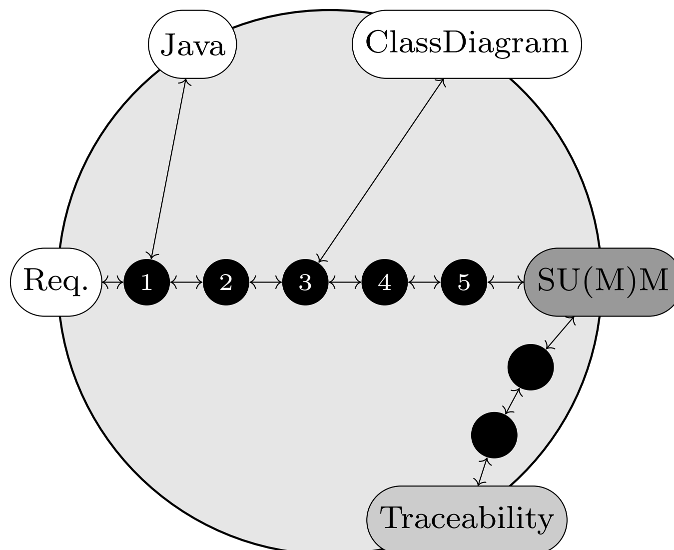
Carl von Ossietzky
Universität
Oldenburg

Ensuring Inter-Model Consistency



Von der Fakultät für Informatik, Wirtschafts- und Rechtswissenschaften
der Carl von Ossietzky Universität Oldenburg zur Erlangung des Grades und Titels

Doktors der Naturwissenschaften (Dr. rer. nat.)

angenommene Dissertation von Herrn

Johannes Meier

geboren am 24. November 1991 in Hannover.

Gutachter	Prof. Dr. Andreas Winter, Universität Oldenburg
Weitere Gutachter	Prof. Dr. Colin Atkinson, Universität Mannheim

Tag der Einreichung	31. Dezember 2021
Tag der Disputation	6. November 2023

Abstract

Increasing size, complexity and heterogeneity of software-intensive systems make it nearly impossible that single persons develop a whole system. Therefore, different stakeholders with different concerns are involved and are supported with tailored views on the system. These views conform to viewpoints and enable multi-view modeling of the system under development. Since these views are realized with models and jointly represent the whole system under development, the models semantically depend on each other in terms of redundant information, explicit links and further constraints, and therefore must be consistent to each other regarding these dependencies. Since the manual ensuring of consistency between models is error-prone, time-consuming and restricted by limited knowledge of users about models of other views, this thesis aims to develop an approach to automatically ensure consistency between multiple models.

MoConseMI (MOdel CONSistency Ensured by Metamodel Integration) is the newly designed and implemented approach for automatically ensuring inter-model consistency. Its main and unique characteristic is the reuse of existing metamodels and conforming models as data sources, which are integrated into an explicit Single Underlying (Meta)Model (SU(M)M). This enables to propagate changes between data sources and the SUM in order to re-establish the consistency after changes in any of the models. By this means, MoConseMI supports users of views with automated fixes for inconsistencies, while the desired project-specific consistency goals are configured only once for each project with reusable operators. MoConseMI does not require a formalization of the desired consistency, but provides a pragmatic strategy to initially create a SU(M)M in bottom-up way from existing (meta)models, which is automatically realized by operators that are manually configured to realize consistency. With this strategy, MoConseMI fills a gap in related work. In MoConseMI, existing data sources are complemented with new view(point)s which can be derived from the SU(M)M and are kept consistent directly with the SUM as well. The technical heterogeneity of models is overcome by reusable adapters for different technical spaces.

MoConseMI is successfully applied to several application examples. This emphasizes that MoConseMI is reusable for and transferable to a broad range of projects, allowing for even more than the presented applications. Additionally, the evaluation of MoConseMI shows, that the designed operators are reusable and reduce the configuration effort, that MoConseMI is combinable with other research and into other applications, and that MoConseMI can even fulfill intra-model consistency. MoConseMI complements its main contribution for ensuring inter-model consistency with further contributions for traceability, model co-evolution and difference representations for models and their metamodels.

Kurzfassung

Die zunehmende Größe, Komplexität und Heterogenität von Software-intensiven Systemen macht es nahezu unmöglich, dass Einzelpersonen ein System vollständig entwickeln. Deshalb werden verschiedene Akteure mit unterschiedlichen Belangen einbezogen und mit passgenauen Sichten auf das System unterstützt. Diese Sichten sind konform zu Sichtbeschreibungen und ermöglichen die sichtenbasierte Modellierung des sich in Entwicklung befindenden Systems. Da diese Sichten durch Modelle realisiert werden und gemeinsam das gesamte sich in Entwicklung befindende System repräsentieren, hängen die Modelle hinsichtlich redundanter Informationen, expliziter Verknüpfungen und weiterer Vorgaben semantisch voneinander ab und müssen deshalb entsprechend dieser Abhängigkeiten konsistent zueinander gehalten werden. Da die händische Sicherstellung von Konsistenz zwischen Modellen fehleranfällig, zeitaufwendig und durch beschränktes Wissen von Anwendern über Modelle anderer Sichten beeinträchtigt ist, entwickelt diese Arbeit einen Ansatz zur automatischen Sicherstellung von Konsistenz zwischen mehreren Modellen.

MoConseMI (MOdel CONSistency Ensured by Metamodel Integration: Modellkonsistenz sichergestellt durch Metamodellintegration) ist der neu entworfene und implementierte Ansatz zur automatischen Sicherstellung von Konsistenz zwischen Modellen. Dessen wichtigste und einzigartige Eigenschaft ist die Wiederverwendung bestehender Metamodelle und konformer Modelle als Datenquellen, die in ein explizites Single Underlying (Meta)Modell (SU(M)M) integriert werden. Dies ermöglicht es, Änderungen zwischen Datenquellen und dem SUM auszutauschen, um die Konsistenz nach Änderungen in einem der Modelle wiederherzustellen. Auf diese Weise unterstützt MoConseMI Anwender von Sichten mit automatischen Korrekturen von Inkonsistenzen, während die gewünschten projektspezifischen Konsistenzziele nur einmal für jedes Projekt mit wiederverwendbaren Operatoren konfiguriert werden. MoConseMI erfordert keine Formalisierung der gewünschten Konsistenz, sondern stellt eine pragmatische Strategie bereit, um ein SU(M)M durch die Wiederverwendung bestehender (Meta)Modelle initial zusammenzustellen, was durch Operatoren automatisiert wird, die manuell für die Realisierung der Konsistenz konfiguriert wurden. Mit dieser Strategie füllt MoConseMI eine Lücke in der Forschungslandschaft. In MoConseMI werden existierende Datenquellen durch neue Sichten gemäß neuer Sichtbeschreibungen ergänzt, die vom SU(M)M abgeleitet werden und ebenfalls mit dem SUM konsistent gehalten werden. Die technische Heterogenität von Modellen wird durch wiederverwendbare Adapter für unterschiedliche technische Lösungsräume überwunden.

MoConseMI wird erfolgreich für mehrere Anwendungsbeispiele angewendet. Dies zeigt, dass MoConseMI wiederverwendbar für und übertragbar auf ein breites Spektrum von Projekten ist, sodass über die gezeigten Anwendungen hinaus weitere möglich sind. Darüber hinaus zeigt die Evaluierung von MoConseMI, dass die

entwickelten Operatoren wiederverwendbar sind und den Konfigurationsaufwand reduzieren, dass MoConseMI kombinierbar mit anderer Forschung und in andere Anwendungen ist und dass MoConseMI Konsistenz sogar innerhalb von Modellen sicherstellen kann. MoConseMI ergänzt seinen Hauptbeitrag für die Sicherstellung von Konsistenz zwischen Modellen um weitere Beiträge für Nachverfolgbarkeit, Co-Evolution von Modellen und Darstellungen von Änderungen in Modellen und deren Metamodellen.

Contents at a Glance

Contents at a Glance 7

Contents in Detail 11

Typesetting Conventions 19

I Introduction 23

1 Motivation 25
- 1.1 Multi-perspective modeling . . . 26
- 1.2 Challenges . . . 31
- 1.3 Aims . . . 41
- 1.4 Summary & Outline . . . 47

II Foundations 49

2 Basic Concepts 51
- 2.1 Views and Viewpoints . . . 54
- 2.2 Modeling . . . 58
- 2.3 Consistency . . . 71
- 2.4 Stakeholders . . . 79
- 2.5 Technical Spaces . . . 84
- 2.6 Summary . . . 89

3 Related Work 93
- 3.1 Criteria for Classification . . . 94
- 3.2 Overall Realization Techniques . . . 99
- 3.3 Synthetic Approaches . . . 108
- 3.4 Single Underlying Model (SUM) . . . 120
- 3.5 Projectional Approaches . . . 121
- 3.6 Further Research Areas . . . 135
- 3.7 Summary: Lessons Learned . . . 146

4 Requirements 153
- 4.1 Functional Requirements . . . 154
- 4.2 Technical Requirements . . . 157
- 4.3 Summary . . . 158

III Approach 161

5 MoConseMI at a glance 163
5.1 Design Decisions . . . 163
5.2 Overview of the Approach . . . 171
5.3 Summary: MoConseMI . . . 179

6 Design 185
6.1 Operators as Transformations . . . 185
6.2 Metamodel Decisions . . . 192
6.3 Model Decisions . . . 198
6.4 Operator Combination . . . 203
6.5 Operator Execution . . . 213
6.6 Model and Metamodel Representation . . . 221
6.7 Model and Metamodel Differences . . . 227
6.8 Summary . . . 238

7 Operators 241
7.1 Related Work . . . 241
7.2 Template to describe Operators . . . 241
7.3 List of Bidirectional Operators . . . 243
7.4 Summary . . . 262

8 Implementation 263
8.1 Overview . . . 263
8.2 Modeling Infrastructure . . . 264
8.3 Operator Implementation . . . 267
8.4 Adapters . . . 271
8.5 Visualizations . . . 279
8.6 Summary . . . 280

IV Application 281

9 Access Data 283
9.1 Application Domain . . . 284
9.2 Integration of existing Data Sources . . . 299
9.3 Definition of a new View(point) . . . 313
9.4 Validation Scenarios . . . 328
9.5 Summary: Contributions . . . 368

10 SEIS Viewpoints 373
10.1 Application Domain . . . 373
10.2 Ensure Consistency between existing Data Sources . . . 374
10.3 Define new Viewpoints . . . 382
10.4 Validation Scenarios . . . 384
10.5 Summary: Contributions . . . 384

11 Knowledge Management **387**
11.1 Application Domain 388
11.2 Integration of existing Data Sources 404
11.3 Definition of a new View(point) 425
11.4 Validation Scenarios 434
11.5 Summary: Contributions 452

12 Application in general **455**
12.1 Process of Configuration 455
12.2 Recommendations for Orchestrations 458
12.3 Process of Use 462
12.4 Summary 463

V Achievements **465**

13 Evaluation **467**
13.1 Fulfillment of Requirements 467
13.2 Properties of Operators 469
13.3 Characteristics of Orchestrations 473
13.4 Conceptual Discussions on MoConseMI 478
13.5 Summary of the Evaluation 481

14 Conclusion **483**
14.1 Contributions 483
14.2 Preconditions 489
14.3 Limitations 490
14.4 Outlook 496
14.5 Summary of the Thesis 499

VI Appendix **503**

A Collected Lists **505**
A.1 Parts of the Ongoing Example 505
A.2 List of Definitions 505
A.3 List of Figures 506
A.4 List of Tables 510
A.5 List of Code Listings 510

Bibliography **513**

Contents in Detail

Contents at a Glance 7

Contents in Detail 11

Typesetting Conventions 19

I Introduction 23

1 Motivation 25
1.1 Multi-perspective modeling . . . 26
1.2 Challenges . . . 31
1.2.1 Model Consistency . . . 31
1.2.2 Reuse existing Artifacts . . . 36
1.2.3 Define new View(point)s . . . 39
1.3 Aims . . . 41
1.3.1 Objectives . . . 42
1.3.2 Demarcation . . . 43
1.3.3 High-level Requirements . . . 46
1.4 Summary & Outline . . . 47

II Foundations 49

2 Basic Concepts 51
2.1 Views and Viewpoints . . . 54
2.2 Modeling . . . 58
2.2.1 Model . . . 59
2.2.2 Metamodel . . . 60
2.2.3 Model Transformation . . . 67
2.3 Consistency . . . 71
2.4 Stakeholders . . . 79
2.4.1 User . . . 80
2.4.2 Methodologist . . . 81
2.4.3 Platform Specialist . . . 82
2.4.4 Adapter Provider . . . 82
2.5 Technical Spaces . . . 84
2.5.1 Related Work: Technical Spaces . . . 84
2.5.2 Descision: EMF . . . 86
2.5.3 Foundations of EMF . . . 87

2.6 Summary 89

3 Related Work 93

3.1 Criteria for Classification 94
3.1.1 Inter-Model Consistency 95
3.1.2 Levels of Heterogeneity 95
3.1.3 Multi-Directionality 97
3.1.4 Stakeholders who decide 97
3.1.5 Summary 99
3.2 Overall Realization Techniques 99
3.2.1 Intermediate Model 100
3.2.2 Explicit Links 101
3.2.3 Change Propagation 102
3.2.4 Choose from multiple Fixes 106
3.2.5 External Support for Multi-Models 108
3.2.6 Summary 108
3.3 Synthetic Approaches 108
3.3.1 Synthetic Consistency Preservation 108
3.3.2 Synthetic View Definition 119
3.4 Single Underlying Model (SUM) 120
3.5 Projectional Approaches 121
3.5.1 OSM 124
3.5.2 Vitruvius 126
3.5.3 RSUM 129
3.5.4 Combining Views into a SUM 131
3.5.5 Projectional View Definition 134
3.6 Further Research Areas 135
3.6.1 UML 136
3.6.2 Domain-Specific Languages (DSLs) 137
3.6.3 Data Bases 139
3.6.4 Ontologies 142
3.6.5 Enterprise Applications 144
3.7 Summary: Lessons Learned 146

