution bound was reached or because a transformation cannot

Failures depending on application algorithm

be applied anymore, because 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 yielded by the application of transformations in the order determined by the orchestration function 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, which yield different inconsistent models.

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 or a string being appended repeatedly, or an infinite number of elements is 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 \perp : The application algorithm may terminate and return \perp 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 mistakes 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 \perp rather than returning inconsistent models or not terminating at all, but still it is not the intended result which may be caused by an actual mistake made by the transformation developer.

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*, which can occur because different consistency preservation rules instantiate an element multiple times, although all of them represent the same one, and *multiple insertion*, which 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 also lead to inconsistent termination, non-termination, or returning \perp , because either the algorithm returns after a finite number of transformation executions without checking consistency or checking it and returning \perp , or the transformations are not able to restore consistency for the models 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 having any other failure, especially not the more specific duplications. Knowing the other failures and their relation to the causal mistakes is still important. First, when a transformation network with such an application algorithm yields \perp in most cases, 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 those transformations do not expect the synchronization scenario and thus assume that consistency is achieved by construction, i.e., that changes must only be processed for one model and thus 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 and eventually to different kinds of failures during transformation execution. In the following, we derive mistakes and faults from the specification levels, depicted in Figure 8.1.

We explicitly focus on conceptual mistakes and faults concerned with the development of transformation networks. This especially excludes 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 they are part of research regarding the individual bidirectional transformations on their own. Mistakes regarding synchronization of transformations, i.e., the case that changes were performed to both models, are, however, relevant.

In fact, technical mistakes usually also 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

Mistakes by specification levels

Excluded mistake types

Correctness by synchronizing transformations

Missing knowledge about synchronization

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, like we have discussed in Section 4.2. First, the relations must reflect some intended, probably informal notion of consistency. If the relations miss to reflect some of the constraints of that consistency notion or if they reflect additional constraints that are not part of that notion, the relations may be considered incorrect. Second, the relations have to 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.

Correctness by combination or individual

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 single transformation to be already correct. Finally, such incorrectness would not be even interesting, because additional constraints do not lead to failures but, in the worst case, only lead to not finding consistent models although they exist, and missing constraints simply lead to inconsistent models, as the result does not fulfill the constraints of the existing, informal notion of consistency.

Individual correctness irrelevant

The relevant correctness notion is the one of compatibility. A transformation developer, or potentially a group of them, can make the mistakes 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

Correctness requires compatibility

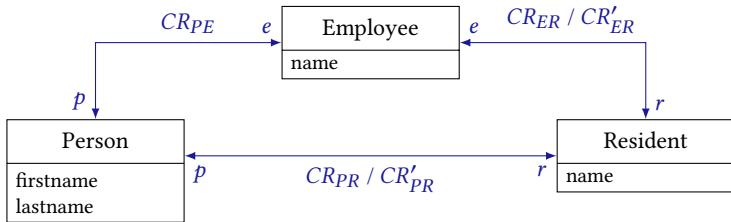
whether it checks consistency. The inconsistency will be deterministic or not, according to whether the execution order of transformations is fixed or not. If the algorithm does not implement such an artificial bound, such a fault can also lead to non-termination of the algorithm, because the execution 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 which are not in the overlap of those 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.

Since we did not find and define a property that a set of transformations and especially their consistency preservation rules have to fulfill and instead concluded to deal with the orchestration problem by means of a conservative application algorithm, we cannot give a reasonable or even complete overview of potential mistakes transformation developers can make at this level.



$$CRPE = \{(p, e) \mid p.firstname + " " + p.lastname = e.name\}$$

$$CRPR = \{(p, r) \mid p.firstname + " " + p.lastname = r.name\}$$

$$CR'PR = \{(p, r) \mid p.lastname + " " + p.firstname = r.name\}$$

$$CRER = \{(e, r) \mid e.name = r.name\}$$

$$CR'ER = \{(e, r) \mid e.name.toLowerCase = r.name\}$$

Figure 8.2.: Adaptation of consistency relations from the extended running example in Figure 5.1.
Adapted from [Kla+19, Figure 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 \perp , depending on how the application algorithm operates. Thus, these dependencies do not give an 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 will further discuss the detection and avoidance of mistakes in the subsequent section.

Mistakes,
faults,
failures de-
pendencies

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

Mistake
examples at
each level

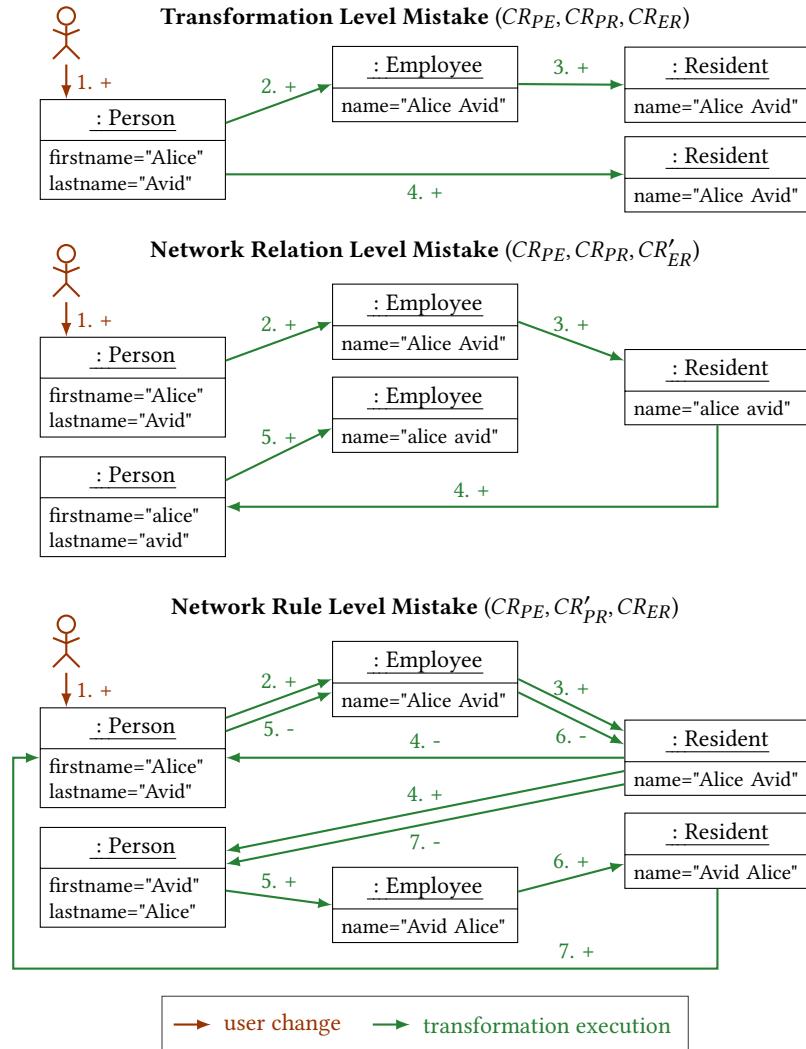


Figure 8.3.: Examples for transformation executions based on the consistency relations given in Figure 8.2 with mistakes at each of the three levels. Numbers indicate transformation execution steps, +/- indicate element addition or removal. Adapted from [Kla+19, Figure 5].

the order depicted by the numbers at the transformations. The creation of an element is denoted by a “+”, whereas the removal of an element is denoted by a “-”. In one transformation step, multiple elements may be removed or created. The arrows indicate that the change of the source element leads to the addition or removal 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 fault 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 transformation may not be able to process the occurring situation, or, as discussed before, assumes consistency by construction and thus identifies the models as consistent although they are actually 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, in fact 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 first and last name 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 remove other elements in the other model for which no corresponding element exists, this leads to the

Missing synchronization

Incompatibility

Contradicting option selection

repeated insertion of persons, employees and residents with first and last name concatenated in one way and the removal of them with the swapped 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 transformation, thus selecting an option from the consistency relations that is not in the overlap with consistency relations of the other transformations.

In fact, 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 mapping, although in the original knowledge the consistency relations contained these two 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

There are two ways to deal with the possibility of errors in transformation networks. 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 of the transformation network 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 one proposed in Section 7.4, may terminate returning \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.

Error avoidance vs. detection

Detection from categorization

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 on a specific scenario. In fact, a network without mistakes does not mean that no failures occur, because the application algorithm can always fail because

Failure absence indicating mistake absence

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 ??, 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 avoids faults at the network levels and thus avoids the possibility that transformation developers can make the discussed mistakes. We will also discuss in ?? an approach how networks of such a topology can be constructed without the necessity that the direct transformations between the metamodels must have such a specific topology. Thus, it is also possible

Mistake avoidance
at different
levels

Avoidance
at transfor-
mation
level

Avoidance
by network
topologies

to avoid such mistakes by construction, but this limits the networks we can define to specific topologies.

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 the relations between mistakes and failure types may at first not sound interesting, because all mistakes at the two network levels can lead to any kind of failure, depending on how the algorithm works. 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 process of identifying 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 the algorithm with one with different properties. Thus, if the algorithm does not terminate, it can be useful to introduce an artificial execution bound to be able to produce an inconsistent state of the models. These inconsistent models can also be retrieved from a conservative algorithm as proposed in Section 7.4, which is specifically developed to improve the ability to find the reasons for the algorithm not finding consistent models.

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.

Tracing
between
mistake and
failure
types

Identifying
failure
cause by
model state

Algorithm
exchange to
identify
mistakes

Duplica-
tions
indicate
missing
synchro-
nization

Failure frequency indication

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 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.

Identifying failure cause by returning \perp

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 be just the conservative behavior to avoid non-termination because of the undecidability of the orchestration problem.

Unique identification of fault existence

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 does also ensure 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 different levels, we have categorized the possible mistakes, faults and failures that can occur in such a network and discussed which of them can be avoided or detected. We have discussed

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 is failing. We did, however, not discuss or 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 his or her models consistent and detects failing transformations, as well as the transformation developer, who is responsible for correcting the potential faults in the implementation. Such a process discussion is out of the scope of this thesis and thus referred to as future work (see Subsection 9.3.4).

Insight 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 terms of 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 were able to derive that mistakes regarding a single transformation cover missing synchronization, which can and has to be avoided by construction. This is especially necessary if transformations assume consistency to be achieved by construction, because then non-synchronizing transformations produce faulty results that they assume to be consistent, because they have generated them. 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 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 cases, because of non-fitting transformations, such as having incompatible consistency relations, the transformation developers need to investigate the reasons for the algorithm to fail.

Several statements validated by proof

Consistency and correctness notion

Compatibility evaluation

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 it for given consistency relations 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 is not able to 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 not 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 finally 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 [Kla+19; Kla+20a; GKB20]. The case studies for our error categorization and achievement of synchronization have been conducted in two Master’s theses [Sym18; Saǵ20]. 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.

Publication
of
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 that 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 that formal approach by Theorem 5.11, Theorem 5.6 and Corollary 5.12, such that we do not need to further evaluate its correctness. 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 [Kla+20a] and that was developed in the Master’s thesis of Pepin [Pep19].

Evaluated
properties

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, thus not introducing further developer effort, and improves the ability of the transformation network to find consistent models after changes. Thus, the correctness and the applicability of the approach are of special importance.

Correctness
and applica-
bility

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?
<i>Metric 1.1.1:</i>	<i>Precision: Ratio between true positives and true plus false positives</i>
Question 1.2: (Applicability)	How often does the analysis not prove compatibility although it is given?
<i>Metric 1.2.1:</i>	<i>Recall: Ratio between true positives and true positives plus false negatives</i>

Table 9.1.: Goals, questions and metrics for compatibility evaluation.

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 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 prove compatibility 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 that evaluation together with the answered questions and 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

Empirical
evaluation
in case
study

Correctness
evaluation

algorithm is thus not allowed to produce false positives, which is why we consider the *precision* metric:

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

This metric needs to be 1, as otherwise the algorithm produces false positives and would thus, per definition, be incorrect. In consequence, correctness of the algorithm directly correlates with that 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 is not able to prove compatibility in most cases in which it is actually given, applicability is reduced as the usefulness of the results for a transformation developer is limited. For that reason, we analyze the *recall* metric:

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

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 that 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 those are 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 presented in Section 5.4 resulted in the implementation of a prototype, which is available in a GitHub repository [Pep]. The prototypical implementation is specific to QVT-R and the OCL constraints 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, thus 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 only consider valid inputs anyway. Finally, the implementation operates completely 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 QVT-R for defining transformations by the Model-driven Architecture (MDA) [Obj14a], and, on the other hand, by the fact that consistency relations are explicitly defined in QVT-R, especially in comparison to imperative languages. We based the implementation on EMF and its Ecore meta-metamodel (see Sub-section 2.2.2) as one of the most common and technically mature modelling frameworks. Within EMF, implementations of transformation languages are provided through the *Eclipse MMT* [Ecl20c] project. In particular, the contained QVT Declarative (QVTD) [Ecl20b] language provides a parser for

QVT-R transformations, which, in turn, uses *Eclipse OCL* [Ecl20a] 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 Satisfiability Modulo Theories (SMT) solver. Most 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 [MB08], which is an SMT solver, which can be used in Java code and supports a large number of theories.

SMT tool selection

9.1.3. Case Study

We have applied our just presented prototypical implementation in a case study to 19 scenarios. 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.

Scenarios of QVT-R transformations

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, thus having 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, respectively.

Ground truth for scenarios

The scenarios were specifically developed for the evaluation of the approach, thus reflecting as many kinds of relations that can be expressed with QVT-R as possible and thus 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 [Pep].

Evaluation-specific scenarios

#	Scenario Description	Compatible
1	Three equal String attributes of three metamodels	✓
2	Six equal String attributes of three metamodels	✓
3	Concatenation of two String attributes	✓
4	Double concatenation of four String attributes	✓
5	Substring in a String attribute	✓
6	Substring in a String attribute with precondition	✓
7	Precondition with all primitive datatypes	✓
8	Absolute value of Integer attribute with precondition	✓
9	Transitive equality for three Integer attributes	✓
10	Inequalities for three Integer attributes	✓
11	Contradictory equalities for three Integer attributes	✗
12	Contradictory inequalities for three Integer attributes	✗
13	Constant property template items	✓
14	Linear equations with three Integer attributes	✓
15	Contradictory linear equations for three Int. attributes	✗
16	Emptiness of various OCL sequence and set literals	✗
17	Equal String attributes for four metamodels	✓
18	Transitive inclusions in sequences	✓
19	Comparison of role names in three metamodels	✓

Table 9.2.: Example scenarios of consistency relations and their compatibility, from [Kla+20a, Table 3].

	Classified Compatible	Unclassified
Compatible	12	4
Incompatible	0	3

Table 9.3.: Compatibility classification of scenarios from Table 9.2 by our approach, from [Kla+20a, Table 4].

9.1.4. Results and Interpretation

We applied the prototypical implementation of our practical approach to prove compatibility 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

We have already discussed that correctness in terms of operating conservative is proven for the formal approach. Since the practical approach is derived from that 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 QVT-R 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 variable have to be considered. In particular, all template expressions need to be considered for the construction of the property graph, namely property template items, preconditions and invariants.

By construction of the approach presented in Section 5.4, we ensure that all 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 intended, not identifying consistency relations as compatible although they are not. This results in a precision value of 1:

$$\text{precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}} = \frac{12}{12 + 0} = 1$$

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 was the case, constraints would have been missed, which could have resulted in identifying consistency relations

Requirements for correctness by construction

Metric evaluation shows correctness

Answer to evaluation question

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 being able to prove compatibility, although the given relations are actually consistent. In particular, conservative behavior of the approach can occur for two reasons:

Redundancy Notion: Compatibility of consistency relations is proven by identifying relations that follow the definition of left-equal redundancy, as introduced in Definition 5.9. Since this redundancy notion is not proven to be the weakest one that is compatibility-preserving, it may be a too strong requirement for identifying compatibility-preserving consistency relations.

Redundancy Undecidability: Definition 4.17 for consistency relations relies on an extension 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. Consistency relations as defined in Definition 4.17 are extensional specifications and thus usually enumerate infinite sets of elements, which are impossible to compare programmatically. OCL is, however, in general undecidable, because it can be transformed into first-order logic [BKS02].

In particular, the number of quantifiers within a formula influences decidability. Since variables in consistency relations are translated to existentially quantified formulae, the number of variables in a consistency relation is crucial for deciding its satisfiability. Not all available OCL constructs may be necessary to describe relevant consistency relations, still constructs involving operations on sets and strings are especially problematic, because operation on collections are transformed into quantified formulae and strings provide problematic OCL operations. For example, `toUpper` and `toLower`, which we have also used in our running example, cannot be easily 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,

so we cannot even systematically evaluate which kinds of relations can be analyzed.

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 %.

$$\text{recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} = \frac{12}{12 + 3} = 0.8$$

This is a first indicator for high applicability of the approach, as it was able to prove compatibility in most of the cases in which the relations were actually 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, roles 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 logic 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 and led to unprovability by the used SMT solver. Most importantly, however, this limitation does only concern 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

Metric evaluation show high recall

Reasons for false negatives

Answer to evaluation question

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 will 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, or more precisely their underlying consistency relations, during their individual development or when combining them to a network. In Subsection 5.1.2 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 will 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 the reasons for them when running the analysis after a failure occurs. Due to its full automation, no further effort than running the analysis is necessary. Additionally, a manual process of ensuring compatibility or finding incompatibilities requires manual alignment of transformations with each other or the definition of test cases, which are only able to validate compatibility, but not to verify it. Thus, such manual techniques can only make existentially quantified statements about the existence of incompatibilities, whereas our approach can make universally quantified statements about their absence.

Finally, even if the proposed approach had a high degree of conservativeness, i.e., if it produces a higher degree of false negatives in other scenarios than in our evaluation, the approach still provides benefits. First, the approach may

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. If this is often necessary, it would be possible to define an interactive approach in which the removal of redundant relations by proof of the approach and by user decision is combined, which we will propose as future work in the subsequent section. In such a process, the user could be asked to manually find and declare redundant relations if the automated approach is not able to 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 us appropriate insights regarding correctness and applicability of the approach. Still, due to limitations in complexity of the considered scenarios there are threats especially regarding external validity of the results. This is why we emphasized that all results only serve as an indicator for the properties to be shown.

Limitations
of external
validity

The evaluation scenarios of the case study were developed specifically for the evaluation of the proposed 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, they 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 used for the prototype and those containing incompatibilities, prevents this.

Scenario
selection

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 rather than a conceptual limitation of the approach. The actual threat in this is that more complex relations, which are currently not supported by the implementation, may not be covered by our definition of redundancy anymore. That would

Scenario
complexity

be an actual limitation not only of the implementation but also the formal approach. In consequence, this has to be further evaluated in subsequent evaluations.

Scenario size

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 anymore. 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.

Conclusion of threats

In consequence, our evaluation gives an initial indicator for the correctness and applicability of our approach based on well-selected evaluation scenarios but 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 a networks is difficult. Especially in existing networks, transformations 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 or alternation 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

Implementation and evaluation limitations

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.