4 Requirements 153

4.1 Functional Requirements 154
4.2 Technical Requirements 157
4.3 Summary 158

III Approach 161

5 MoConseMI at a glance 163

5.1 Design Decisions 163
5.1.1 Bottom-Up: Existing Artifacts as Starting Point 164
5.1.2 Projectional with an explicit SUM as End Point 165
5.1.3 Adjustable Approach towards an essential SUM 167
5.1.4 Model Synchronization for Change Propagation 168
5.1.5 Methodologists decide the final Fix 169

5.1.6 Reuse Parts of Model Transformations . . . 170
5.2 Overview of the Approach . . . 171
5.2.1 Specify Consistency . . . 171
5.2.2 Fix Inconsistencies automatically . . . 173
5.2.3 Initialize SU(M)M . . . 176
5.2.4 Develop Adapter . . . 178
5.3 Summary: MoConseMI . . . 179

6 Design 185
6.1 Operators as Transformations . . . 185
6.1.1 Related Work: Operator-based Approaches . . . 186
6.1.2 Related Work: Model Transformations . . . 188
6.1.3 Design . . . 189
6.2 Metamodel Decisions . . . 192
6.2.1 Related Work: Model Co-Evolution . . . 193
6.2.2 Design . . . 196
6.3 Model Decisions . . . 198
6.3.1 Related Work . . . 199
6.3.2 Design . . . 200
6.4 Operator Combination . . . 203
6.4.1 Related Work . . . 203
6.4.2 Chain of Operators . . . 204
6.4.3 Integration of existing Data Sources . . . 205
6.4.4 Definition of new View(point)s . . . 209
6.4.5 Final Result: Tree . . . 213
6.5 Operator Execution . . . 213
6.5.1 Related Work . . . 214
6.5.2 Executing single unidirectional Operators . . . 214
6.5.3 Execution Loop . . . 217
6.5.4 Initial Execution . . . 219
6.5.5 Ongoing Change Propagation . . . 220
6.6 Model and Metamodel Representation . . . 221
6.6.1 Related Work . . . 222
6.6.2 Metamodel Representation . . . 222
6.6.3 Model Representation . . . 223
6.6.4 UUIDs . . . 225
6.6.5 Adapters . . . 226
6.7 Model and Metamodel Differences . . . 227
6.7.1 Related Work . . . 228
6.7.2 Model Difference Representation . . . 229
6.7.3 Model Difference Co-Evolution . . . 235
6.7.4 Branch Difference Calculation . . . 236
6.8 Summary . . . 238

7 Operators 241
7.1 Related Work . . . 241
7.2 Template to describe Operators . . . 241
7.3 List of Bidirectional Operators . . . 243
7.3.1 AddDeleteOppositeRelation . . . 243

7.3.1.1 AddOppositeRelation . . . 243
7.3.1.2 DeleteOppositeRelation . . . 244
7.3.2 AddDeleteAttribute . . . 244
7.3.2.1 AddAttribute . . . 245
7.3.2.2 DeleteAttribute . . . 246
7.3.3 DeleteAddNamespace . . . 246
7.3.3.1 DeleteNamespace . . . 246
7.3.3.2 AddNamespace . . . 247
7.3.4 ChangeAttributeType . . . 247
7.3.4.1 ChangeAttributeType . . . 247
7.3.5 ChangeModel . . . 249
7.3.5.1 ChangeModel . . . 249
7.3.6 ChangeMultiplicity . . . 250
7.3.6.1 ChangeMultiplicity . . . 250
7.3.7 MergeSplitClasses . . . 251
7.3.7.1 MergeClasses . . . 251
7.3.7.2 SplitClass . . . 254
7.3.8 RenameClassifier . . . 257
7.3.8.1 RenameClassifier . . . 258
7.3.9 RenameFeature . . . 258
7.3.9.1 RenameFeature . . . 258
7.3.10 ReplaceAttributeByReference . . . 259
7.3.10.1 ReplaceAttributeByReference . . . 259
7.3.10.2 ReplaceReferenceByAttribute . . . 261
7.4 Summary . . . 262

8 Implementation **263**
8.1 Overview . . . 263
8.2 Modeling Infrastructure . . . 264
8.3 Operator Implementation . . . 267
8.3.1 Unidirectional Operators . . . 267
8.3.2 Bidirectional Operators . . . 270
8.3.3 Java-API for Orchestration . . . 270
8.4 Adapters . . . 271
8.4.1 Dynamically typed EMF . . . 272
8.4.2 Statically typed EMF . . . 273
8.4.3 Excel . . . 273
8.4.4 CSV . . . 275
8.4.5 Xtext . . . 278
8.5 Visualizations . . . 279
8.6 Summary . . . 280

IV Application **281**

9 Access Data **283**
9.1 Application Domain . . . 284
9.1.1 DataSource Htpasswd . . . 284
9.1.2 DataSource Authz . . . 285

9.1.3 DataSource Htaccess 289
9.1.4 SU(M)M 290
9.1.5 New ViewPoint Overview 292
9.1.6 Realization Overview 295
9.2 Integration of existing Data Sources 299
9.2.1 Integrate Htpasswd and Authz 302
9.2.2 Integrate Htaccess 307
9.3 Definition of a new View(point) 313
9.4 Validation Scenarios 328
9.4.1 Initialization by Execution 328
9.4.2 Scenario: renamed Mapping, reload externally (Authz) 332
9.4.3 Scenario: removed Mapping, reload externally (Authz) 336
9.4.4 Scenario: added Mapping, reload externally (Authz) 340
9.4.5 Scenario: added Mapping, reload externally (Authz) 344
9.4.6 Scenario: removed Mapping, reload externally (Authz) 348
9.4.7 Scenario: renamed HtaccessUser, reload externally (Htaccess) 352
9.4.8 Scenario: change Htaccess right for Bob (Overview) 356
9.4.9 Scenario: change number of Authz rights for Bob (Overview) . 359
9.4.10 Scenario: change the name of Bob to David (Overview) 359
9.4.11 Scenario: removed user, reload externally (Overview) 363
9.5 Summary: Contributions 368

10 SEIS Viewpoints 373
10.1 Application Domain 373
10.1.1 Conceptual Viewpoint 373
10.1.2 Module Viewpoint 373
10.1.3 Execution Viewpoint 373
10.1.4 Code Viewpoint 374
10.1.5 Topology Viewpoint 374
10.1.6 Data Viewpoint 374
10.2 Ensure Consistency between existing Data Sources 374
10.2.1 Conceptual — Module 375
10.2.2 Module — Data 375
10.2.3 Module — Code 376
10.2.4 Module — Execution — Code 379
10.2.5 Execution — Topology 380
10.2.6 Realization Overview 382
10.3 Define new Viewpoints 382
10.3.1 Conceptual-Module-Mappings 383
10.3.2 Intersections 383
10.3.3 ModulesOnly 384
10.3.4 LayersOnly 384
10.4 Validation Scenarios 384
10.5 Summary: Contributions 384

11 Knowledge Management 387
11.1 Application Domain 388
11.1.1 DataSource Work 388
11.1.2 DataSource Employees 391

11.1.3 DataSource Tasks 393
11.1.4 DataSource Materials 395
11.1.5 SU(M)M 396
11.1.6 New ViewPoint Costs 398
11.1.7 Realization Overview 401
11.2 Integration of existing Data Sources 404
11.2.1 Improve Work 404
11.2.2 Integrate Employees with Work 408
11.2.3 Integrate Tasks with Work 413
11.2.4 Integrate Materials with Work 420
11.3 Definition of a new View(point) 425
11.4 Validation Scenarios 434
11.4.1 Initialization by Execution 434
11.4.2 Scenario: Create new Work 437
11.4.3 Scenario: Change real Human Costs 442
11.4.4 Scenario: Delete existing Work 442
11.4.5 Scenario: Renamed Task in Costs View 446
11.4.6 Scenario: Change Salary 449
11.5 Summary: Contributions 452

12 Application in general 455
12.1 Process of Configuration 455
12.2 Recommendations for Orchestrations 458
12.2.1 Explicit Links between Models 459
12.2.2 Reduce Redundancies 460
12.2.3 Remaining Dependencies in the SUM 460
12.2.4 New View(point)s 461
12.3 Process of Use 462
12.4 Summary 463

V Achievements 465

13 Evaluation 467
13.1 Fulfillment of Requirements 467
13.2 Properties of Operators 469
13.2.1 Formal Properties 469
13.2.2 Completeness of Operators 470
13.2.3 Complexity of Operators in O-Notation 471
13.2.4 Reusability of Operators 471
13.2.5 Imperative vs Declarative Operators 471
13.2.6 Design of Operators revised 472
13.3 Characteristics of Orchestrations 473
13.3.1 Language Evolvability 473
13.3.2 MoConseMI without reusing Data Sources 474
13.3.3 Characteristics of the SU(M)M 475
13.3.3.1 Quality of the SU(M)M 475
13.3.3.2 Content of the SU(M)M 476
13.3.3.3 Large (Meta)Models 476

13.3.4 Reusability of Orchestrations . . . 477
13.4 Conceptual Discussions on MoConseMI . . . 478
13.4.1 Reuse existing Modeling Techniques . . . 478
13.4.2 Integrate Data Sources with different Abstraction Levels . . . 479
13.4.3 Integrate other Research into MoConseMI . . . 479
13.4.4 Integrate MoConseMI into other Applications . . . 480
13.4.5 Intra-Model Consistency . . . 480
13.5 Summary of the Evaluation . . . 481

14 Conclusion **483**
14.1 Contributions . . . 483
14.1.1 Contributions to Model Consistency . . . 483
14.1.1.1 New SUM Approach . . . 483
14.1.1.2 Characteristics of MoConseMI revised . . . 484
14.1.2 Contributions to other Research Areas . . . 486
14.1.2.1 Traceability . . . 486
14.1.2.2 Round-trip Engineering . . . 487
14.1.2.3 Model Co-Evolution . . . 487
14.1.2.4 Model Transformations . . . 488
14.1.2.5 Model Differences . . . 488
14.1.3 Contributions to Application Domains . . . 488
14.1.3.1 SEIS Architecture . . . 488
14.1.3.2 Table-oriented Data Management . . . 489
14.2 Preconditions . . . 489
14.2.1 Data as Model . . . 489
14.2.2 Supportable Consistency Goals . . . 489
14.3 Limitations . . . 490
14.3.1 Limitations of the Approach . . . 490
14.3.1.1 Termination . . . 491
14.3.1.2 Performance . . . 492
14.3.1.3 Skills of Stakeholders . . . 493
14.3.2 Limitations of the Implementation . . . 495
14.3.3 Limitations of the Evaluation . . . 495
14.4 Outlook . . . 496
14.4.1 Outlook for the Approach . . . 496
14.4.2 Outlook for the Implementation . . . 497
14.4.3 Outlook for further Applications . . . 498
14.5 Summary of the Thesis . . . 499

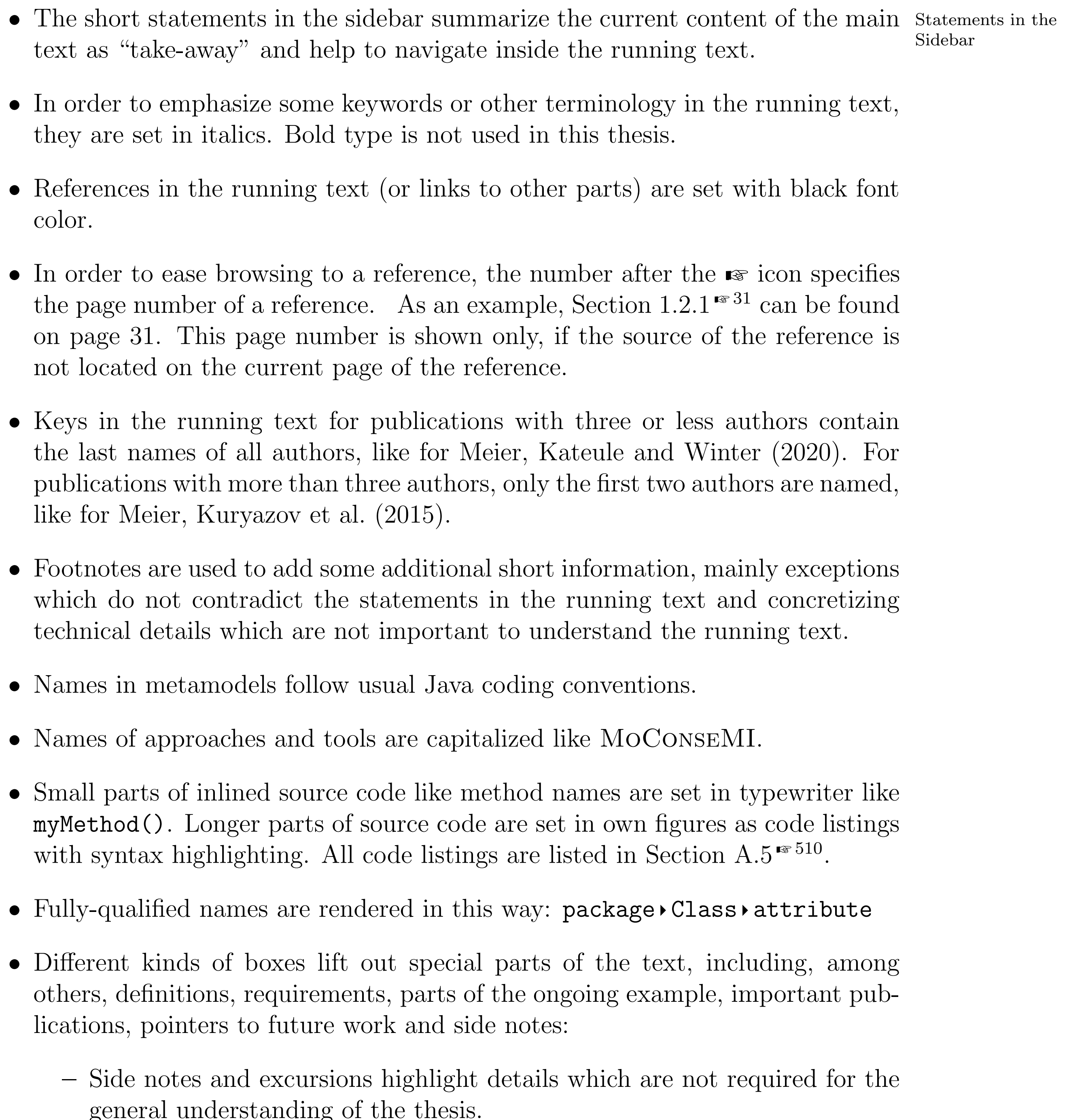
VI Appendix **503**

A Collected Lists **505**
A.1 Parts of the Ongoing Example . . . 505
A.2 List of Definitions . . . 505
A.3 List of Figures . . . 506
A.4 List of Tables . . . 510
A.5 List of Code Listings . . . 510