Technical
realization
limitations

In future work, we will thus extend the operations for which translations to logic formulae are defined in future work, so that we can apply the approach to more sophisticated case studies. This will provide further indicators for the general applicability of the approach.

Approach
completion

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 that 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 cannot. Although this improves the effort for developing the translations, the simultaneous use of different symbolic computation tools can increase the chances of finding redundancy proofs. Additionally, it may even be beneficial to simplify the OCL statements transformed into logic formulae where possible, like discussed in 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 expressions the algorithm worked on are simpler.

Operationaliza-
tion
alternatives

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

Evaluation
limitations

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 give 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 is not able to 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 but should still be considered as they are. 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 developed from the goal of finding contradictory consistency relations that can prevent transformations from finding consistent models after changes. Additionally, it prevents from 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 already discussed that we defined the specific notion of left-equal redundancy (see Definition 5.9), which has the property of being compatibility-preserving (see Theorem 5.11). 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 unpredictable. 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 must not fulfill real-time requirements. Results should, however, still occur in an acceptable amount of time to achieve acceptance of the approach.

Optimality
of
redundancy
notion

Perfor-
mance and
scalability
evaluation

9.2. Errors, Orchestration and Synchronization

Context: categorization 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 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.

Empirical evaluation of categorization and synchronization

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 according failures to evaluate the correctness and applicability of that approach.

Empirical evaluation of orchestration problem relevance

Since the orchestration problem can always lead to the situation that an application algorithm for a transformation network cannot find consistent models by applying the transformations, we also utilize this case study to investigate how problematic the orchestration problem actually is in practice. We already know from the halting problem that just because an essential problem in software engineering is undecidable, this must not necessarily be that relevant in practice.

9.2.1. Goals and Methodology

Combining existing transformations

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 those case studies looks as follows: We combine independently developed transformations and execute existing sets of test cases developed for the individual transformations, which we extended by validations of the further models generated by the additional transformations. We then validate the failures occurring in the test case execution. The information about the failures is used to trace back to the causing faults and mistakes, such as missing matchings of elements when multiple instantiations occur. For each identified failure, we fix the causing fault and re-execute the test cases to validate whether the failure was resolved by fixing the fault.

The process is applied iteratively until no more failures occur. Since failures due to one mistake can hide failures caused by another mistake, it is possible that after fixing all faults that led to the failures in one iteration, still failures occur 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, but 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 those 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

Error identification and correction

Iterative process

Categorization completeness evaluation

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?
<i>Metric 2.1.1:</i>	<i>Classified failure ratio: Ratio between classified failures and identified failures</i>
Question 2.2: (Correctness)	Are identified failures caused by mistakes they are related to according to the categorization?
<i>Metric 2.2.1:</i>	<i>Resolved failure ratio: Ratio between resolved failures and total failures</i>
Question 2.3: (Relevance)	How relevant is each type of mistake, i.e., how likely is it to be made?
<i>Metric 2.3.1:</i>	<i>Mistake type occurrence ratio: Ratio between occurrences of faults due to each type of mistake and total occurrences of faults</i>
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?
<i>Metric 3.1.1:</i>	<i>Fail ratio: Ratio between algorithm failures due to the orchestration problem and all failures</i>

Table 9.4.: Goals, questions and metrics for categorization and orchestration evaluation.

failures, thus indicating a higher degree of completeness with a higher value with a maximum of 1:

$$\text{classified failure ratio} = \frac{\# \text{ of classified failures}}{\# \text{ of total failures}}$$

Categorization
correctness
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:

$$\text{resolved failure ratio} = \frac{\# \text{ of resolved failures}}{\# \text{ of total failures}}$$

Mistake
type
relevance
evaluation

While we 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., however often they are likely to lead to faults in actual projects, as defined in Question 2.3. This especially influences how important it is to avoid or identify specific types of mistakes. Therefore, we measure how often each type of mistake leads to a fault in the transformation implementations of the case study 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:

$$\text{mistake type occurrence ratio} = \frac{\# \text{ of faults due to mistake type}}{\# \text{ of total faults}}$$

Finally, directly related to the 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 consistency 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 failure 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 yet 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, whenever the algorithm fails we validate whether an alternative order of transformation executions would have delivered consistent models. 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. We thus measure the following metric for the ratio of failures due to the orchestration problem:

$$\text{fail ratio} = \frac{\# \text{ of failures due to orchestration problem}}{\# \text{ of total failures}}$$

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.

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?
<i>Metric 4.1.1:</i>	<i>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</i>
Question 4.2: (Completeness)	In how many cases can the approach (not) be applied?
<i>Metric 4.2.1:</i>	<i>Application ratio: Ratio of faults at transformation level that can be resolved by the approach to all faults at that level</i>

Table 9.5.: Goals, questions and metrics for synchronization evaluation.

We 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:

$$\text{success ratio} = \frac{\# \text{ of tests with resolved failures after approach application}}{\# \text{ of tests due to which approach was applied}}$$

In fact, we only count the test cases after applying the approach which failed before due to faults at the transformation level, because we are only interested

Synchronization
correctness
evaluation

in test cases that failed before. Otherwise the metrics might become larger than 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 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 the 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 synchronizing by adding matchings for existing elements. To measure this, we counted the number of faults at the transformation level that we were able to resolve to the total number of faults:

$$\text{application ratio} = \frac{\# \text{ of resolved faults at transformation level}}{\# \text{ of total faults at transformation level}}$$

Although we applied the approach for achieving synchronizing transformations after identifying those transformations 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 the conduction of the case studies presented in the subsequent section, we have used a prototypical implementation in the VITRUVIUS framework (see Subsection 2.3.2) [Kla+20b]. It support 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 user. 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. It is incremental, because it updates the existing models rather than creating new ones upon changes. It operates delta-based, as it does not receive the modified state of a model, but a delta between the old and the new state. This conforms to what we introduced as a change in

our formalism (see Definition 4.3). To achieve this, the framework records atomic changes to the models, i.e., element creations and deletion, 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 transformation to a network for keeping multiple models consistent, which is why we implemented our approaches in a case study with that framework.

The VITRUVIUS framework provides, among others, the Reactions language [Kla16, Kra17] for defining 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. We use the Reactions language for implementing consistency preservation in our case studies.

The Reactions language uses a so called *correspondence model* to identify corresponding elements according to the implicitly defined consistency relations and thus implements a witness structure according to Definition 4.18. It consists of *correspondences*, of which each contains two sets of one or more elements. It enables to trace when elements were changed to update the corresponding elements rather than always deleting and adding a corresponding element. We have discussed in Subsection 4.4.1 that this still conforms to our formalism, although we explicitly omitted any kind of trace model there.

A transformation rule defined in that language is called a *Reaction*. To give an impression of how such rules look like, an example that transforms a PCM component into a class with appropriate naming in UML among its creation is depicted in Listing 9.1. A Reaction specifies after which type of change it should be executed, which, in this case, is the insertion of a component into a repository. It may then call one or more reusable *routines* that are supposed to restore consistency according to a consistency relation. Such a routine consists of a *match* block, which checks whether a consistency relation applies and retrieves all elements involved into that relation, and an *action* block, which restores consistency after the change. In this case, the routine checks that no corresponding class already exists to avoid multiple instantiation and afterwards retrieves an appropriate package in the UML model to place the class in. It then creates a class, assigns it an appropriate

Reactions
language

Correspon-
dence
model

Example for
Reactions

```
1 reaction {
2   after element pcm::Component
3     inserted in pcm::Repository[components]
4   call {
5     val component = newValue
6     createClass(component)
7   }
8 }
9
10 routine createClass(pcm::Component component) {
11   match {
12     require absence of uml::Class
13     corresponding to component
14     val componentsPkg = retrieve uml::Package
15       corresponding to component.repository
16       tagged with "componentsPackage"
17   }
18   action {
19     val class = create uml::Class and initialize {
20       class.package = componentsPkg
21       class.name = component.name + "Impl"
22     }
23     add correspondence between component and class
24   }
25 }
```

Listing 9.1.: Reaction creating a UML class among creation of a PCM component. Adapted from [Kla+20b].

name and adds a correspondence between the elements. For the complete explanation of that example, we refer to [Kla+20b].

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, however, not only consider already executed transformations in the recursion and does not define an execution bound. In consequence, that 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 correct by construction. We have already 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 in a GitHub repository [Vitb].

Implicit
consistency
relations

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. Transformations are defined between each pair of these metamodels, based on consistency relations that we discuss in more detail in the following. 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 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

Transformations
for
PCM, UML
and Java

PCM Element	Object-oriented Design Element
Repository	Three packages: main, contracts, data types
BasicComponent	Package within the main package and a public component realization class within the package
OperationInterface	Interface in the contracts package
Signature & parameters	Method & parameters
CompositeDatatype	Class with getter and setter (or appropriate read-only property) for inner types
CollectionDatatypes	Class that inherits from a collection type (e.g., <code>ArrayList</code> in Java)
RequiredRole	Field typed with required interface in the component realization class and constructor parameter for the field in the component realization class
ProvidedRole	Component realization class of providing component implements the provided interface

Table 9.6.: Consistency relations between elements of the PCM repository metamodel and object-oriented design elements (UML/Java), adapted from [Lan17, Table 4.1].

to the one proposed by Kramer et al. [Kra+16] to exclude different types of possible biases.

The used transformations base on two sets of consistency relations. 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 selected the mapping of architectural components to classes, as the one that was studied most intensively and whose implementation is most mature. This conforms to the mapping that we already introduced in Chapter 1 and referred to throughout the preceding chapters. Second, the relations between UML and Java reflect the usually implicitly known mapping between the two languages, as both describe the object-oriented structure of a software system in a similar way.

Table 9.6 shows the relevant consistency relations between PCM models and object-oriented design, which can be reflected in both UML and Java. A

Two sets of underlying consistency relations

Relations between PCM and object-oriented design

PCM repository model consists of data types, interfaces and components, which are all contained in one repository. The repository is represented as a package structure in object-oriented design. Components are 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 any collection type. Finally, provided and required roles define that a component provides or requires an interface. Provided roles are realized by an implementation of the provided interfaces in the component realization class. A required role, on the contrary, is represented as a field in the component realization class, which must be set via constructor parameter.

The preservation of consistency between PCM and Java according to these relations 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, Section 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*, which are an abstract specification of the behavior of a service provided by a component. Consistency between these behavior specifications in PCM and their implementation in Java code was researched in detail by Langhammer [Lan17] and is one of the reasons why in this scenario consistency between PCM and Java cannot only be expressed across UML. 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.

Table 9.6 shows the relevant consistency relations between UML models and Java code. They reflect the intuitive notion of the relation between UML and

Transformations with
PCM

Behavioral
consistency
of PCM and
Java

Relations
between
UML and
Java

UML Element	Java Code Element
Package	Package
Class	Class
Enum	Enum
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 9.7.: Consistency relations between UML class models and Java code.

Java of mostly one-to-one mappings, since we do 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 also be expressed as associations in UML, as well as parameters, fields and associations, which can have multiplicities in UML that have to be expressed as collection types with an appropriate type parameter in Java if the upper bound is higher than 1.

The preservation of consistency between UML and Java according to these relations was implemented using the Reactions language within a Bachelor's thesis supervised by the author of this thesis. Like for the transformation between PCM and Java, this one was implemented to be used in one direction only and thus, especially, not to be used in a transformation network.

The implementations of all transformations are available in a corresponding GitHub repository of the VITRUVIUS project [Vita]. Each of them also contains a sophisticated set of test cases, which were supposed to test each 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

$\downarrow \text{From} / \text{To} \rightarrow$	PCM	UML	Java
PCM	-	57	40
UML	68	-	63
Java	16	49	-

Table 9.8.: 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.

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 number of Reactions in each of the six unidirectional consistency preservation rules in Table 9.8. This conforms to the number of change types each of these consistency preservation rules is able to react to. The lower number of Reactions between Java and PCM is mainly due to the fact that several elements of the PCM are 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 must react to less change types and instead make more distinctions within the routines to separate the affected consistency relations.

Complexity
of transfor-
mations

The scenarios we have 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.

Test cases
and
evaluation
scenario

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 9.9, expressing the number of test cases for each underlying consistency relation. We have split the test cases between PCM and UML into two categories, because the second case study only uses the

Existing test
cases

Consistency Relation	# of Test Cases
<i>PCM ↔ UML Core</i>	
Repository	4
Interface	2
System	2
Composite Data Type	4
Repository Component	2
Assembly Context	2
Total	16
<i>PCM ↔ UML Additional</i>	
Signature	6
Parameter	6
Attribute	6
Required Role	3
Provided Role	2
Total	23
<i>UML ↔ Java</i>	
Package	6
Class	23
Enum	14
Interface	10
Class Method + Parameter	29
Interface Method	9
Class	19
Total	110

Table 9.9.: Number of test cases for the different consistency relations in the case studies.

first of these categories. In total, we used 39 existing test cases between PCM and UML as well as 110 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, which requires more test cases. 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 existed. We simulated the construction of that system model by producing a change sequence that would have been executed 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 [Lan17; Kla+20b]. Afterwards, we have validated that the expected models, according to the defined consistency relations, were created. This is covered by five additional test cases.

Based on these test cases, we have performed two case study. 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 [Kla+19]. 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 [Vita].

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

Existing
case study
system

Two case
studies

Error with
two trans-
formations

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 thus 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 other direction processes this creation without assuming that this may not be a user change that needs to be processed, it will likely create another corresponding element, due to missing matching of existing elements.

Synchronization of transformations

The transformation between PCM and UML was developed in the context of this case study and, purposely, designed to already be synchronizing. This allowed us to get an impression of whether it is possible to develop a transformation to be synchronizing by construction. The transformation between UML and Java was pre-existing and only designed to be executed in one direction. Thus, it did neither perform matching of existing elements using implicit unique information to be synchronizing, nor matching of existing elements using explicit unique information, i.e., correspondences, to be executed in both directions without creating duplications of elements.

Used test scenarios

Based on these transformations, we conducted the already depicted process of executing the existing scenarios of test cases and case study system, identifying the occurring failures, tracing them back to the causing mistakes and then fixing the faults in the implementations to resolve the failures. We employed all test cases that we summarized in Table 9.9 without further modifications and in addition the construction simulation of the Media Store system.

Circular Network Study

Three transformations containing cycle

In the second study, we started with the results of the first study, 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

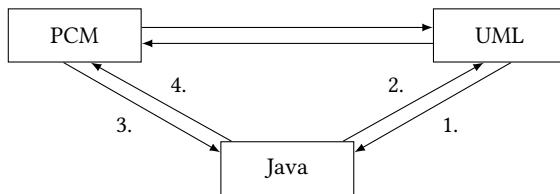


Figure 9.1.: Phases of the circular network study by depicting the transformations that are incrementally added in each of the phases.

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 not only validate consistency of the two models they were designed for, but of all three models. We employed the $PCM \leftrightarrow UML$ Core tests depicted in Table 9.9, which perform different types of changes in PCM and UML models, and extended them to also validate the generated Java models.

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 directions, and then first added the rule between PCM and Java and finally 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.

Used test scenarios

Incremental transformation addition

9.2.4. Results and Interpretation

We present the results of the introduced iterative process of identifying failures, the causing faults and mistakes, as well as fixing the faults to resolve

Aggregated mistakes, faults and failures

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 9.10.: Mistakes, numbers of faults, and number and type of faults detected in the case studies.

the failures. In Table 9.10 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 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 149 existing test cases and five 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 inverse 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 because of divergence in three cases and 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

Mainly
missing
synchro-
nization

Further
mistakes

single bidirectional transformation the unidirectional consistency relations can be incompatible. The alternation occurred in terms of repeated changes of visibilities of methods and constructors between UML and Java, because different options for mapping default visibilities exist, for which the consistency preservation rules in both directions chose contradicting ones, thus swapping the visibilities endlessly.

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 matching existing elements, operated properly by construction.

Circular Network Study

In the circular network study, we performed in total 29 iterations, which conforms to the 29 identified faults. This is due to the reason that 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 then 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 successfully executed. The details about the failures resolved by the fix for each fault are described in [Sag20].

Across these iterations, 118 failures occurred. In contrast to the first study, we also counted incorrectness of the transformations in this study, which are 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 the transformation within a network rather than using a transformation 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

Proper synchronization by construction

Iteration per fault

Incorrectness revealed by combination

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:

$$\text{classified failure ratio} = \text{resolved failure ratio} = 1$$

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

Further mistakes

Evaluation of metrics

Categorization completeness and correctness

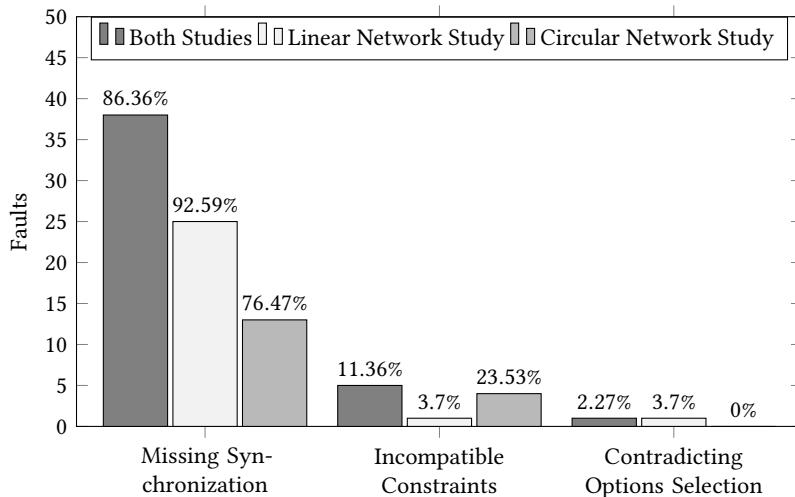


Figure 9.2.: Absolute numbers of faults due to different mistake types in both case studies. Percentages are relative to total number of faults in the particular case study.

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 in our case studies. We summarize the results of this analysis, depicting the evaluation of the according metrics, 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 still made up more than 75 % of the 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 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 unprecise 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 transformations. Since synchronization can be achieved by construction without knowing about the other transformations in a transformation 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 transformation 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 exists, 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.

Relevance
of orches-
tration
problem

Simple or-
chestration
strategy
sufficiency

9.2.4.2. Synchronization

Synchronization approach correctness

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 both 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 which failed due to missing synchronization, i.e., mistakes at the transformation level, and in which we were able to 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 study, as reflected by the success ratio metric:

$$\text{success ratio} = 1$$

Synchronization approach completeness

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 were 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 proposed 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:

$$\text{application ratio} = 1$$

Synchronizing transformation by construction

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.

Avoidance by construction

In conclusion, we found the proposed approach for constructing synchronizing transformations to be correct and complete in the considered case studies.

↓ Mistake Type	PCM ↓ UML ↓ Java	PCM ↓ UML ↓ Java	PCM ↓ UML ↓ Java	PCM ↓ UML ↓ Java
	Phase →			
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 9.11.: Number 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.

This serves as an indicator for its general correctness and completeness, and thus the possibility to use it is 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 9.11.

Mistake types per case study phase

In the first two phases, the consistency preservation rules of the transformations between UML and Java are added. Since these two phases were already

Incorrectness revealed by combination

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 the 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.