Bibliography **513**

Typesetting Conventions

To support the structure and readability, this thesis uses some conventions, renderings and fonts. They are introduced here and are valid in all parts of the thesis.



Statements in the Sidebar

- The short statements in the sidebar summarize the current content of the main text as "take-away" and help to navigate inside the running text.
- In order to emphasize some keywords or other terminology in the running text, they are set in italics. Bold type is not used in this thesis.
- References in the running text (or links to other parts) are set with black font color.
- In order to ease browsing to a reference, the number after the ☞ icon specifies the page number of a reference. As an example, Section 1.2.1☞31 can be found on page 31. This page number is shown only, if the source of the reference is not located on the current page of the reference.
- Keys in the running text for publications with three or less authors contain the last names of all authors, like for Meier, Kateule and Winter (2020). For publications with more than three authors, only the first two authors are named, like for Meier, Kuryazov et al. (2015).
- Footnotes are used to add some additional short information, mainly exceptions which do not contradict the statements in the running text and concretizing technical details which are not important to understand the running text.
- Names in metamodels follow usual Java coding conventions.
- Names of approaches and tools are capitalized like MoConseMI.
- Small parts of inlined source code like method names are set in typewriter like `myMethod()`. Longer parts of source code are set in own figures as code listings with syntax highlighting. All code listings are listed in Section A.5☞510.
- Fully-qualified names are rendered in this way: `package‣Class‣attribute`
- Different kinds of boxes lift out special parts of the text, including, among others, definitions, requirements, parts of the ongoing example, important publications, pointers to future work and side notes:
  - Side notes and excursions highlight details which are not required for the general understanding of the thesis.

Side note / Excursion

These side notes and excursions are set in dark gray boxes. They are not set as footnotes, since some of them are too long or contain graphics.

– All definitions in this thesis are rendered inside dark gray boxes like for the following Definition 1:

Definition 1: **Definition of Definitions**

To make important terminology clear, this thesis defines several terms in form of definitions.

A list of all definitions can be found in Section A.2 ☞506.

– Requirements are documented in the following way:

Requirement R 0: Structure of Requirements

Requirements must be documented in a structured way.

Requirement R 0 consists of a label ("R 0"), a short summary ("Structure of Requirements") and the main requirement as text ("Requirements must be documented in a structured way."). Requirements can be concretized by sub-requirements, its label would be R 0.1 for example. In the digital version of this thesis, clicking on parts of a requirement allows to jump to its main definition, not to its first occurrence. A list of all requirements of this thesis can be found in Section 4.3 ☞158.

– This thesis develops an ongoing example, which is extended throughout the thesis. Each extension is rendered inside a box like this one:

Ongoing Example, Part 1: **Typesetting Conventions** ← List →

Parts of this example are highlighted by rendering them in boxes like this one. Clicking on the left/right arrows in the top-right corner of the box allows to jump to the previous/next part of this ongoing example. "List" refers to Section A.1 ☞505 listing all parts of this ongoing example.

In Part 2 ☞25 of the ongoing example, the ongoing example is introduced. Note, that these parts of the ongoing example are different from the parts of this document, that contain chapters and are numbered with Roman letters.

– Important aspects to extend the results of this thesis are indicated as future work using boxes like the following one:

Future Work: **Outlook to Future Work**

Important future work is made explicit at the place, where it is identified. All future work is picked up and summarized in Section 14.4 ☞496.

- Important statements for this thesis are set in darf gray boxes:

Important Statement

These boxes are used to clarify the problem statement, objectives, deliverables and final result of this thesis.

- Own publications (i. e. publications with the author of this thesis being one of their authors), which are relevant for this thesis, are rendered in prominent way:

Related MoConseMI Publication

Johannes Meier and Andreas Winter (2016): *Towards Metamodel Integration Using Reference Metamodels.* In: Proceedings of the 4th Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO 2016), pp. 19–22.

This publication is cited as Meier and Winter (2016) in this thesis.

- Operators are set in the following ways: ⇄AddRemoveAssociation indicates a bidirectional operator, which consists of the unidirectional operator →AddAssociation in main direction and the unidirectional operator ←RemoveAssociation in inverse direction.

- To indicate the position or role of (meta)models for an application realized with MoConseMI, the following renderings are used:

  - Data sources as introduced in Definition 4☞37 are rendered as DataSource.
  - New view(point)s which are introduced in Definition 5☞40 are rendered as NewView(Point).
  - The SU(M)M as introduced in Section 3.4☞120 is rendered as SU(M)M.
  - Intermediate nodes at position $i \in \mathbb{N}$ as introduced in Section 6.4.2☞204 are rendered as ❶ i, while a chain of intermediate nodes $k, k+1, \ldots, l-1, l$ with $k, \ldots, l \in \mathbb{N}$ is rendered as $k \rightarrow l$.

  Using terms like view(point)s indicates, that views (on model level) and their viewpoints (on metamodel level) are both affected. The same counts for SU(M)M comprising the SUM and its SUMM. Linguistically not that obvious, a data source consists of a view and its viewpoint, too (see Definition 4☞37). This counts also in general for a (meta)model comprising the model and its metamodel.

- Changes in graphics are visualized by using colors to draw the changed elements: Added elements are rendered with red color. Deleted elements are rendered with green color.

- As the title of this thesis indicates, consistency is an important concept, which is concretized into consistency goals and consistency rules in Section 2.3☞71. Consistency goals and consistency rules are rendered in the following way:

Consistency Goal C 0 | Requirements + Java

Requirements must be linked with their fulfilling Java methods.

This Consistency Goal C 0 summarizes a consistency issue in the ongoing example between the data sources Requirements and Java.

Consistency Rule C 0 a | for C 0

Links between requirements and fulfilling methods are added manually.

This Consistency Rule C 0 a adds a hint, how to realize its related Consistency Goal C 0.

Legend
- Mandatory
- Optional
- Or
- Xor
- Sub-Diagram

- This thesis uses feature models (Kang, Cohen et al., 1990; Nešić, Krüger et al., 2019) to depict possible features including their dependencies and restricting constraints. The legend for feature models is placed in the sidebar and explains their concepts. An example can be found in Figure 3.2☞100. Selected features are marked with light gray color.

Part I

Introduction

This part motivates this thesis by establishing multi-perspective modeling as usual paradigm for the development of software-intensive systems. Since these perspectives are realized with models and depend on each other in order to jointly represent the same system under development, ensuring the consistency of depending models is an important challenge. The development and evaluation of a new approach for overcoming this challenge is the objective of this thesis.

Chapter 1

Motivation

Since today's software-intensive systems increase regarding size, complexity and heterogeneity, their development cannot be realized by single persons anymore. To manage the development of systems, different persons with different skills are involved. They use different tools tailored to their tasks. These different tools deal with different information and represent *multiple perspectives* of the system under development.

multiple Perspectives

Since perspectives describing the same system are realized with different tools, but have interrelations like overlaps or inter-perspective constraints, the data managed by different perspectives depend on each other. Therefore, inconsistencies between perspectives can occur, if stakeholders change information in one perspective only and this information is contained also in another perspective. To fix such inconsistencies, the other perspective must be changed accordingly. Since such fixes are error-prone and time-consuming, they should be automated. This problem is illustrated along an example in the following box.

Inconsistencies between multiple Perspectives

Ongoing Example, Part 2: **Introduction** ← List →

As running example, a small and strongly simplified Software Development Project is chosen with perspectives for requirements, UML class diagrams and Java source code. Goal of this project is to develop a new information system managing students and lectures at a university. Initially, this project does not use any approach or tool for ensuring consistency automatically.

simplified Software Development Project

To keep the example manageable, only *three different perspectives* are described. To cover multiple steps of a software development lifecycle, requirements as early artifacts (list of textual requirements), UML class diagrams to specify the design like data models (only classes with associations), and Java source code for the implementation are chosen (only classes with methods).

Since these three perspectives together describe the system under development, they overlap content-wise: The same classes are modeled with class diagrams and implemented in the source code. When a software architect renames an existing class in the class diagram only, this results in an inconsistency with the source code. Renaming the class in the source code accordingly fixes this inconsistency. This renaming should be automated to relieve the software architect and to keep the system consistent.

As another example, each attribute defined in the class diagram must have a corresponding getter-method in the source code. When a software architect deletes the attribute in the class diagram only, this results in an inconsistency with the source code. Deleting the corresponding getter in the source code fixes this inconsistency. This deletion should be automated to relieve the software architect and to keep the system consistent.

There is also an consistency issue for which no automation is available: Requirements should be traceable to those methods which implement the functionalities specified by

these requirements (and vice versa). This can be realized with traceability links between requirements and their fulfilling methods. When a developer creates a new method in order to fulfill a particular requirement, the developer must also create a traceability link between both. Without creating this traceability link, the developer introduced a new inconsistency.

ongoing Example for the whole Thesis

This sketched example is used as ongoing example for the whole thesis. Therefore, this example is not described completely here, but is picked up continuously in the following sections for conceptual clarifications and technical realizations. Until the last part, the described project will have applied the new approach of this thesis. Variants of this example are already used in some publications (Meier and Winter, 2018a; Meier, Klare et al., 2019; Meier, Werner et al., 2020). This ongoing example is described in more detail regarding the existing perspectives in Part 5[☞37] of the ongoing example and regarding the consistency issues in Part 3[☞34] of the ongoing example.

The problem are inconsistencies between related perspectives: This problem is not restricted to this small ongoing example or software development on a larger scale. Instead, it is a general problem of all perspectives which together describe the same system. It occurs also in, among others, the management of distributed access rights or knowledge management in research projects. Therefore, this thesis aims to solve this problem in general and shows application examples also for these domains in Chapter 9[☞283] and Chapter 11[☞387].

Challenges in multi-perspectives Consistency Preservation

In order to solve the problem of consistency between perspectives, some challenges must be overcome: There are different kinds of consistency issues between perspectives, i. e. redundancies when same information is contained in multiple perspectives, links establishing explicit connections between elements of two or more perspectives and constraints in form of additional rules which must hold between perspectives. These consistency issues are project-specific, like the classes in the ongoing example as an example for redundancies, since each class conceptually exists only once, but is represented twice in source code and class diagrams. Additionally, already existing data must be reused by approaches. Since such data conform to the structure of their tools, these structures must be reused, too. These challenges are elaborated in Section 1.2[☞31].

Outline of the Chapter

The main objective of this thesis (Section 1.3.1[☞42]) is to overcome those challenges, whose outline is motivated in Section 1.4[☞47]. Before that, Section 1.1 describes the idea of modeling with multiple perspectives for system development in more detail and derives problems in multi-perspective modeling.

1.1 Multi-perspective modeling

Introduction of Term *multiple Perspectives*

During the development of systems including software-intensive systems, a perspective allows to focus on only some selected parts of the system, instead of dealing with all information of the whole system under development. This reflects the principle of abstraction, which is central for modeling (see Definition 12[☞59]). Using *multiple perspectives* enables different stakeholders to concentrate on their individual tasks *and* to work together on the same system.

Reasons for using multiple Perspectives

With growing size, complexity and heterogeneity of software-intensive systems, also their description amounts during development increase accordingly. This requires using multiple perspectives due to the following reasons:

Size At some point, the development by a single person becomes inefficient or even impossible due to the systems size. To align the development power to the systems size, *multiple persons* or even teams are required for development. Therefore, the systems needs to be decomposed into different parts, which are manageable by single persons. These parts represent different perspectives. As an example from industry,

Burden, Heldal and Whittle (2014) reported on single models containing 8000 UML sequence diagrams or converted 3000 pages of specifications, whose size decreased its understanding and maintenance.