Cycles introducing synchronization errors

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 first introduce a cycle in the transformations, which leads to the scenario that transformation actually 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.

Transformation correctness not assumable

Thus, 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 when constructing a transformation network. 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 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 relation. 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, avoid them by construction.

In the circular network case 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.

Most faults
avoidable
by construc-
tion

Synchro-
nization
faults hide
others

Multiple in-
stantiation
by incom-
patible
relations

Relevance
of orches-
tration
problem

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 it is at least an indicator that it is not such a central problem that transformation networks are unlikely to ever work properly. Still, this has to be validated in further studies to improve external validity of the statement.

Results
indicate
statement

9.2.5.2. Threats to Validity

In the following, we discuss different possible threats to the validity of the discussed results. Due to the limited set of case studies, which especially limits external validity, 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 be actually restricted, distinguished by construct, internal, conclusion and external validity [Woh+12].

Pre-
alignment
of transfor-
mations

Construct Validity We assumed the transformations not to be aligned with each other a priori, because otherwise certain errors would not occur at all, thus reducing the number of faults and influencing their distribution. We have, however, mitigated this threat by developing each transformation in an isolated project without knowing that it is supposed to be combined with other transformations, as well as by giving the development task to different, independent 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 the developers may also exchange information, but at least not have an explicit representation of common knowledge.

Language
selection

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 from problems that may occur with a state-based approach to consistency preservation. We did, however, explicitly 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 from 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 synchronized before. 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 an appropriate case study in future work to further validate how well synchronization can be achieved by construction and how this influences the relevance of other mistakes, as we will discuss in Subsection 9.2.6.

Constructing vs.
fixing
synchronization

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. Like we have already discussed, 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.

Amount of data

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, but rather dependent on the existence of specific patterns, such as the possibility for transformations to select from multiple options to restore consistency, and potentially from

Number of case studies

the size of a transformation network. Especially the evidence of our results regarding 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, while the number of case studies is rather low, the number of considered scenarios within the case studies is not that low, thus representing a comprehensive set of scenarios.

Scenario selection

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.

Limited options in consistency relations

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, especially at the network rule level, and thus 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 anyway. 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 also revealed some limitations of our approaches and insights, which represents 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 especially differentiates in the key information used to search for, it could be embedded into an API or language construct to be reusable.

Manual implementation

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 provides the possibility to define keys for matching existing elements [Obj16a, p. 7.10.2.], which represents the essential idea of the proposed approach.

Language-integrated synchronization

Semantic Element Matching In the evaluation, we have detected cases which may be expected to be consequences of incompatibilities, but are actually not. For example, the transformation between UML and PCM creates a repository starting lowercase, whereas the transformation between Java and PCM generates a repository 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 intuitively would be assumed to be incompatible. Intuitively, it is expected that both repositories are supposed

Syntactic element matching

to represent them same element (cf. [Sağ20, Figure 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 global specification of consistency or a mapping to 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 ?? was the swap of first name and last name by one consistency relations, 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 ??, we will 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, representing 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 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 to align user decision across multiple transformations with each other.

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 consistent orchestration is difficult due to the 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.

Consistency
preserva-
tion rule
combi-
nation
correctness
notion

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.

Synchro-
nization
evalua-
tion
evidence

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)	Does the provenance algorithm improve the ability of identifying the reasons for a network not being able to find a consistent orchestration regarding an arbitrary strategy?
<i>Metric 5.1.1:</i>	<i>Considered transformations ratio: Ratio between the number of transformations to consider for finding a fault and the total number of transformations</i>

Table 9.12.: Goals, questions and metrics for orchestration evaluation.

9.3. Orchestration Algorithm

In Section 7.4, we have proposed the provenance algorithm, which is proven correct, i.e., which returns only consistent models and does always terminate. Thus, it conservatively approximates the orchestration problem. We have motivated the strategy 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 8) iteratively achieves consistency according to 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

Usefulness
evaluationRecapture
algorithmEvaluation
goal and re-
quirements

to be able to find a consistent orchestration together with an according evaluation question and metric in Table 9.12. 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 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.

Controlled experiment

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:

$$\text{considered transformations ratio} = \frac{\# \text{ of transformations to consider}}{\# \text{ of total transformations}}$$

Scenario-based discussion

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 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.

Two example scenarios

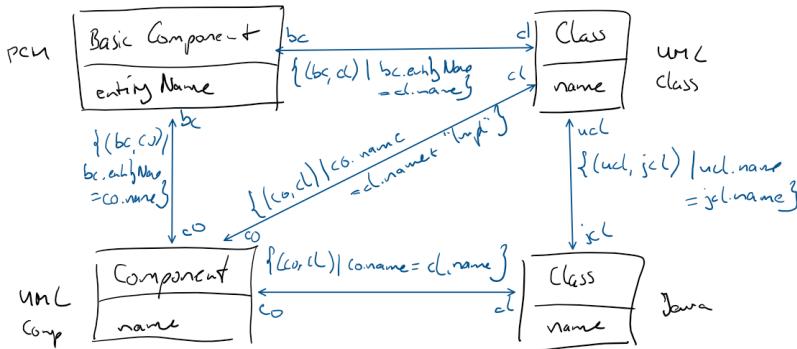


Figure 9.3.: Consistency relations between basic components in PCM model, components in UML component models, classes in UML class models and classes in Java models.

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 PCM and UML, as well as classes in 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

Scenario description

Reasons for failing

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 two categories.

Distinction
of failure
cases

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, 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 even 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 gets the information which addition of a transformation led to the failure and, in addition, the state of the models in which the algorithm aborted. There is at least one consistency relation that is violated, which led to the abortion of the algorithm, and this consistency relation must belong to one of the transformations within the cycle containing the fault. In consequence, the transformation developer must only 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 this 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:

$$\text{considered transformations ratio} = \frac{3}{5}$$

Even if the transformation between UML class and component models is the last to be executed, still only three and not all five transformations need to 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 consistency relations. This can, however, not be the case, as consistency to these transformations was already achieved before. In the example, if the cycle of transformations between UML component models, UML class models and Java code did not contain 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 as well.

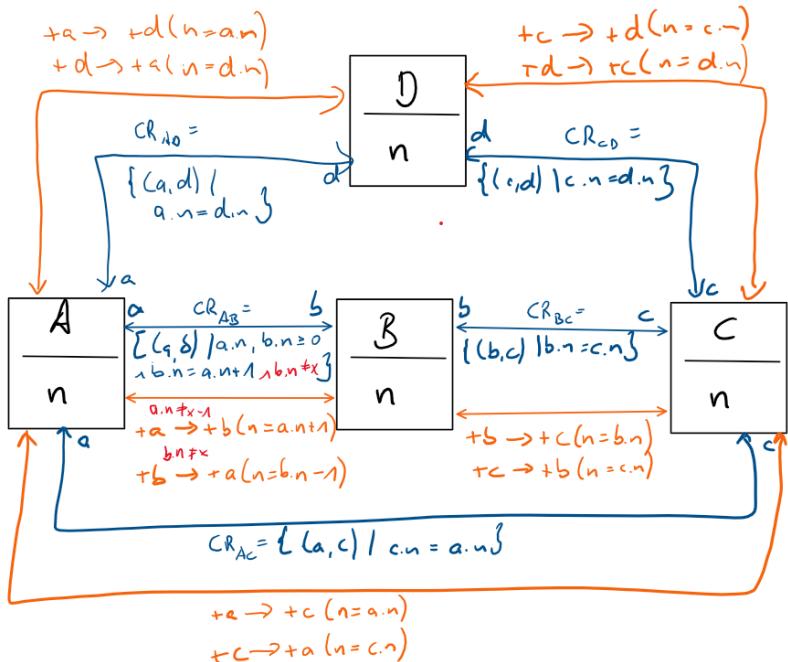


Figure 9.4.: Extension of the example in Figure 7.2 with consistency relations that require an arbitrary number of transformation execution, depending on value x .

Orchestration Problem

Figure 9.4 shows 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 cycles 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, in this case we have:

$$\text{considered transformations ratio} = \frac{4}{5}$$

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.

Reasons for failing

Number of considered transformations

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 yet 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. 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. Rather than choosing an arbitrary transformation to be executed next, first cycles should be closed, 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, we can only consider our finding as an initial indicator for the usefulness of the provenance algorithm in terms of improving the ability to identify reasons for it to fail as asked by Question 5.1. Still, we found a criterion in the scenarios that limits the number of transformations that need to be considered in case of a failure by construction of the algorithm. Thus, regarding the number of transformations that need to be considered to identify a fault, it guarantees an improvement regarding an arbitrary other strategy, which, in the worst case, can require the investigation of all transformations. Whether or not this metric reasonably reflects usefulness of

General restriction
considered transformations

Transformation selection order

Evidence of results

the approach does, however, remain one threat to validity until its validation in an experiment.

9.3.3.2. Threats to Validity

In the evaluation, we found a general 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.

Construct validity 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 also presented qualitative arguments and discussed further quantifiable improvements, such as the restriction of consistency relations to consider in the failure case.

External validity Finally, we 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 developed a simulator for evaluating different strategies, but unfortunately we found each strategy to be outperformed by at least one other strategy regarding their the ability to find a consistency 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

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 metrics we have investigated actually reflect usefulness of the strategy in terms of reducing the time and effort when identifying faults in transformation network. Thus, in future work, 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 further 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.

Usefulness
of design
property

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.

Systematic
execution
order

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

Fault identi-
fication
process

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+20b]), to which the results of this thesis contribute, and thus 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

Limitations
by assump-
tions

Extension
to multiary
relations

rely on a binary notion of consistency. While this is a limitation, since not every multiary consistency relation can be decomposed into binary ones (cf. [Ste20]), 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. Without explicitly considering that case, however, we currently need to accept it as a limitation, for which we have to validate in future work whether and for which statements it actually is a limitation. This also resolves the issue that our approaches can currently only be applied to relations that are denoted as *binary-definable* by Stevens [Ste20].

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 first discussed one and thus imply the same necessity for future work.

Considering
behavioral
consistency

Concurrent Editing Finally, we assumed that a user only changes one model, for which consistency 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, such as [Xio+13; Xio+09].

Difference
to synchro-
nization

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 a 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 levels 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 this 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 takeaways, it is particularly important to ensure that transformations are synchronizing by construction.

A. Appendix

A.1. Compatibility Proofs

Compatibility of consistency relation trees

Auxiliary lemma

In Section 5.3, we have given Theorem 5.6 for the inherent compatibility of consistency relation trees as defined in Definition 5.6. Due to the complexity of the according proof, we provide it in this appendix rather than the running text of that chapter.

To proof the statement of Theorem 5.6, we first present a lemma that shows that in a consistency relation tree you 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 in the relations between classes.

Lemma A.1 (Consistency Relation Tree Unique Paths)

Let $\mathbb{CR} = \{CR_1, CR^T_1, \dots, CR_k, CR^T_k\}$ be a symmetric, connected set of consistency relations. \mathbb{CR} is a consistency relation tree if, and only if, for each $CR \in \mathbb{CR}$ there exists a sequence of consistency relations $\mathbb{CR}'[] = [CR'_1, \dots, CR'_k]$ with $CR'_1 = CR$, containing for each i either CR_i or CR^T_i , i.e.,

$$\begin{aligned} \forall i \in \{1, \dots, k\} : & (CR_i \in \mathbb{CR}'[] \wedge CR^T_i \notin \mathbb{CR}'[]) \\ & \vee (CR^T_i \in \mathbb{CR}'[] \wedge CR_i \notin \mathbb{CR}'[]) \end{aligned}$$

such that:

$$\begin{aligned} \forall s \in \{1, \dots, k-1\} : \forall t \in \{i+1, \dots, k\} : \\ \mathfrak{C}_{r, CR'_s} \cap \mathfrak{C}_{r, CR'_t} = \emptyset \wedge \mathfrak{C}_{l, CR'_s} \cap \mathfrak{C}_{r, CR'_t} = \emptyset \end{aligned}$$

Proof. We start with the forward direction, i.e., given a consistency relation tree \mathbb{CR} we show that there exists a sequence according to the requirements in Lemma A.1 by constructing such a sequence $\mathbb{CR}'[] = [CR'_1, \dots, CR'_k]$ for any $CR \in \mathbb{CR}$. Start with $CR'_1 = CR$ for any $CR \in \mathbb{CR}$. We now inductively add further relations to that sequence. Take any consistency relation $CR_s = CR_{s,1} \otimes \dots \otimes CR_{s,m} \in \mathbb{CR}^+$ with $\mathfrak{C}_{l, CR_{s,1}} \subseteq \mathfrak{C}_{r, CR}$. Such a sequence must exist because of \mathbb{CR} being connected. Now add all $CR_{s,1}, \dots, CR_{s,m}$ to the sequence, such that we have $[CR, CR_{s,1}, \dots, CR_{s,m}]$, which fulfills both requirements to that sequence in Lemma A.1 by definition. The following

addition of further consistency relations can be inductively applied. Take any other consistency relation $CR_t = CR_{t,1} \otimes \dots \otimes CR_{t,n} \in \mathbb{CR}^+$ such that:

$$\begin{aligned} \exists CR' \in \{CR, CR_{s,1}, \dots, CR_{s,m}\} : \mathfrak{C}_{l,CR_{t,1}} &\subseteq \mathfrak{C}_{r,CR'} \\ \wedge CR_{t,1}, CR'^T_{t,1} &\notin \{CR, CR_{s,1}, \dots, CR_{s,m}\} \end{aligned}$$

In other words, 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, add all $CR_{t,1}, \dots, CR_{t,n}$ to the sequence. Per construction, for each CR' in the sequence, there is a non-empty concatenation of relations within the sequence $CR \otimes \dots \otimes CR'$, because relations were added in a way 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 $CR'^T_{t,i}$ is already contained in the sequence $[CR, CR_{s,1}, \dots, CR_{s,m}]$. Second (2. ,3.), we show that both further conditions for the sequence defined in Lemma A.1 are still fulfilled for the sequence $[CR, CR_{s,1}, \dots, CR_{s,m}, CR_{t,1}, \dots, CR_{t,n}]$.

1. Let us assume that the sequence $[CR, CR_{s,1}, \dots, CR_{s,m}]$ already contained one of the $CR_{t,i}$ or $CR'^T_{t,i}$. If $CR_{t,i}$ is contained in the sequence, there is a concatenation $CR \otimes \dots \otimes CR_{t,i}$ with relations in $[CR, CR_{s,1}, \dots, CR_{s,m}]$, as well as a concatenation $CR \otimes \dots \otimes CR_{t,1} \otimes \dots \otimes CR_{t,i}$. Since $CR_{t,1} \notin \{CR, CR_{s,1}, \dots, CR_{s,m}\}$ by construction, these two concatenations relate the same class tuples, i.e., they contradict the definition of a consistency relation tree. If $CR'^T_{t,i}$ was contained in the sequence $[CR, CR_{s,1} \otimes \dots \otimes CR_{s,m}]$, there is a concatenation $CR \otimes \dots \otimes CR_w \otimes CR'^T_{t,i}$ with relations in $[CR, CR_{s,1}, \dots, CR_{s,m}]$ and, like before, the concatenation $CR \otimes \dots \otimes CR_{t,1}, \dots, CR_{t,i}$. Due to $\mathfrak{C}_{r,CR_w} \cap \mathfrak{C}_{l,CR_{t,i}} \neq \emptyset$ and $CR'^T_{t,1} \notin \{CR, CR_{s,1}, \dots, CR_{s,m}\}$ by construction, the two concatenations $CR \otimes \dots \otimes CR_w$ and $CR \otimes \dots \otimes CR_{t,1} \otimes \dots \otimes CR_{t,i}$ have an overlap in both their left and right class tuples, i.e., they contradict the definition of a consistency relation tree. In consequence, the sequence $[CR, CR_{s,1}, \dots, CR_{s,m}]$ cannot have contained any $CR_{t,i}$ or $CR'^T_{t,i}$ before.
2. Let us assume there were any CR'_u and CR'_v in the sequence $[CR, CR_{s,1}, \dots, CR_{s,m}]$ such that $\mathfrak{C}_{r,CR'_u} \cap \mathfrak{C}_{r,CR'_v} \neq \emptyset$. As discussed before, for each of

these relations exists a concatenation of relations in the sequence $CR \otimes \dots \otimes CR'_u$ and $CR \otimes \dots \otimes CR'_v$, which is contained in \mathbb{CR}^+ . This contradicts the definition of a consistency relation tree, so there cannot be two such relations with overlapping classes in the right class tuple.

3. Let us assume there were any CR'_u and CR'_v ($u < v$) in the sequence $[CR, CR_{s,1}, \dots, CR_{s,m}, CR_{t,1}, \dots, CR_{t,n}]$ such that $\mathfrak{C}_{l,CR'_u} \cap \mathfrak{C}_{r,CR'_v} \neq \emptyset$. Again per construction, there must be a non-empty concatenation $CR \otimes \dots \otimes CR'_w \otimes CR'_u$ with $w < u$. Since $\mathfrak{C}_{l,CR'_u} \subseteq \mathfrak{C}_{r,CR'_w}$ per definition, it holds that $\mathfrak{C}_{r,CR'_w} \cap \mathfrak{C}_{r,CR'_v} \neq \emptyset$. In other words, the relation CR'_v introduces a cycle in the relations. 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 already 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 that a sequence according to the requirements in Lemma A.1 exists 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 $CR_s = CR_{s,1} \otimes \dots \otimes CR_{s,m} \in \mathbb{CR}^+$ and $CR_t = CR_{t,1} \otimes \dots \otimes CR_{t,n} \in \mathbb{CR}^+$ 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 otherwise we could instead consider the sequence 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, due to 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:

$$\begin{aligned} \mathfrak{C}_{l,CRT_{s,i-1}} \cap \mathfrak{C}_{r,CRT_{s,i}} &\neq \emptyset \wedge \mathfrak{C}_{l,CRT_{s,1}} \cap \mathfrak{C}_{r,CRT_{s,1}} \neq \emptyset \\ \wedge \mathfrak{C}_{l,CRT_{s,i}} \cap \mathfrak{C}_{r,CRT_{s,i-1}} &\neq \emptyset \wedge \mathfrak{C}_{l,CRT_{s,m}} \cap \mathfrak{C}_{r,CRT_{s,n}} \neq \emptyset \end{aligned}$$

Because of that cycle in the overlap of class tuples, there is no order of these relations $CR''_1, \dots, CR''_{m+n}$ such that for all of them it holds that $\mathfrak{C}_{l,CR''_u} \cap \mathfrak{C}_{r,CR''_v} \neq \emptyset$ ($u < v$), which contradicts the assumptions regarding the sequence in Lemma A.1. The analog 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. \square

The previous lemma shows that the definition of consistency relation trees based on unique concatenations of the same class tuples is equivalent the possibility to find sequences of the relations that do not contain cycles in the related class tuples. The definition is supposed to be easier to check in practice. However, we can now show that a consistency relation tree is always compatible with 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 that contains the condition element and is consistent, i.e., that 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 5.13, based on the relations in the consistency relation tree in Figure 5.12.