Complexity To manage increasing complexity regarding functionality, the system needs to be decomposed into multiple parts, i. e. *multiple sub systems* representing different functional parts of the system. Different sub systems can be treated as different perspectives and provided to different groups of persons for development. Thus different perspectives can cover different sub systems. As an example, Bucchiarone, Cabot et al. (2020, p. 8) require different perspectives to cope with the complexity of smart city applications.

Heterogeneity The heterogeneity of systems and their sub systems regarding design and realization techniques (Lee, 2010) requires multiple tasks during development with different skills. Involving multiple persons with different skills or specializations helps to distribute development tasks according to the required skills. Thus, different perspectives are tailored to different skills of different persons. As an example, the software architect is provided with higher-level information like data classes, while details of the implementation like single Java statements are hidden.

Summarizing, the use of multiple perspectives tailors system descriptions to characteristics of sub systems and to the skills of the involved persons.

In order to realize perspectives for developing systems, modeling is used, since modeling is a fundamental activity in the context of software engineering (Ludewig, 2004): The development of systems is done via modeling using multiple perspectives, which leads to the term *multi-perspective modeling.* Modeling represents all information as models conforming to metamodels, since metamodels define the structure for and the concepts usable by models. While information is represented as model, the structure of this information, i. e. the concepts, is represented as metamodel. Summarizing, *everything is a model* (Bézivin, 2005), including source code (Heidenreich, Johannes et al., 2009).

multi-perspective Modeling

Therefore, required terminology is concretized according to modeling terminology and to the ISO Standard for Architecture Description 42010:2011 (IEEE, 2011), which is detailed in Chapter 2☞51: Instead of persons involved in system development, the term *stakeholder* is used (see Definition 8☞55). The interests of stakeholders in the system are described by *concerns* (see Definition 9☞55). Perspectives reflect these concerns by defining, which parts of systems are selected, and are named *viewpoints* (see Definition 10☞55). When a stakeholder looks at one concrete system, a viewpoint targeting his concerns determines, what is shown to him. The shown result is named *view* (see Definition 11☞56). In this thesis, views are technically realized as models (see Definition 12☞59), while viewpoints are technically realized as metamodels (see Definition 13☞61). These concepts help to clearly distinguish the particular, provided information about the particular system of perspectives as views from the general, structural concepts of perspectives as viewpoints.

Terminology

View vs Viewpoint

Using multiple perspectives in form of viewpoints is an established and often used concept to model complex systems, as the following examples demonstrate:

widely established Use of multi-perspective Modeling

- The Unified Modeling Language (UML) (Object Management Group, 2017) uses different diagrams to focus on different aspects of the system under developement, e. g. UML class diagrams for static aspects of a system like data classes or UML activity diagrams and UML state machines to describe dynamic aspects.
- For the development and management of software systems and organizations, Winter (2000) identified lots of visual modeling languages representing multiple perspectives for structures, tasks, processes and data including organization charts, communication networks, data flow diagrams, use case diagrams, state charts, petri nets, Gantt

charts, decision tables, object diagrams and collaboration diagrams. These languages are classified and their general, overlapping concepts are identified and represented in form of reference metamodels for their conceptual integration.

- For automated production systems engineering, Feldmann, Herzig et al. (2015b) propose multiple viewpoints including SysML models for systems engineers to define the mechatronic architecture, MATLAB/Simulink models for simulating and evaluating properties of the system and unspecified viewpoints for non-functional requirements and test cases. Feldmann, Wimmer et al. (2016) add CAD drawings for geometric visualizations for mechanical engineers and circuit diagrams for electrical engineers.
- For Industry 4.0 and its concerns like 3D modeling, architectures, specification exchange, formal modeling and simulation, Wortmann, Barais et al. (2020) collected various different languages including AutomationML, UML modeling, SysML, AutoCAD, MATLAB/Simulink and petri nets.
- Pohlmann, Meyer et al. (2014) present multiple viewpoints for developing hardware platforms for smart cyber-physical systems (CPSs), including viewpoints for resource types describing possible kinds of resources including its main parts, resource instances describing all required instances of the defined resource types for the particular CPS, platform types describing possible kinds of platforms including its main parts, platform instances describing all required instances of the defined platform types for the particular CPS and allocation planning for mapping software components to the defined platform instances for execution.
- Bucchiarone, Cabot et al. (2020, p. 8) require different perspectives to cope with the complexity of smart city applications.
- For enterprise modeling, Frank (2014) propose different viewpoints including strategy nets and value chain diagrams for defining business strategies, business process diagrams for modeling the organization, and class diagrams for describing the underlying information system (Section 3.6.5 ☞ 144).

cover Problem in application-domain independent Way

This small excerpt of examples using multi-perspective modeling in different application domains shows its wide usage and justifies to the cover the problem of inconsistencies between corresponding views in a generic, i. e. application domain-independent way.

multiple Views are enabled by Redundancy

The different viewpoints are established by, among others, different tools (Broy, Feilkas et al., 2010), environments, file formats, and domain-specific languages (DSLs) (France and Rumpe, 2007). These tools allow to manage some information of the whole system as views. For that, each tool provides its own view to its current user, i. e. stakeholder. In general, different tools are used by multiple stakeholders at the same or different time and possibly at different locations. That leads to the fact, that information of the system under development is managed separately or is duplicated in multiple views by multiple stakeholders. These problems are also reported by Broy, Feilkas et al. (2010) in the domain of embedded software-intensive systems and by Thomas and Nejmeh (1992) as important challenge for tool integration. The benefit of this idea is, that each stakeholder can work independently from all other stakeholders and on only that information which is currently relevant for him. Therefore, redundancy of the information presented in views enables environments with multiple views and viewpoints working on the same system.

Problems of multiple Viewpoints:

technical Separation vs contentwise Interrelations

While the development with multiple viewpoints is a widely used principle, since it supports different stakeholders with tailored views, it introduces also some problems: Due to the tools which manage only the current information in different formats, data bases or files, the different views are separated on technical level. But the different views describe

different aspects of the same system, which form one system. Therefore, the different views are interrelated contentwise to describe the same system under development in consistent way.

Interrelations between Views always exist

Between multiple views of the same system, there are always interrelations: If there are no interrelations between two views, they describe two independent systems, instead of two sub systems of the same system. The practical background for this theoretic finding is, that some information is required, how to stick two views together to form the whole system. The points where to stick the views together represent the interrelations between the views. Nevertheless, Atkinson and Tunjic (2014a, p. 49) recommend to minimize the interrelations of viewpoints, but in the sense of an optimization problem.

Dependencies

Main forms of these interrelations are redundancies and dependencies: Redundancies are characterized by information which are represented in multiple views (Hailpern and Tarr, 2006). Goldschmidt, Becker and Burger (2012) identified elements occurring redundantly within multiple views as important feature of views, but use the term overlaps instead. Dependencies define relations which hold between information encoded in different views. Since these information could also be the same, redundancies are a special case of dependencies. Therefore, this thesis calls all interrelations between view(point)s *dependencies*. This thesis assumes no order between two depending views regarding time or construction process, while Persson, Torngren et al. (2013) classify two depending views as one input view and one output view for each other.

Inconsistencies

All kinds of dependencies can lead to *inconsistencies* between views: If a relation holds between two views, they are consistent to each other regarding this relation. If a stakeholder changes one of the views, this relation can be hurt, if the other view is not changed accordingly to ensure a valid relation. In that case, inconsistencies are introduced and must be fixed. The potential for inconsistencies grows with the number of involved views (Mussbacher, Amyot et al., 2014).

changing Views requires Changes in related Views

As motivated above, changing only one view is the usual way of systems development in multi-view environments. At the same time, consistency relations hold between multiple views. Therefore, stakeholders working on views usually introduce inconsistencies, which must be fixed in the related views afterwards. This problem is an inherent one, since each stakeholder works only on a single view and has no chance to directly change the other views accordingly. This can cause also organizational problems in bigger companies, if multiple stakeholders have to know about other views and their responsible persons and have to agree with them on joint changes in different views, as exemplarily reported by Burden, Heldal and Whittle (2014). Summarizing, changing one view requires to change related views according to consistency relations. If all consistency relations are fulfilled, i. e. there are no inconsistencies, consistency is achieved.

Consistency $\Leftrightarrow$ $\nexists$ Inconsistency

manual Consistency Preservation is error-prone

If consistency between views is ensured *manually*, the stakeholders change their view and have to change all related views accordingly. This step is error-prone, since required adjustments may be forgotten, some related information may not be found or the applied changes may be incorrect. Another problem is, that stakeholders usually know only “their own” view, lack knowledge of “other foreign” views (which is the idea of having multiple views!) or do not even know about the existence of related views, which limits the chance to find senseful changes fixing inconsistencies in the other views (Hailpern and Tarr, 2006). Another problem is incomplete understandings of stakeholders about the reasons for occurred inconsistencies (Grundy, Hosking and Mugridge, 1998, p. 975). Even if other views can be fixed in principle, with a large number of views it is difficult to keep track of views that are still inconsistent and views that are already fixed (Burden, Heldal and Whittle, 2014). Such manual consistency preservation requires high effort in time and complexity, even though the consistency preservation usually follows strict rules, which could be automated. As idea, specialists describe the strict rules for consistency once, so that related views can be changed accordingly automatically, after the stakeholder changed his view

manually. Finally, the amount of involved views and requests for instant feedback for users requires an automation (Egyed, Zeman et al., 2018).

unstructured Consistency Preservation requires high Effort

To get rid of such manual work, sometimes transformations between two views are written to (re-)generate one view, if the other view changed. Alternatively, scripts are hacked to synchronize some information between views (Burden, Heldal and Whittle, 2014). Usually, only two views (e. g. source and target) are involved in such transformations. In practice, multiple pairs of such transformations might be written by different users, lacking synchronization due to missing knowledge of foreign views and lacking a holistic understanding of the whole system. Sindico, Natale and Sangiovanni-Vincentelli (2012) report on a bigger industrial development process with multiple transformations and scripts between the involved artifacts. Usually, such automation efforts are done in an unstructured way without general idea, approach or framework behind. These findings motivate the development of a new approach for rigorously ensuring consistency between multiple views.

Problem Statement

Summary: changing separated Views introduces Inconsistencies with related Views

In order to manage size, complexity and heterogeneity of software-intensive systems, multiple viewpoints are used to describe parts of the system regarding the concerns of involved stakeholders. These views are used independently from other views, but all views together describe the whole system under development and therefore depend on each other.

If a stakeholder changes one view, this view may become inconsistent with the other views. Inconsistencies between views prevent successfully developing the system and must be fixed to realize the desired system in a consistent way. Doing this in manual or in an unstructured way is error-prone and time-consuming.

Summarizing, *the problem are inconsistencies between multiple views which are not automatically fixed.* This thesis aims at overcoming this problem in a structured and automated way.

Consistency of Views is important in Literature

This problem of upcoming inconsistencies between different views is an important one in literature: Changing other views according to the change made in one view is called *change propagation* by Persson, Torngren et al. (2013). Stevens (2008) emphasizes the need for handling inconsistencies in views which are manually changed by users, in contrast to views which are completely and automatically generated from other views. In 2014, Mussbacher, Amyot et al. (2014, p. 188) state, that inconsistencies between artifacts are still a major problem, which is not solved during the last 20 years. Still in 2020, Bucchiarone, Cabot et al. (2020) emphasize the need for traceability and consistency across different views. Mohagheghi, Gilani et al. (2013a, p. 102) identified, that companies may expect consistency preservation as benefit when using modeling techniques. France and Rumpe (2007) propose synchronization transformations to propagate changes from one view to other views. Persson, Torngren et al. (2013) give a broad overview of groups of approaches for multi-view modeling. The spectrum of existing tools and approaches targeting inter-view consistency in various forms is investigated in Chapter 3 ☞ 93 and its broadness shows the importance of this problem. Additionally, that section shows, that there is no uniform approach yet, which overcomes all challenges, which are concretized in Section 1.2 ☞ 31.

Example for bypassing Inconsistency

Lettner, Tschernuth and Mayrhofer (2011, p. 236) report on an example, what can happen, if consistency between views for different stakeholders is not ensured in a structured way: Users started to split the model into parts and worked only on their parts, in order to prevent the occurrence of inconsistency, and put the changed parts together afterwards. Here, inappropriate granularity or structuring of the complete model are identified to be problematic for splitting the model, while the problem behind the problem is missing support for consistency.

The root of this problem, respectively the problem behind this problem, i. e. inconsis-

tencies between multiple views, are conflicts of involved principles: A holistic description of the system under development describes it completely (principle of completeness), but becomes usually too big and too complex for single stakeholders. Therefore, views follow the principle of separation of concerns (Tarr, Ossher et al., 1999) (and the related principle of decomposition) to manage the complexity and size of the system. Since multiple views of the same system have overlaps, they introduce redundancy between the views, which hurts the principle of redundancy reduction.

conflicting Principles: Separation of Concerns vs Redundancy Reduction

In essence, most existing tools including UML tools work on and with single views, as if the represented information is independent first-order information. But in fact, this information describes only a sub system of the whole system, i. e. a view on the system, which must be kept consistent to all other views and the underlying system. This issue is an important criteria in the classification of viewpoints by Darke and Shanks (1996), already few years after the introduction of viewpoints by Finkelstein, Kramer and Goedicke (1990).

Summarizing, on the one hand, multi-perspective modeling fulfills the need of *separation of different concerns* in system development *by providing multiple views*. On the other hand, *multiple views introduce danger of inconsistencies between them*, since they describe together the whole system under development and therefore depend on each other. This problem behind the problem is inherent and is not solved by this thesis. Instead, this thesis ensures inter-model consistency in a structured way to cope with this conflict of paradigms. The next Section 1.2 identifies the challenges which must be overcome in order to fix these problems.

multiple Views enable Separation of Concerns, but introduce Consistency Challenges

1.2 Challenges

In order to solve the general problem, i. e. inconsistencies between multiple views, some challenges must be overcome to realize multi-perspective modeling (Section 1.1 ☞ 26). The challenges are identified in this section and are derived from the motivation (Chapter 1 ☞ 25) and literature.

Challenges in ensuring multi-view Consistency

Keeping different models consistent to each other is already identified as main problem, therefore, it is concretized as challenge in Section 1.2.1. Since consistency is defined as a relationship between views (cf. Definition 2 ☞ 32), these views are investigated and classified regarding their temporal origin: Views which are already existing before applying an approach for ensuring consistency are called *data sources* and are challenging due to the reuse of already existing information, as investigated in Section 1.2.2 ☞ 36. Views which are derived after applying an approach for ensuring consistency are called *new views* and are challenging due to the supply of information stemming from multiple data sources in an editable way, as investigated in Section 1.2.3 ☞ 39. This classification is complete, since each view is established either before or after introducing an approach for ensuring consistency in the particular project. This distinction was already made by Guerra and de Lara (2006) calling data sources as "system views" and new views as "derived views". Their terms are not used here, since data sources emphasize the already existing information more than system views and new views emphasize their late definition reusing information and providing no new information more than derived views, since data sources can be also seen as derived after their integration into a SUM.

1.2.1 Model Consistency

As motivated in Section 1.1 ☞ 26, the main problem is to ensure consistency between multiple views. This section identifies the main challenges to overcome in order to solve the problem of inconsistencies between views. Before discussing challenges of consistency, the term *consistency* is clarified here and summarized in Definition 2 ☞ 32. Before that, definitions

for consistency from the related work are discussed.

reviewing Definitions for Consistency

Persson, Torngren et al. (2013) define, that views are inconsistent to each other, if there is no system which matches the semantics of all views. While consistency between views is expressed clearly as a question of satisfiability by this definition, the underlying system is a bit vague, since the definition allows, that different systems are described by views at different points in time.

Spanoudakis and Zisman (2001) define in a more formal way, that overlaps of views which are defined as overlaps of interpretations of these views are the source for inconsistencies. Basing on that, an inconsistency arises, if a so-called consistency rule is hurt by the views. This definition makes clear, that consistency depends on the interpretations of stakeholders, which are project-specific. Again, the strong relation of the views to their underlying system is missing in this definition.

Engels, Küster et al. (2001) distinguish between syntactic and semantic consistency: *Syntactic consistency* is given, if a model conforms to its metamodel. This definition is complemented with a confusing example, since it does not target the model-metamodel-relation, but a relation between two models. This understanding of model consistence as conformance with its metamodel goes along with other works like Maro, Steghöfer et al. (2015). *Semantic consistency* includes syntactic consistency and requires, that views semantically correspond with the described system. This definition conforms to the other definitions for consistency from literature, as discussed above. This distinction will be taken up when discussion different kinds of heterogeneity of models in Section 3.1 ☞94.

In similar way, Paige, Brooke and Ostroff (2007) distinguish model conformance (syntactic consistency), i. e. the model conforms to its metamodel, and multi-model consistency (semantic consistency), i. e. the models "*do not contradict each other according to a set of (metalevel) rules*" (Paige, Brooke and Ostroff, 2007).

The IEEE defines consistency as "*[t]he degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component*" (IEEE Standards Board, 1990, p. 21). This definition emphasizes the relation of the parts i. e. views of interest to their system. While the other definitions allow only consistency and inconsistency, this definitions enables more graduations for the degree of (in)consistency.

The SWEBOK defines, that "*[c]onsistency is the degree to which models contain no conflicting requirements, assertions, constraints, functions, or component descriptions*" (Bourque and Fairley, 2014). Again, the strong relation of the models i. e. views to their underlying system is missing in this definition.

Sometimes, other terms are used instead of the term consistency, like "model integrity" (Rose, Kolovos et al., 2010), but consistency emphasizes the semantics more clearly than integrity with its legal connotation. Instead of change propagation for ensuring consistency, Berardinelli, Biffl et al. (2015) use the term "co-evolution", which is not used here, since it is usually used for required adaptations of artifacts depending on changed schemata respectively metamodels, for which Section 6.2.1 ☞193 presents several related approaches.

Definition 2: **Consistency**

One or more views are consistent, if these views describe parts of the same system under development without semantic contradictions within a particular project. All views together describe the system in its entirety.

Contradictions occur, when defined conditions for dependencies between views do not hold for particular views, e. g. when expected overlaps between two views are hurt due to a mismatch of corresponding elements in the views, explicit links are missing or broken due to elements which are deleted in one view only or defined constraints do not hold. These conditions for dependencies between views are clarified by Definition 15 ☞72 and

Definition 16☞75 later on, which concretize the relevant contradictions. Contradictions occur between a view and its system in the first place. In the second place, contradictions between a view and its system transitively lead also to contradictions between that view and all other views.

Contradictions target the *semantics of the involved views* according to Engels, Küster et al. (2001), but syntactical differences of the views, e. g. different metamodels of their viewpoints, must be overcome, too. This definition makes clear, that the consistency is specific and depends on the understandings of the views within the current project, in which the system is developed using multiple views. Therefore, *the challenge is to take the semantics of the involved views into account during the automation of ensuring consistency.* If the views are consistent, they describe the system in a consistent way. If the views are inconsistent, they describe the system in an inconsistent way, meaning that the system in its current state cannot be created respectively used due to the inconsistencies. Since such an inconsistent description of the system does not delete its semantic purpose, fixing the inconsistencies leads to a consistent description of the *same system with the same purpose* again.

Challenge: Consistency depends on Semantics of Views

After defining the term consistency, this section now discusses, how consistency challenges look like, i. e. which kinds of dependencies (as motivated on page 29) are possible between elements of different views. These kinds must be targeted by approaches for ensuring consistency, which is the first challenge.

Kinds of Dependencies:

Redundancies describe information which is contained in multiple views. Changing redundant information in one view requires to change the other views accordingly.

redundant Information in multiple Views

Explicit links depict strong and explicit relations between information of different views. Kuhn, Murphy and Thompson (2012) studied a large automotive company and found, that explicit links for traceability are strongly needed, but insufficiently supported.

explicit Links between Views

Constraints describe additional rules which have to hold between information of different views, independently from redundant information and explicit links. Such conventions might be documented explicitly as guidelines for the current project or company or represent implicit best-practices.

Constraints between different Views

Comparing with a classification of Persson, Torngren et al. (2013), these kinds are complete: They call redundancies as semantic overlap instead. They call explicit links as associations instead, but think of an additional explicit "association view" linking elements of two other views explicitly. Here, explicit links subsumes also links between views, which are not made explicit, but should be explicit. They treat constraints as special kind of associations between two views. Additionally, Persson, Torngren et al. (2013) define the categories semantic equivalence and refinement/abstract between two views. Since they describe semantic overlap of complete views, these two categories are subsumed under redundancies here. Syntantic overlap of two views as last category is not relevant here, since combination of concrete syntax is out of scope of this thesis. Following Persson, Torngren et al. (2013), the three listed kinds of dependencies are complete. In contrast to Persson, Torngren et al. (2013), this classification targets single elements of views, which allows to have multiple dependencies with different kinds of dependencies between two views.

Another possible kind of dependency could be instance-of, i. e. elements in the first view are instances of types which are defined in the second view. In other words, the first view describes a model and the second view describes the metamodel of that model. This refers to multi-level modeling and is out of the scope of this thesis. Generalizations between elements of views are not possible, while generalizations between elements of viewpoints are possible and might be used for combining viewpoints.

Summarizing, *the challenge is to cover the heterogeneity of possible kinds of dependencies*, i. e. redundant information in multiple views, explicit links between views and

Challenge: support different Kinds of Dependencies

additional constraints between information of different views. The ongoing example has some examples for consistency challenges, which can be classified according the presented kinds of dependencies:

Ongoing Example, Part 3: **Consistency Challenges** ← List →

Consistency Challenges:

Since the three views for requirements, class diagram and source code together describe the same system under development in entirety, there are *consistency challenges* between these three view due to their contentwise overlaps:

Traceability between Requirements and Methods

1. Since programmers develop the Java source code to fulfill the requirements, there are interrelations between the requirements and the source code. Making these interrelations explicitly, allows to navigate from the requirements to their realizing methods in the Java source code and to trace methods back to their motivating requirements. Therefore, the traceability between requirements and their fulfilling methods should be kept consistent here.

   As an example for possible inconsistencies, the programmer realized the requirement `r1` (“`The student must be able to register for an event.`”) in the method “register” of the class “Student”. Therefore, a traceability link between this requirement and this method must be created. To keep the example simple, traceability links are manually created, e. g. by the programmer.

   This consistency challenge is chosen as an example for different concepts in different views which are interrelated by *explicit links* as kind of dependency: Usually, these links are only implicitly existing in practice, but should be explicitly maintained. The simplest way to enable explicit links between views is to connect their viewpoints by associations. Of course, such explicit relations can be modeled differently, e. g. by annotating links with additional information or by typifying them. An additional challenge is, that these traceability links represent additional content which is contained in none of the existing views. This consistency challenge is managed manually, since interactions with users are usually required, as demonstrated by Gorp, Altheide and Janssens (2006) in general and reported for this case by Becker, Herold et al. (2007, p. 288).

Classes in UML $\subseteq$ Classes in Java

2. Class diagrams and Java have overlaps in form of the redundant description of classes. Same classes in class diagrams and Java can be identified by matching names. All classes must be represented always in Java source code, but not necessarily in the UML class diagram. Note that although the concept of classes is existing redundantly, the features of the classes are different in the two viewpoints of this restricted example, since associations are only described in class diagrams and methods are only part of the Java source code in this restricted example.

   As an example for possible inconsistencies, the architect decided to rename the class “Student” to “Person” in the class diagram. Now the source code is inconsistent, since the corresponding class has still the name “Student”. To fix this inconsistency, the class “Student” in the source code must be renamed to “Person”, too. This fix can be automated.

   This consistency challenge is chosen as an example for *redundancies* as kind of dependency, while the concepts of classes are completely redundant, but the amount of concrete classes is overlapping, but is not equal in Java and UML. Another challenge is to deal with different properties of the redundant concepts (associations only in UML, methods only in Java). This consistency challenge can be managed automatically.

3. Usually, associations in UML class diagrams are implemented as attributes in Java source code. Since these attributes are private due to the paradigms of object-orientation, public methods are required which provide the values for the attributes. Therefore to be usable, each UML association should have a Java getter-method.

   Getters for Associations

   As an example for possible inconsistencies, the software architect just adds the association "university" in the "Student" class in the class diagram. Since the "Student" class in the source code has no getter method called "getUniversity", there occurred an inconsistency. This inconsistency can be automatically fixed by creating such a method in the source code.

   This consistency challenge is chosen as an example for two concepts in two different viewpoints, which have no direct overlap, but are related to each other by *constraints* as kind of dependency. Compared to traditional software development, the relation between an association and a method playing a special role for that association is not made explicit and exists only implicitly by convention. This consistency challenge can be managed automatically. To keep the example short, corresponding setters are not required.

Up to now, these consistency challenges are tried to fix manually in the ongoing development project. Since some of these consistency challenges are automatable, an automated solution is desired. Part 4 of the ongoing example discusses some possible alternatives for these consistency challenges.

Usually, consistency targets only some elements of a view, not the whole view. While consistency establishes strong conditions for these elements, the other elements are not related and therefore do not depend on elements of other views. When ensuring consistency of depending elements, *the challenge is to keep these non-depending elements unchanged.* In particular, it must be ensured, that these elements do not get lost when using model transformations (Section 2.2.3☞67) for ensuring consistency, since simple model transformations can generate only those parts of the target view which are somehow encoded in the source view. In the ongoing restricted example, classes are contained in class diagrams and source code and will be kept consistent, but their associations are contained only in class diagrams, while their methods are contained only in the source code.

Challenge: keep non-depending Content

Another challenge is, that the concrete consistency challenges are specific for the current project: Depending on the currently used tools, the project settings and the involved stakeholders, the consistency challenges can be different. While the general traceability between requirements and Java source code is natural, its granularity must be specified, e. g. if requirements are linked to classes or to methods or even to single statements. Spanoudakis and Zisman (2001) advocate a clear policy how to manage inconsistencies, which depends also on the team members. Usually, such specifications are defined in a project manual to be clear and binding for all involved stakeholders. Already with their definition of consistency, Spanoudakis and Zisman (2001) make clear, that consistency depends on the interpretations of models by stakeholders regarding a particular setting. Lucas, Molina and Toval (2009, p. 1639) support the need for configurable consistency challenges. Even the desired consistency in the ongoing example is project-specific, as indicated by demonstrating some alternatives for the desired consistency:

Challenge: Consistency is project-specific

Ongoing Example, Part 4: **Alternative Consistency Challenges** ← List →

Part 3☞34 of the ongoing example introduced consistency challenges for a simplified software development project. While those consistency specifications are usable for that project setting, it is possible to sketch some alternatives for the desired consistency for another

project setting:

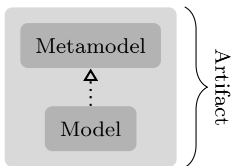
1. Instead of specifying, that the classes in UML are a subset of the classes in Java, all classes could be shown always in Java and UML. With this alternative, all classes can be seen completely in UML, which helps (only) in small development projects. Another alternative is to allow classes in UML which are not part of Java, which helps e. g. for designing a conceptual data model.