Base case: Take any $CR \in \mathbb{CR}$ and any of its left side condition elements $c_l = \langle o_{l,1}, \dots, o_{l,m} \rangle \in \mathfrak{C}_{l,CR}$. Select any $c_r = \langle o_{r,1}, \dots, o_{r,n} \rangle \in \mathfrak{C}_{r,CR}$, such that c_l and c_r constitute a consistency relation pair $\langle c_l, c_r \rangle \in CR$. Now construct

the model tuple \mathbf{m} that contains only $o_{l,1}, \dots, o_{l,m}$ and $o_{r,1}, \dots, o_{r,n}$. In consequence, we have a minimal model tuple \mathbf{m} , such that \mathbf{m} contains c_l and \mathbf{m} consistent to CR . Additionally, \mathbf{m} is consistent to CR^T due to symmetry of CR and CR^T : It is $c_r \in \mathbb{C}_{l,CR^T}$ and $\langle c_r, c_l \rangle \in CR^T$ and no other condition element of \mathbb{C}_{l,CR^T} is contained in \mathbf{m} by construction, thus \mathbf{m} is consistent to CR^T . In consequence, we know that for all $CR \in \mathbb{CR}$, $\{CR, CR^T\}$ is compatible. Considering the example in *Figure 5.13*, 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: According to Lemma A.1, there is a sequence $[CR_1, \dots, CR_k]$ of the relations in \mathbb{CR} with $CR_1 = CR$, such that:

$$\forall s \in \{1, \dots, k-1\} : \forall t \in \{i+1, \dots, k\} : \\ \mathbb{C}_{r,CR'_s} \cap \mathbb{C}_{r,CR'_t} = \emptyset \wedge \mathbb{C}_{l,CR'_s} \cap \mathbb{C}_{r,CR'_t} = \emptyset$$

Considering the example in Figure 5.13, 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 . If, in general, we know that $\{CR_1, CR^T_1, \dots, CR_i, CR^T_i\}$ ($i < k$) is compatible, for every $c_l \in \mathbb{C}_{l,CR}$, we can find a model tuple \mathbf{m} that contains c_l and is consistent to $\{CR_1, CR^T_1, \dots, CR_i, CR^T_i\}$ by definition. We can especially create a minimal model according to our construction for the base case and the following inductive completion.

Induction step: Consider CR_{i+1} . There is at most one condition element $c_l \in \mathbb{C}_{l,CR_{i+1}}$ with \mathbf{m} contains c_l . If there were at least two condition elements $c_l, c'_l \in \mathbb{C}_{l,CR_{i+1}}$, both contained in \mathbf{m} , then by construction there is a consistency relation CR_s ($s < i+1$) with $c_l, c'_l \in \mathbb{C}_{r,CR_j}$. 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 $CR \otimes \dots \otimes CR_s$ and $CR' \otimes \dots \otimes CR_t$ with $\mathbb{C}_{l,CR} \cap \mathbb{C}_{l,CR'} \neq \emptyset$, because we started the construction with elements from the left condition of CR , so every element is contained because of a relation to those elements, and with $\mathbb{C}_{r,CR_s} \cap \mathbb{C}_{r,CR_t} \neq \emptyset$, because both condition elements c_l and 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 $c_{II}, c'_{II} \in \mathbb{C}_{I,CR_s}$ with $\langle c_{II}, c_I \rangle, \langle c'_{II}, c'_I \rangle \in CR_s$, because per construction m was consistent to CR_s and so there must be a witness structure with a unique mapping between condition elements contained in m . The above argument can be applied inductively until we finally find that there must be two condition elements $c_{III}, c'_{III} \in \mathbb{C}_{I,CR}$, which are contained in m . This is not true by construction, as we started with only one element from $\mathbb{C}_{I,CR}$, so there is only one such condition element $c_I \in \mathbb{C}_{I,CR_{i+1}}$ with m contains c_I .

For this condition element $c_I \in \mathbb{C}_{I,CR_{i+1}}$, select an arbitrary $c_r = \langle o_1, \dots, o_s \rangle \in \mathbb{C}_{r,CR_{i+1}}$, such that $\langle c_I, c_r \rangle \in CR_{i+1}$. Now create a model tuple m' by adding the objects o_1, \dots, o_s to m . Since c_I is the only of the left condition elements of CR_{i+1} that m contains, model tuple m' is consistent to CR_{i+1} per construction. m' is also consistent to CR^T_{i+1} , because due to the symmetry of CR_{i+1} and CR^T_{i+1} , it is $c_r \in \mathbb{C}_{I,CR^T_{i+1}}$ and due to $\langle c_r, c_I \rangle \in CR^T_{i+1}$, a consistent corresponding element exists in m' . Furthermore, there cannot be any other $c' \in \mathbb{C}_{I,CR^T_{i+1}}$ with m' contains c' , because otherwise there would have been another consistency relation CR' that required the creation of c' , which means that there are two concatenations of consistency relations $CR \otimes \dots \otimes CR'$ and $CR \otimes \dots \otimes CR_{i+1}$ that both relate instances of the same classes, which contradicts Definition 5.6 for a consistency relation tree.

Additionally, due to Lemma A.1, for all CR_s ($s < i + 1$), we know that $\mathbb{C}_{I,CR_s} \cap \mathbb{C}_{r,CR_{i+1}} = \emptyset$. Since the newly added elements c_r are part of $\mathbb{C}_{r,CR_{i+1}}$, these elements cannot match the left conditions of any of the consistency relations CR_s ($s < i + 1$). So m' is still consistent to all CR_s ($s < i + 1$). Finally, due to Lemma A.1, for all CR_s ($s < i + 1$), we know that $\mathbb{C}_{r,CR_s} \cap \mathbb{C}_{r,CR_{i+1}} = \emptyset$. Again, since the newly added elements c_r are part of $\mathbb{C}_{r,CR_{i+1}}$, these elements cannot match the left conditions of any of the consistency relations CR^T_s ($s < i + 1$). So m' is still consistent to all CR^T_s ($s < i + 1$). In consequence, we know that m' consistent to $\{CR_1, CR^T_1, \dots, CR_{i+1}, CR^T_{i+1}\}$.

Considering the example in Figure 5.13, 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

$$m' \text{ consistent to } \{CR_1, CR^T_1, \dots, CR_k, CR^T_k\} = \mathbb{CR}$$

Since the construction is valid for each condition element in every consistency relation in \mathbb{CR} , we know that a consistency relation tree \mathbb{CR} is compatible.

□

Bibliography

- [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: *2017 7th International Conference on Computer and Knowledge Engineering (ICCKE)*. 2017, pp. 154–160.
- [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.
- [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.
- [BKS02] B. Beckert, U. Keller, and P. H. Schmitt. “Translating the Object Constraint Language into First-order Predicate Logic”. In: *VERIFY Workshop (VERIFY 2002) at FLoC 2002: Federated Logic Conferences*. 2002, pp. 113–123.
- [BCG05] D. Berardi, D. Calvanese, and G. D. Giacomo. “Reasoning on UML class diagrams”. In: *Artificial Intelligence* 168.1 (2005), pp. 70–118.
- [Bra+98] A. Brandstädt et al. “Dually Chordal Graphs”. In: *SIAM J. Discret. Math.* 11.3 (1998), pp. 437–455.
- [Bro87] F. P. Brooks. “No Silver Bullet Essence and Accidents of Software Engineering”. In: *Computer* 20.4 (1987), pp. 10–19.
- [Cab+10] J. Cabot et al. “Verification and Validation of Declarative Model-to-Model Transformations through Invariants”. In: *Journal of Systems and Software* 83.2 (2010), pp. 283–302.

- [Che+17] J. Cheney et al. “On principles of Least Change and Least Surprise for bidirectional transformations”. In: *Journal of Object Technology* 16.1 (2017), 3:1–31.
- [Cle+19] A. Cleve et al. “Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491)”. In: *Dagstuhl Reports* 8.12 (2019), pp. 1–48.
- [Cua19] J. S. Cuadrado. “A verified catalogue of OCL optimisations”. In: *Software & Systems Modeling* (2019).
- [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.
- [CH03] K. Czarnecki and S. Helsen. “Classification of Model Transformation Approaches”. In: *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*. 2003.
- [Dis+11] Z. Diskin et al. “From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case”. In: *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODELS 2011)*. Vol. 6881. LNCS. Springer-Verlag, 2011, pp. 304–318.
- [Ecl20a] Eclipse Foundation, Inc. *Eclipse OCL (Object Constraint Language)*. 2020. URL: <https://projects.eclipse.org/projects/modeling.mdt.ocl> (visited on 08/24/2020).
- [Ecl20b] Eclipse Foundation, Inc. *Eclipse QVTd (QVT Declarative)*. 2020. URL: <https://projects.eclipse.org/projects/modeling.mmt.qvtd> (visited on 08/24/2020).
- [Ecl20c] Eclipse Foundation, Inc. *Model to Model Transformation (MMT)*. 2020. URL: <https://www.eclipse.org/mmt/> (visited on 08/24/2020).
- [Est+07] J. A. Estefan et al. “Survey of model-based systems engineering (MBSE) methodologies”. In: *Incose MBSE Focus Group* 25.8 (2007), pp. 1–12.
- [ETA] ETAS Group. *ASCET-DEVELOPER*. URL: <https://www.etas.com/ascet> (visited on 02/12/2020).
- [FM08] S. Fraser and D. Mancl. “No Silver Bullet: Software Engineering Reloaded”. In: *IEEE Software* 25.1 (2008), pp. 91–94.