2. Instead of manually maintaining traceability links between requirements and Java, traceability links could be automatically created by using a (very simple) heuristic: If the name of a method is contained in the text of a requirements, that method must be linked with this requirement.

Summarizing, even a strongly simplified setting for software development allows different specifications of the desired consistency, depending on the current project, on company guidelines and on other concerns of stakeholders. Therefore, approaches for ensuring consistency must support project-specific consistency challenges.

After identifying challenges for the consistency of interrelated models in this section, the following two sections (Section 1.2.2 and Section 1.2.3☞39) discuss the temporal origins of these models and their roles in multi-perspective modeling. All these challenges are summarized and condensed into the objectives of this thesis in Section 1.3.1☞42.

1.2.2 Reuse existing Artifacts



This section investigates views and their viewpoints which already exist before applying an approach for ensuring consistency in the particular project. Therefore, these views and their viewpoints must be reused by such approaches, which raises challenges as discussed below. These views respectively viewpoints introduce new content respectively new concepts, which must be kept consistent. Reuse as important principle of software engineering is done here "as the view(point)s are" without variation and customization in the sense of Kienzle, Mussbacher et al. (2016).

Since the term "artifact" allows different interpretations (Méndez Fernández, Penzenstadler et al., 2010), it is introduced for the software engineering area with the following reused Definition 3:

> Definition 3: **Artifact**
>
> "*An artefact is a self-contained work result, having a context-specific purpose and constituting a physical representation, a syntactic structure and a semantic content, forming three levels of perception.*" (Méndez Fernández, Böhm et al., 2019)

Artifact described by Model and its Metamodel

While the users of artifacts interact with their physical representations, their syntactic structures enable the automatic processing of artifacts. Here, these syntactic structures are realized with metamodels, since an artifact should be described by an explicit metamodel (schema) and an explicit model (instance) conforming to the metamodel, even when the schema is only implicitly defined by the artifact (Jin, Cordy and Dean, 2002). This model reflects the content of the artifact and is used for automatic processing instead of using the original physical representation. Pfeiffer and Wasowski (2013, p. 386) use the term "mogram" instead, including, among others, models, source code and configuration files. Summarizing, existing artifacts provide metamodels and conforming models, which can be reused in order to avoid to recreate them.

The viewpoints used by stakeholders during system development are realized by tools

or files in fixed file formats, as motivated in Section 1.1☞28. Usually, existing tools for development in a project are initially identified, selected and used, instead of the creation of new tools. The tool selection depends on, among others, already existing tools, company guidelines, experience with tools, guidelines of the customers and further concerns of the stakeholders. Since such tools and file formats are predefined and should not be changed in the project, since the tools usually do not support the evolution of their expected data structures, they are fixed and must be covered by approaches for ensuring consistency "as they are". The existing viewpoints to be reused might be different, even if the same information is encoded, depending on different versions of tools or file formats, e. g. xlsx vs xls for EXCEL. Findings from practice show, that long support for modeling languages is very important for industry, so existing languages must be reused (Briand, Falessi et al., 2012; Whittle, Hutchinson et al., 2013). *The challenge is to reuse existing viewpoints.*

existing Viewpoints are project-specific

Challenge: reuse existing Viewpoints

Nowadays, new systems are not developed from scratch on the greenfield, but in interplay with existing systems and environments, leading to initial information for the system architecture. Early prototypes for requirements elicitation or technical feasibility studies lead to initial implementations. Established knowledge in the company can be reused in form of libraries or reference models leading to some initial data to be reused. Therefore, approaches for ensuring consistency need to deal with already existing models. This includes importing existing models into such approaches *and* exporting them again to keep the initial models up-to-date and conforming to the fixed viewpoints. This is summarized as *challenge to reuse existing views as they are.* Moreno and Vallecillo (2004) identified the reuse of existing systems as an important challenge in modeling, since there exist already developed COTS components, which should be reused for developing a new system or there exist legacy code, which should be maintained. Bucchiarone, Cabot et al. (2020) emphasize the need for ongoing co-existence of legacy models, since they have been developed over decades and should continue to be used.

Challenge: reuse existing Views

Before using an approach for automatically ensuring consistency, existing views might conform to their viewpoints, but are not necessarily consistent to the other existing views, since they were managed without help of such automated approaches. Reusing such views requires to fix them to be consistent to the other views. Therefore, approaches for ensuring consistency are faced with *the challenge to fix possible inconsistencies in the initial models.*

Challenge: fix existing Views

Predefined viewpoints and their already existing views are subsumed as *data sources*, as defined in the following Definition 4. A concrete data source is rendered as DataSource in this thesis.

Definition 4: **Data Source**

A data source incorporates one view and its viewpoint which both exist before starting to apply an approach for ensuring consistency. Data sources represent input for such approaches.

This definition is applied to the ongoing example in order to emphasize the data sources to reuse:

Ongoing Example, Part 5: **Data Sources** ← List →

The ongoing example comes with already existing data sources for requirements, Java and UML, as introduced in Part 2☞25 of the ongoing example. This box emphasizes the existence of data sources with metamodels, models and concrete syntax. This part of the ongoing example describes the involved data sources in more detail regarding their concepts in general (the viewpoint as itemize list), the already developed parts for the university information system in particular and their concrete syntaxes (the view as figure).

textual Requirements Viewpoint

Requirements are elicited by requirements engineers in CSV files focusing on the functionalities of the project to realize. To keep the example simple, textual requirements (instead of user stories) are used with the following features:

- A *requirements specification* contains multiple *requirements.*
- All *requirements* have a unique *identifier*, a *text* representing the content of the requirement in one sentence, and the *author* of the requirement.
- The requirements specification is written in form of a CSV table.

textual Requirements View

For the university information system, there are already two requirements, as depicted in Figure 1.1, referring to two functionalities for students, i. e. the enrollment at the university and the registration for events like lectures. The requirements are stored in CSV-formatted files. This CSV format is supported directly, as shown later in Part 24☞276 of the ongoing example.

Table 1.1: The initial input of Requirements in CSV format

#	ID	Author	Text
1	r1	Andreas Winter	The student must be able to register for an event.
2	r2	Johannes Meier	The student must be enroled at the university.

Java Source Code Viewpoint

Java source code is written by programmers in their IDE to fulfill the requirements according to the design in form of class diagrams. Java source code is strongly simplified with the following features:

- The *Java source code* consists of multiple classes, without packages.
- *Classes* are described by their *names.*
- Classes contain *methods* identified by their *names*, without parameters.
- Methods know which methods they *call* and by which methods they are *called.*

Java Source Code View

For the university information system, there is already some source code, written in Java, depicted in Figure 1.1☞39. It contains the two classes “University” and “Student”. The Java source code is developed using the Eclipse IDE. Since the figure uses a screenshot of Eclipse, it shows some more aspects of Java, which are ignored in this strongly reduced ongoing example.

Class Diagram Viewpoint

UML class diagrams are developed by software architects using modeling tools for important parts of the architecture and data models to provide a solid foundation for the system under development. Class diagrams are simplified with the following features:

- A *class diagram* contains multiple classes.
- *Classes* are described by their *names.*
- Classes contain unidirectional *associations* having a *name*, *lower bound*, *upper bound* and a class as *type.*

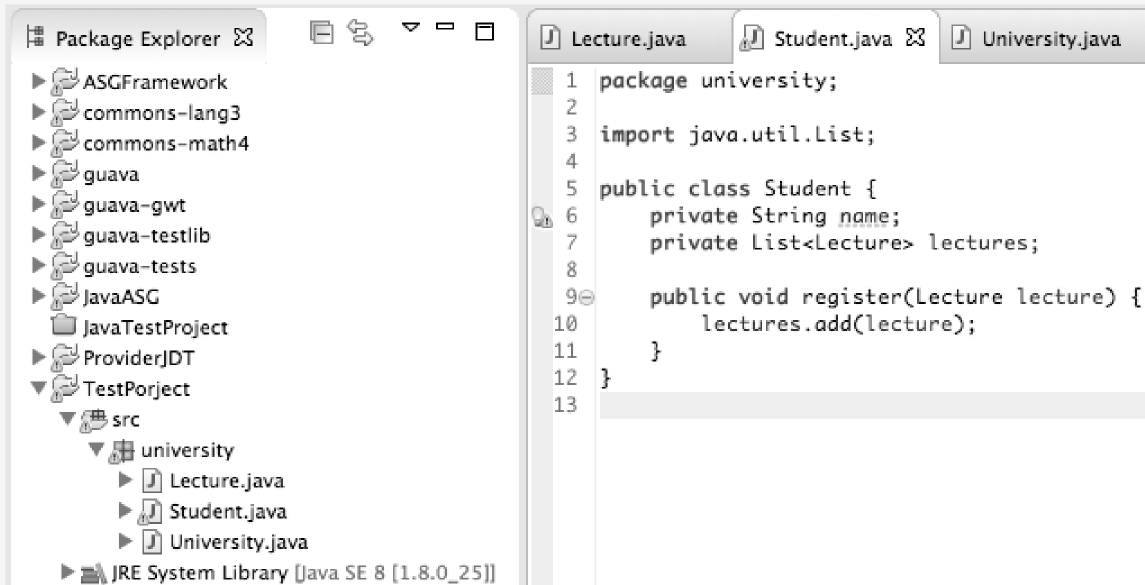


Figure 1.1: Java source code of the ongoing example

Class Diagram View

For the university information system, there is already a first design for the system, in form of a class diagram, as depicted in Figure 1.2. It contains only one class named "University".

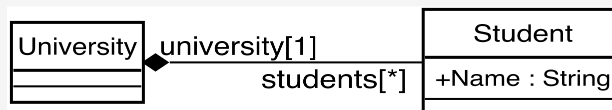


Figure 1.2: Class diagram of the ongoing example

The amount of concepts contained in the three viewpoints is chosen to cover the consistency challenges described in Part 3☞34 of the ongoing example, to be as small as possible and to represent a practical application from the software engineering domain. The workshop series for consistency challenges in software engineering with UML (Huzar, Kuzniarz et al., 2005) shows the relevance of ensuring consistency for UML models and therefore motivates the use of UML in the ongoing example.

The class diagrams for the corresponding viewpoints behind these visualizations of the data to reuse are shown later in following boxes, e. g. in Part 9☞64 of the ongoing example. This counts also for the views rendered as object diagrams.

1.2.3 Define new View(point)s

This section investigates views and their viewpoints which are derived after applying an approach for ensuring consistency in the particular project. Therefore, such approaches must provide techniques to define these views and their viewpoints, which raises challenges as discussed below. These views respectively viewpoints do not introduce new content respectively new concepts, but structure already existing content respectively concepts in a different way.

new View(point)s represent existing Information in new i. e. different Ways

Therefore, one overall challenge in multi-perspective modeling is to introduce new viewpoints to support additional stakeholders with information about the system under development targeting their concerns. New viewpoints reduce the big amount of information

about the whole systems and provide only that information in form of a new view which is needed by the stakeholders.

Newly created viewpoints and conforming views are called *new view(point)s*, as defined in the following Definition 5. A concrete new view(point) is rendered as NewView(Point) in this thesis.

Definition 5: New View(Point)

A new view(point) incorporates a view and its viewpoint which are introduced newly during the application of an approach for ensuring consistency. New view(point)s represent already existing information in a different way.

new View(point)s are project-specific

In contrast to data sources, new viewpoints represent concepts already encoded in the systems description or the data sources, but are tailored and restructured according to the needs of other stakeholders. Since the reused data sources are project-specific (see Section 1.2.2 ☞36), new views and their new viewpoints are project-specific, too. Additionally, their purposes and therefore their designs are depending on the stakeholders and their concerns of the current project.

Other related work also emphasizes the need for defining new view(point)s: France and Rumpe (2007) require new views to be customized to better understand single parts of the system on the one hand and their interactions on the other hand, supported by different kinds of new views in order to reduce complexity and to cover different levels of abstraction. Persson, Torngren et al. (2013) classified the inclusion of new views as one of the main challenges in multi-view modeling under the term extendability, but without further motivations.

Challenge: restructure Information stemming from *multiple* Data Sources

The reuse of already existing information from views of data sources includes also information covering multiple views and information "between" views, e. g. explicit links between views (see page 33). *The challenge is to collect information from multiple views and to provide it in a uniform and restructured way.*

Challenge: keep new Views consistent

Another challenge is to keep such new views consistent to the already existing ones, since new views contain information stemming from other views by definition. Therefore, changes in the other views must be propagated also into new views, according to Section 1.2.1 ☞31.

Challenge: enable *editable* new Views

The most important challenge is to support *editable* new views: A read-only mode for new views is helpful for reading information, in particular for getting an high-level overview with aggregated values. But changing information shown in new views is very important to enable stakeholders with concerns which are not supported by the existing data sources to work actively on the current information. Challenging is the propagation of changes in new views back into all other views due to the view-update problem (Bancilhon and Spyratos, 1981), for which not always a solution exists or the solution is not always unique (Dayal and Bernstein, 1982; Reder and Egyed, 2012). More related work in this area is discussed in Chapter 3 ☞93. Summarizing, *the challenge is to enable editable new views by propagating changes in these views back into the other views.*

Ongoing Example, Part 6: New View(Point)s ← List →

In the project of the ongoing example, a project manager wants to see the progress of the project. More concrete, the project manager is interested, if requirements are already realized in Java source code. Such mapping between requirements and Java will be realized as shown in Figure 1.3 ☞41.

The requirements are shown with their ID (first column) and text (second column). The third column lists all methods in the Java source code (separated by comma, prefixed by their class names), which realize the corresponding requirements (it is empty, since the

data sources do not contain such traceability links). Since the project manager does not want to deal with technical documents, this new view will be realized as EXCEL sheet and not as CSV file.

But the project manager wants not only to see the current realization of requirements, but also to manage them: In particular, the project manager wants to add Java methods, which were added by programmers in the mean time to fulfill the requirements. Removing methods should be possible, too, e. g. after new test cases showed, that the implementation still contains errors.

ID	Text	Fulfilling Methods
r1	The student must be able to register for an event.	
r2	The student must be enroled at the university.	

Figure 1.3: The final concrete syntax of Traceability in Excel format

It is not possible to create or to delete methods, only their mapping to requirements can be adjusted. Renaming methods or classes is impossible, too. If the project manager uses methods which do not exist, they will be removed from the new view. Additionally, it is possible to change the text of the shown requirements.

This example shows the need to define new view(point)s containing information stemming from multiple data sources (here: requirements and Java), in a different structure as in their original data source (here: qualified method names instead of mesh of objects) and in an editable way.

These three challenges, i. e. consistency between multiple views (Section 1.2.1[☞ 31]), which might already exist (Section 1.2.2[☞ 36]) or are newly derived (Section 1.2.3[☞ 39]), are the foundation for the objectives of this thesis, as clarified in the next Section 1.3.

1.3 Aims

After establishing the problem of inconsistencies between multiple views in Section 1.1[☞ 26] and identifying related major challenges in Section 1.2[☞ 31], the main objectives to overcome these problems and challenges are clarified in Section 1.3.1[☞ 42], forming first high-level requirements for a corresponding approach in Section 1.3.3[☞ 46]. As demarcation, Sec-

tion 1.3.2☞43 presents other challenges and related research areas which are no objectives of this thesis.

1.3.1 Objectives

The main challenge is to ensure consistency between multiple views, as discussed in Section 1.2.1☞31. As preparation for the objectives of this thesis to be defined in this section, the term *consistency* as summarized in Definition 2☞32 is concretized here: This thesis distinguishes consistency into intra-model consistency in Definition 6 and inter-model consistency in Definition 7, depending on the number of involved models. The models describe the information of relevant views which represent parts of the system under development:

intra vs inter-model Consistency

Definition 6: Intra-Model Consistency

One model is consistent, if this model describes one system without semantic contradictions within a particular project.

intra: #Models = 1

Intra-model consistency targets the consistency of a single model and requires, that this model describes a system without (internal) contradictions. Intra-model consistency considers only one view and its representation of a system. This system might be part of a bigger system, which is represented by some more views, which leads to inter-model consistency, while intra-model consistency is restricted to the (internal) consistency of one view. An example for intra-model consistency in Java is the specification, that all methods within the same class must be unique regarding their names in the ongoing example (and the types of their parameters in complete Java), as discussed in Part 28☞480 of the ongoing example.

According to Definition 6, Definition 7 defines:

Definition 7: Inter-Model Consistency

Two or more models are consistent to each other, if these models describe parts of the same system without semantic contradictions within a particular project. All models realize views which together describe the whole system. Each model must fulfill intra-model consistency.

inter: #Models ≥ 2

Inter-model consistency focuses on the relations between multiple models and requires, that the models do not contradict each other. If some of these models do not fulfill intra-model consistency, they introduce inconsistencies into the whole system. Therefore, all involved models must also fulfill intra-model consistency as precondition. All presented consistency issues in the ongoing example target inter-model consistency.

The terms intra and inter-model consistency are not explicitly used by Persson, Torngren et al. (2013), but they define the consistency between views only for views which are consistent internally, which corresponds to intra and inter-model consistency. Egyed, Zeman et al. (2018) use the corresponding terms "intratool consistency" and "intertool consistency" from a tooling point of view. Goldschmidt, Becker and Burger (2012) defined "intra view overlap" and "inter view overlap" comparable as the multiple occurrence of same elements within one view respectively two or more views.

alternative Definitions from Related Work

In contrast to these definitions, Huzar, Kuzniarz et al. (2005) define *intra-model and inter-model consistency* regarding the level of abstraction of the involved models: Intra-model consistency targets models with same abstraction level, while inter-model consistency targets models with different abstract levels. This distinction is not needed here, since this approach aims to keep arbitrary models (i. e. with same or different abstraction levels) consistent. With requirements and source code, the ongoing example directly targets

different abstraction levels. Section 13.4.2☞479 discusses different levels of abstractions in general.

Broy, Feilkas et al. (2010) distinguish between *vertical and horizontal consistency*, targeting consistency issues in models of the same (horizontal) or of different (vertical) development phases like requirements elicitation and implementation. This distinction is not necessary here, since this approach aims to keep models consistent independently of their number, order or size of changes, as shown by the ongoing example targeting requirements and source code.

Engels, Küster et al. (2001) do not clearly distinct the level of abstraction and the development phase from each other, when using the terms horizontal and vertical consistency. Usually, early development phases correspond with views of higher abstraction (e.g. requirements), while subsequent development phases correspond with views of lower abstraction (e.g. source code). But this is not always true, e.g. for prototypes in early design space explorations.

Lucas, Molina and Toval (2009) reuse the definitions for intra/inter-model consistency from Huzar, Kuzniarz et al. (2005) and syntactic/semantic consistency from Engels, Küster et al. (2001). They complement them by examples, which seem to intermix syntactic/semantic consistency as defined by Engels, Küster et al. (2001) with consistency between views for static and dynamic aspects of the system. This Definition 2☞32 does not distinguish between consistency of models describing static aspects of the system and models describing dynamic aspects of the same system, since both kinds are targeted.

Regarding versioning of models, Van Der Straeten, Mens et al. (2003) distinguish *horizontal consistency* targeting different models at the same version and *evolution consistency* targeting one model at different versions. *Vertical consistency* between a model and its refining model is mentioned, too, but not deepened. This thesis focuses on the consistency between models at the same version by fixing occurred inconsistencies. Of course, fixing an inconsistent model leads to a new version of this model.

These clarifications of the terminology for consistency help to state the objectives of this thesis:

Objectives

The main objectives of this thesis are development and evaluation of a new approach to ensure *consistency* between interrelated models (inter-model consistency) *automatically*. The involved models can be *reused* from ongoing activities or derived as *new views*.

This thesis is focusing on inter-model consistency and assumes as precondition, that the data sources to manage are already consistent in itself. This allows to focus on consistency issues between different models, without being contradicted by intra-model inconsistency. Nonetheless, Section 13.4.5☞480 shows, that the approach of this thesis can also fulfill intra-model consistency.

Usually, the interrelated models are conforming to *different* metamodels, since they describe different parts of the system under development. Therefore, changes in one model have to be synchronized into all related models while taking different metamodels into account. Later, this is realized by using an explicit Single Underlying Model (SUM), as introduced and discussed in the related work (Section 3.4☞120).

1.3.2 Demarcation

This section makes explicit, which aspects are *not* objective of this thesis, and demarcates it from other fields of related research. This section thereby emphasizes the main objective of this thesis.

fix Inconsistencies specifically for the Project

Here, it is not sufficient to detect inconsistencies, which is called "impact analysis" by Persson, Torngren et al. (2013). Objective is to fix inconsistencies, which requires changing actions. While manual fixes are senseful in some cases (Gorp, Altheide and Janssens, 2006), the approach of this thesis focuses on automated fixes for inconsistencies. These fixes must fulfill the needs of the particular project setting, rejecting approaches which determine fixes in non-deterministic way like graph repair (Sandmann and Habel, 2019).

Dimensions of Integration

This thesis focuses on *data integration* in terms of Wasserman (1990) only. With the background of tool integration, Wasserman (1990) classifies five dimensions of integration:

Platform Integration covers network and operating system services as provided by middleware or operating systems. While software, tools, and services are not in the focus of this thesis, the Eclipse Modeling Framework (EMF) as central data format (Section 2.5.3☞87) could be seen as precondition for integration in this category.

Presentation Integration covers unique interfaces for the users of tools, e. g. graphical user interfaces. Focusing on the pure data, their (re)presentations are not relevant in this thesis and can be realized in different ways (Section 6.6.5☞226), i. e. the concrete syntax does not matter and can be realized with any approach, e. g. with XTEXT (Chapter 9☞283) or EXCEL.

concrete Syntax does not matter

Data Integration covers exchange and sharing of data used by multiple tools. This is highly relevant for this thesis, since different tools using different data lead to the need for a management of these data in terms of consistency, as motivated in Section 1.1☞26.

Control Integration covers communication between tools in order to send notifications from one tool to other tools about occurred events. Since the interaction between the different views can be summarized as "changes inside one view must be propagated into all other views", only one trigger like "view is changed" is required to start the process of change propagation. As long as this trigger is existing, a general solution for arbitrary communication between tools or views is not required here.

Process Integration covers the integration of tools by a superordinate process management. Since the change propagation is designed to run automatically without user interaction, it can be realized within one process and can be easily integrated into other process managers or tools.

Tool vs Data Integration

While this distinction shows, that tool integration is a multi-dimensional challenge in general, this thesis focuses on the data dimension: The main challenge of change propagation, identified as objective of this thesis, falls in the category of data integration. The survey of Asplund and Törngren (2015) shows, that the dimension of data integration is the most important aspect, together with control integration. In this thesis, the term *tool integration* subsumes platform, presentation, control and process integration. Additionally, the distinction above indicates, that the aspects for tool integration do not introduce additional serious challenges when overcoming the challenge of data integration.

ensure tool-independent Data Consistency

This thesis aims for integration regarding data and not for integration regarding tools, as there are other approaches for tool integration like SENSEI (Jelschen, 2015). Consequently, the change propagation is not realized in form of another more or less enclosed tool like JETBRAINS MPS (Section 3.6.2☞137), but in a stand-alone library for ensuring tool-independent data consistency. Thomas and Nejmeh (1992) summarize very accurate:

> "*The goal is to maintain consistent information, regardless of how parts of it are operated on and transformed by tools.*" (Thomas and Nejmeh, 1992)

But an integrated data structure can be used as data structure for an integrated tooling (Wasserman, 1990). Therefore, solving the challenge of data integration provides foundations for realizing tool integration.

As already motivated earlier, this thesis concentrates on data described by an explicit schema (Jin, Cordy and Dean, 2002). The data are instances conforming to one schema. This thesis narrows the data to one instance-of level, while approaches for multi-level modeling with two or more instance-of levels are not supported. Later, data and their schema are realized by a model and its metamodel, as concretized in Section 2.2 ☞58.

project-specific Solutions

Since the consistency is project-specific (see page 35), this thesis does not provide a generic integration which is valid for each application domain, but enables a specific integration for each particular system, developed in a particular project by a particular company with a particular team. This counts not only for the consistency rules, but also for the involved models and metamodels.

Software Development

This thesis will not solve all problems of software development: Software development is used only for demonstration with the small, ongoing example. Applications in Part IV ☞283 outside of software development show, that the new approach solves data consistency in various domains, not only software development.

Types of Collaboration

This thesis does not focus on enabling collaboration between multiple users on the same data *in real-time*, but enables stakeholders to work independently from other data on their individual data subsets, which are kept consistent to the data of the other stakeholders. The stakeholder can work in spatial separation, as discussed in Section 13.4.4 ☞480.

Model-based Engineering

This thesis will not solve all problems in model-based engineering (introduced in Section 2.2 ☞58), in particular, the integration of various modeling techniques and their tools is not covered. But with the focus on consistent models managed through multiple views, this thesis realizes a step towards realizing integrated model-based engineering, as motivated in the vision of Broy, Feilkas et al. (2010).

Multi-Paradigm Modeling

A related, higher-level research topic is *multi-paradigm modeling* (MPM) (Mosterman and Vangheluwe, 2004): MPM is motivated by the heterogeneity in the dimensions application domains, viewpoints, development activities (like implementation and verification) and levels of abstraction and aims to overcome the heterogeneity of models corresponding to these four dimensions (Hardebolle and Boulanger, 2009). MPM emphasizes the behavior of systems and development activities like model-based simulations (Vangheluwe, de Lara and Mosterman, 2002). The heterogeneity of viewpoints leads to consistency challenges between views, which is the main motivation of the new approach this thesis. This approach is project-specific (see above) covering the heterogeneity of application domains and supports models with different levels of abstraction (see Section 1.3.1 ☞42 and Section 13.4.2 ☞479). On the one hand, the heterogeneity of development activities requires to integrate different tools, which is not the objective of this thesis (see above). On the other hand, development activities require tailored views, which can be provided by the new approach.

read/write consistent Data in Processors

In the field of distributed systems and shared data storages, the term consistency targets the results of read and write operations by multiple processors on the same data, for which many different consistency models exist (Viotti and Vukolić, 2016). This is not relevant here, since it deals with the consistency of the *same* data for multiple processors, while this thesis ensures consistency of *different* data to each other in terms of contentwise interrelations.

Data Replication

Out of scope is also the concept of *data replication* (Tos, Mokadem et al., 2015), since it duplicates the *same* data for improving non-functional properties like robustness. In contrast to keeping the same data up-to-date at different locations, this thesis keeps different but interrelated data consistent with each other regarding these interrelations. Some insights into data replication are given by Antkiewicz and Czarnecki (2008, p. 40f) from a model synchronization perspective, basing on the survey for replication strategies by Saito and Shapiro (2005).

With a similar motivation, the identification and management of *model clones* (Störrle,

Model Clones

2013) is also out of scope, since clones represent duplicates within models conforming to the same metamodel. Since model clones are finally detected by heuristics, human interaction is required to verify found candidates to be actual clones, which prevents fully-automated approaches as desired here.