- [Gib69] N. E. Gibbs. “A cycle generation algorithm for finite undirected linear graphs”. In: *Journal of the ACM (JACM)* 16.4 (1969), pp. 564–568.
- [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*. Vol. 5765. LNCS. Springer Berlin / Heidelberg, 2010, pp. 555–579.
- [Gle17] J. Gleitze. “A Declarative Language for Preserving Consistency of Multiple Models”. Bachelor’s Thesis. Karlsruhe Institute of Technology (KIT), 2017.
- [Gle20] J. Gleitze. *Transformation Network Simulator*. not archived and finalized yet. 2020. URL: <https://github.com/jGleitz/transformationnetworksimulator>.
- [GKB20] J. Gleitze, H. Klare, and E. Burger. *Finding a Universal Execution Strategy for Model Transformation Networks*. not published yet. 2020.
- [Gui+18] H. Guissouma et al. “An Empirical Study on the Current and Future Challenges of Automotive Software Release and Configuration Management”. In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2018, pp. 298–305.
- [Her+12] F. Hermann et al. “Concurrent Model Synchronization with Conflict Resolution Based on Triple Graph Grammars”. In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering (FASE 2012)*. Vol. 7212. LNCS. Springer-Verlag, 2012, pp. 178–193.
- [ITE] ITEA. *AMALTHEA4public – An Open Platform Project for Embedded Multicore Systems*. URL: <http://www.amalthea-project.org/> (visited on 02/12/2020).
- [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 (MODELS): Companion Proceedings*. 2018, pp. 156–161.

- [Kla+19] H. Klare et al. “A Categorization of Interoperability Issues in Networks of Transformations”. In: *Journal of Object Technology* 18.3 (2019). The 12th International Conference on Model Transformations, 4:1–20.
- [Kla+20a] H. Klare et al. *A Formal Approach to Prove Compatibility in Transformation Networks*. Tech. rep. 3. Karlsruher Institut für Technologie (KIT), 2020. 40 pp.
- [Kla+20b] H. Klare et al. “Enabling Consistency in View-based System Development – The Vitruvius Approach”. In: (2020).
- [KS06] A. Königs and A. Schürr. “MDI: A Rule-based Multi-document and Tool Integration Approach”. In: *Software and Systems Modeling (SoSyM)* 5.4 (2006), pp. 349–368.
- [Kra+16] M. E. Kramer et al. “A Controlled Experiment Template for Evaluating the Understandability of Model Transformation Languages”. In: *Proceedings of the Second International Workshop on Human Factors in Modeling*. Vol. 1805. 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. 278 pp.
- [Lan17] M. Langhammer. “Automated Coevolution of Source Code and Software Architecture Models”. PhD thesis. Karlsruhe Institute of Technology (KIT), 2017. 259 pp.
- [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*. Vol. 4. 2015.
- [Lap13] P. A. Laplante. *Requirements Engineering for Software and Systems*. 2nd. Auerbach Publications, 2013.
- [Leb+14] E. Leblebici et al. “A Comparison of Incremental Triple Graph Grammar Tools”. In: *Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014)*. Vol. 67. Electronic Communications of the EASST. EASST, 2014.

- [MC16] N. Macedo and A. Cunha. “Least-change bidirectional model transformation with QVT-R and ATL”. In: *Software & Systems Modeling* 15.3 (2016), pp. 783–810.
- [MCP14] N. Macedo, A. Cunha, and H. Pacheco. “Towards a framework for multi-directional model transformations”. In: *3rd International Workshop on Bidirectional Transformations - BX*. Vol. 1133. CEUR-WS.org, 2014.
- [Mat] MathWorks. *Simulink – Simulation and Model-Based Design – MATLAB & Simulink*. URL: <https://www.mathworks.com/products/simulink.html> (visited on 02/12/2020).
- [Maz+17] M. Mazkatli et al. “Automotive Systems Modelling with Vitruvius”. In: *15. Workshop Automotive Software Engineering*. Vol. P-275. Lecture Notes in Informatics (LNI). GI, Bonn, 2017, pp. 1487–1498.
- [MB08] L. de Moura and N. Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008, pp. 337–340.
- [MBF11] S. Murer, B. Bonati, and F. J. Furrer. *Managed Evolution – A Strategy for Very Large Information Systems*. Springer Berlin Heidelberg, 2011, p. 264.
- [Obj16a] Object Management Group (OMG). *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Version 1.3. 2016.
- [Obj14a] Object Management Group (OMG). *Model Driven Architecture (MDA) – MDA Guide rev. 2.0*. 2014.
- [Obj14b] Object Management Group (OMG). *Object Constraint Language*. Version 2.4. 2014.
- [Obj16b] Object Management Group (OMG). *OMG Meta Object Facility (MOF) Core Specification*. Version 2.5.1. 2016.
- [Obj19] Object Management Group (OMG). *OMG System Modeling Language (OMG SysML)*. Version 1.6. 2019.
- [Obj17] Object Management Group (OMG). *OMG Unified Modeling Language (OMG UML)*. Version 2.5.1. 2017.
- [OPN20] F. Orejas, E. Pino, and M. Navarro. “Incremental Concurrent Model Synchronization using Triple Graph Grammars”. In: *Fundamental Approaches to Software Engineering*. 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.
- [Pep] A. Pepin. *Decomposition GitHub Repository*. Move to archive. URL: https://github.com/aurelienpepin/KIT_ConsistencyPreservation-Decomposition (visited on 08/27/2020).
- [Pep19] A. Pepin. “Decomposition of Relations for Multi-model Consistency Preservation”. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2019.
- [PRV08] M. Petrenko, V. Rajlich, and R. Vanciu. “Partial Domain Comprehension in Software Evolution and Maintenance”. In: *2008 16th IEEE International Conference on Program Comprehension*. 2008, pp. 13–22.
- [Pil+08] J. von Pilgrim et al. “Constructing and Visualizing Transformation Chains”. In: *Model Driven Architecture – Foundations and Applications*. Springer Berlin Heidelberg, 2008, pp. 17–32.
- [Reu+16] R. H. Reussner et al. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. 408 pp.
- [Sağ20] T. Sağlam. “A Case Study for Networks of Bidirectional Transformations”. MA thesis. Karlsruher Institut für Technologie (KIT), 2020. 81 pp.
- [Sch15] O. Scheid. *AUTOSAR Compendium - Part 1: Application and RTE*. AUTOSAR - Compendium Series. CreateSpace Independent Publishing Platform, 2015.
- [SV06] T. Stahl and M. Völter. *Model-Driven Software Development*. John Wiley & Sons, 2006.
- [Ste+09] D. Steinberg et al. *EMF - Eclipse Modeling Framework*. 2nd ed. Addison-Wesley Professional, 2009.
- [Ste10] P. Stevens. “Bidirectional model transformations in QVT: semantic issues and open questions”. In: *Software & Systems Modeling* 9.1 (2010), pp. 7–20.
- [Ste20] P. Stevens. “Maintaining consistency in networks of models: bidirectional transformations in the large”. In: *Software and Systems Modeling* 19.1 (2020), pp. 39–65.

- [SK16] M. Strittmatter and A. Kechaou. *The Media Store 3 Case Study System*. Tech. rep. 2016,1. Faculty of Informatics, Karlsruhe Institute of Technology, 2016.
- [Sym18] T. Syma. “Multi-model Consistency through Transitive Combination of Binary Transformations”. Master’s Thesis. Karlsruhe Institute of Technology (KIT), 2018.
- [Tar72] R. Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.
- [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.
- [TA16] F. Trollmann and S. Albayrak. “Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models”. In: *Theory and Practice of Model Transformations*. Springer International Publishing, 2016, pp. 91–106.
- [TA15] F. Trollmann and S. Albayrak. “Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models”. In: *Theory and Practice of Model Transformations*. Springer International Publishing, 2015, pp. 214–229.
- [Val+12] A. Vallecillo et al. “Formal Specification and Testing of Model Transformations”. In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18–23, 2012. Advanced Lectures*. Springer Berlin Heidelberg, 2012, pp. 399–437.
- [Van+07] B. Vanhooff et al. “UniTI: A Unified Transformation Infrastructure”. In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2007, pp. 31–45.
- [Vea+12] M. Veanes et al. “Symbolic Finite State Transducers: Algorithms and Applications”. In: *SIGPLAN Not.* 47.1 (2012), pp. 137–150.
- [Vita] Vitruv Tools. *VITRUVIUS Component-based Systems Case Study (GitHub)*. Move to archive. URL: <https://github.com/vitruv-tools/Vitruv-Applications-ComponentBasedSystems> (visited on 08/27/2020).

- [Vitb] Vitruv Tools. *VITRUVIUS Framework (GitHub)*. Move to archive. URL: <https://github.com/vitruv-tools/Vitruv> (visited on 08/27/2020).
- [Völ+13] M. Völter et al. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.
- [WVD10] D. Wagelaar, R. Van Der Straeten, and D. Deridder. “Module superimposition: a composition technique for rule-based model transformation languages”. In: *Software & Systems Modeling* 9.3 (2010), pp. 285–309.
- [Wag+11] D. Wagelaar et al. “Towards a General Composition Semantics for Rule-Based Model Transformation”. In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2011, pp. 623–637.
- [Woh+12] C. Wohlin et al. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.
- [Wol+15] C. Wolff et al. “AMALTHEA – Tailoring tools to projects in automotive software development”. In: *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. Vol. 2. 2015, pp. 515–520.
- [Xio+09] Y. Xiong et al. “Supporting Parallel Updates with Bidirectional Model Transformations”. In: *Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2009, pp. 213–228.
- [Xio+13] Y. Xiong et al. “Synchronizing concurrent model updates based on bidirectional transformation”. In: *Software & Systems Modeling* 12.1 (2013), pp. 89–104.