Data Migration

Data migration, i. e. the migration of data during migration projects, is also not in the focus of this thesis. A major difference is, that migrations are done once (or a limited number) during the limited duration of the migration project, while data consistency is an ongoing task and is required as long as the involved data are used. Additionally, data migration usually migrates existing data from the old system into the new system, but not vice versa. In order to keep existing data sources up-to-date, fulfilling this challenge of Section 1.2.2☞36 requires to propagate changes back from the new system into the old system, too. Since the new approach supports this as "additional use case", the new approach could be used to realize also data migration, as discussed in Section 14.4.3☞498.

Data Interoperability

Data interoperability, i. e. supplying the same data in different data formats, is also not the motivation for this thesis. Thanks to the adapters (Section 6.6.5☞226), such data conversations are possible to some degree, as discussed in Section 14.4.3☞498, but are not in the foreground of this thesis.

Conformance of Models to their Metamodels

Model Co-Evolution

Model Editors

The *conformance of models to their metamodels* is important in general and also in particular for this thesis. In order to distinguish this relation between models and their metamodels from definitions for syntactical consistency from related work (Section 1.2.1☞31), it is called conformance and not (syntactical) consistency in this thesis. This conformance can be hurt by two cases, i. e. by changing the metamodel or by editing the model: *Changing the metamodel* requires to change existing models and is called model co-evolution (Wachsmuth, 2007). Model co-evolution is neither the main motivation nor the main objective of this thesis, but is an important part of the solution for the chosen approach, since metamodels are changed together with their models (Section 6.2☞192). Therefore, this new approach can be used for realizing model co-evolution, too, as discussed in Section 14.1.2.3☞487. *Editing the model* must comply with the specifications and constraints of the metamodel. Additional semantic constraints represent intra-model consistency. Since this thesis targets the inter-model consistency between multiple models, this case is out of scope. There are other approaches in related work focusing on this case, e. g. Nassar, Radke and Arendt (2017), Barriga, Rutle and Heldal (2019) and Kehrer, Taentzer et al. (2016). Nevertheless, this new approach can realize also this case to some degree, as discussed in Section 13.4.5☞480.

Consistency vs Correctness

This thesis ensures *consistency between different views*, but not the *correctness of single views* or the whole system under development. In particular, information which is wrong or does not make any sense is allowed in views, but this (wrong) information must be identical (i. e. consistent) in all views. Therefore, the approach of this thesis does not locate bugs in models (Arcega, Font et al., 2019), but inconsistencies between models.

legal Issues as Precondition

While this thesis demonstrates, how data can be kept consistent by their integration, there are also legal conditions for managing data: Before integrating data, legal issues and conditions regarding their handling and combination must be checked and fulfilled. This might limit some application scenarios of the presented approach in practice, but is out of scope of this thesis in general.

1.3.3 High-level Requirements

After defining the objective of this thesis in Section 1.3.1☞42 and demarcating it from other aspects in Section 1.3.2☞43, the following high-level requirements are directly derived from the objectives established on page 43:

First high-level Requirements

R 1 Changes in one model have to be propagated into all related models. (**Model Consistency**)

R 2 The approach must allow to reuse existing artifacts. (**Reuse existing Artifacts**)

R 3 The approach must allow to define new view(point)s. (**Define new View(point)s**)

Requirement R 1[☞154] covers the main objective of this thesis, to keep models consistent to each other after occurred changes, as motivated in the challenge Model Consistency (Section 1.2.1[☞31]). Change propagation is the behavior desired by users of models, as deepened in Section 2.3[☞71]. Additionally, change propagation is no concrete realization strategy for ensuring consistency, since Section 3.2[☞99] identifies several classes of concrete realization strategies for change propagation. Requirement R 2[☞155] covers the challenge Reuse existing Artifacts (Section 1.2.2[☞36]) to reuse existing artifacts and keep them consistent, too. While the focus is to reuse existing artifacts and to keep their models consistent to each other, it is also possible to start without any reused artifacts, as discussed in Section 13.3.2[☞474]. Requirement R 3[☞156] covers the challenge Define new View(point)s (Section 1.2.3[☞39]) to define new viewpoints with reuse of existing and consistent information. These high-level requirements will be concretized in following sections, in particular in Section 4.1[☞154].

1.4 Summary & Outline

Summary

This Part I[☞25] identified the preservation of consistency in multi-view environments as important challenge. It forms the main motivation and objectives of this thesis, complemented with the need to support already existing artifacts as data sources and to define new view(point)s. The introduced ongoing example showing a strongly simplified software development project is used during all parts of the thesis for demonstration purpose.

Outline

The most important basic concepts for managing the consistency for models, like terminology for consistency, modeling foundations and technical spaces, are introduced in Chapter 2[☞51]. More basics which are less important are introduced at that location, where they are needed the first time.

The main related work discussing existing approaches for ensuring consistency in multi-view environments is located in Chapter 3[☞93]. More related work discussing alternatives for the current topic is located directly at the location of that topic. This counts in particular for design decisions in Chapter 6[☞185], which are enriched with related approaches. The lessons learned from analyzing related approaches are used to establish requirements for the new approach of this thesis in Chapter 4[☞153]. The requirements take up also the main challenges of model consistency.

The new approach for ensuring consistency between models is designed and realized in Part III[☞163], starting with an overview of the new approach called MoConseMI in Chapter 5[☞163]. The main design decisions are made and discussed in Chapter 6[☞185] to fulfill the requirements. Operators as central part of the developed approach are documented in Chapter 7[☞241]. Chapter 8[☞263] presents the implementation of the designed approach including the operators in form of a reusable framework.

This framework is used in Part IV[☞283] to apply the designed approach in different application domains, including management of access rights (Chapter 9[☞283]), viewpoint-driven architectures for smart environmental information systems (Chapter 10[☞373]), and knowledge management for projects (Chapter 11[☞387]). Best practices for application in

general are derived in Chapter 12☞455 from these concrete applications.

In Part V☞467, these applications provide reliable arguments to evaluate the applicability of MoConseMI and the fulfillment of the requirements (Chapter 13☞467). These results are summarized in Chapter 14☞483 with contributions of this thesis, identified limitations and possible future work.

Deliverables

The described outline aims at developing the targeted *deliverables* of this thesis, mainly a new approach for ensuring inter-model consistency, targeting the main objective of this thesis. This conceptual approach is realized as reusable framework in Java. The framework is applied to multiple application scenarios evaluating the design approach. These deliverables are summarized in the following box:

Deliverables

To fulfill the objectives of this thesis, the following deliverables are developed and documented in this thesis:

1. a new *approach* to ensure the consistency between interrelated models automatically (called MoConseMI)
2. a *framework* realizing this approach (the MoConseMI framework)
3. several *applications* using this framework to evaluate the developed approach

To realize these deliverables, required foundations are introduced in the next Part II☞51. Central design decisions for the new MoConseMI approach are motivated in Chapter 5☞163, while its details are developed in Chapter 6☞185. The supporting MoConseMI framework is implemented in Chapter 8☞263. The applications are documented in Part IV☞283.

Part II

Foundations

This part provides foundations for ensuring inter-model consistency. Basic concepts for views, their realization with models and technical spaces as well as concretized concepts for consistency and their involved stakeholders lay out foundations for the analysis, classification and evaluation of related approaches. Although there are lots of approaches for ensuring inter-model consistency in various research areas with various realization techniques and strategies, there is no approach which fulfills all needs for ensuring inter-model consistency with satisfactory degree, which motivates to develop a new approach for ensuring inter-model consistency in this thesis. The findings from analyzing basic concepts and related approaches are summarized as requirements for this new approach.

Chapter 2

Basic Concepts

Outline of this Section

This section clarifies some terms from the motivation (Chapter 1☞25) and lays out the foundations for the analysis of related work (Chapter 3☞93) and the design of the new approach (Part III☞163). Since the use of multiple views during system development introduces a potential for inconsistencies, Section 2.1☞54 clarifies the understanding of *views* and their *viewpoints*. Since views of interest and their represented information are conceptually realized by *models*, Section 2.2☞58 introduces terminology of model-based engineering.

In order to ensure consistency between views as main objective of this thesis, the understanding of *consistency* is increased in Section 2.3☞71. Since different groups of persons are differently involved in the specification and application of consistency, Section 2.4☞79 introduces four types of *stakeholders* involved in the management of consistency. In order to technically realize models for automation, Section 2.5☞84 motivates the use of EMF as *technical space* for realizing models in this thesis. Section 2.6☞89 summarizes the resulting foundations.

Concretize Requirements

During the presentation and discussion of foundations, the high-level requirements of Section 1.3.3☞46 are concretized respectively complemented by additional requirements, whose need is motivated by currently discussed foundations. All requirements are motivated and collected in Chapter 4☞153 as summary.

Big Picture of Concepts along the Ongoing Example:

In order to clarify the relationships between the concepts used within this section and to show their impact for consistency, these concepts are applied to the ongoing example resulting in a big picture, which shows also the relevance of the terminology. Afterwards, this terminology is introduced in detail in the following parts.

Ongoing Example, Part 7: **Applied Concepts as Megamodel** ← List →

Megamodels formalize Relationships between Models

The concepts and terminology of this section are applied to the ongoing example in Figure 2.1☞52. Since these concepts can be treated as models and in particular views are realized with models (Section 1.1☞26), the idea of megamodeling (Bézivin, Jouault and Valduriez, 2004) is used to analyze possible relationships between different models. A megamodel is a graph, whose nodes represent models and whose edges represent different kinds of relationships between these models. According to the principle of "everything is a model" (Section 2.2☞58), it is possible to treat the depicted elements as models.

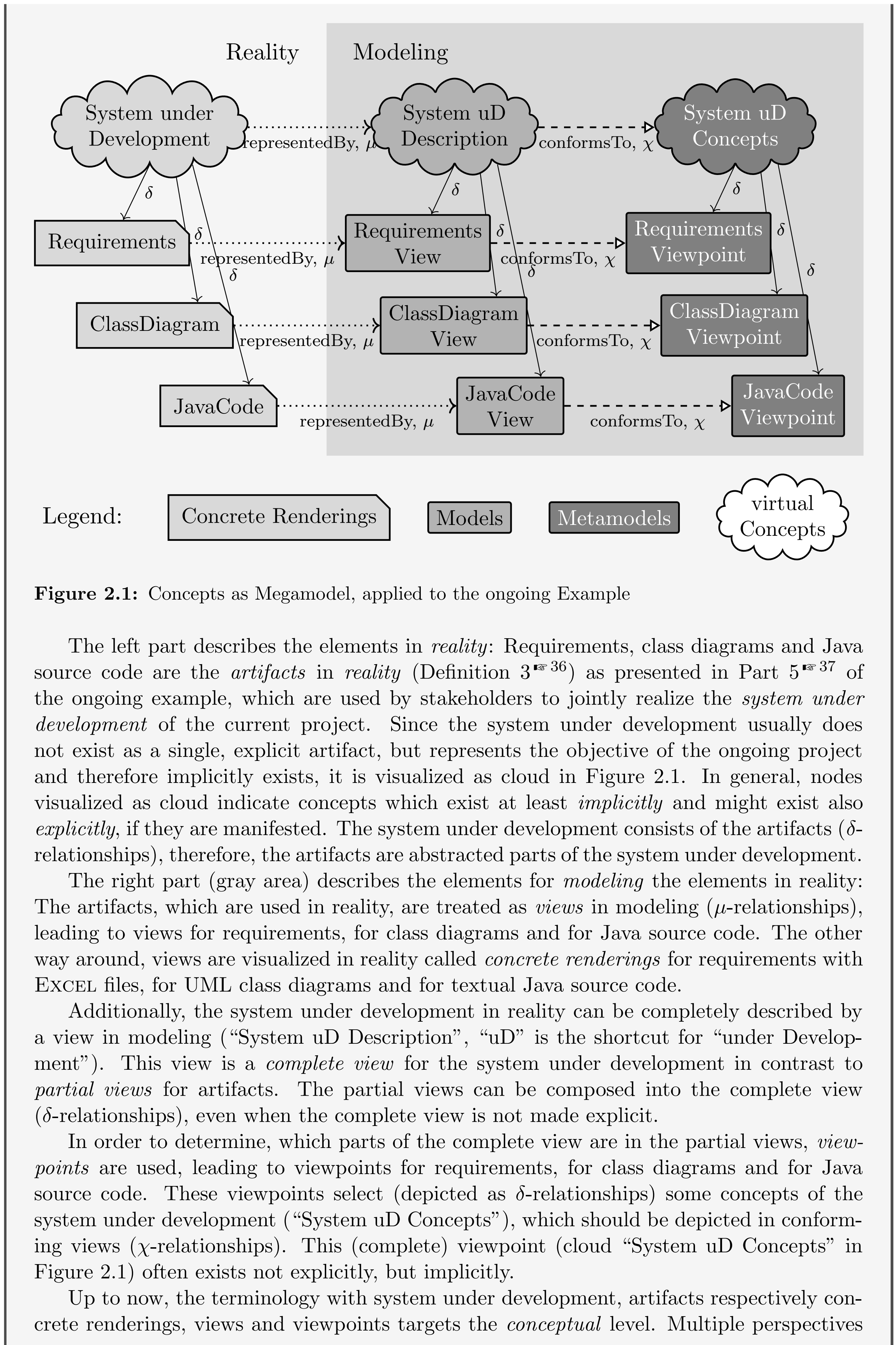


Figure 2.1: Concepts as Megamodel, applied to the ongoing Example

Reality: Artifacts for the System under Development

The left part describes the elements in *reality*: Requirements, class diagrams and Java source code are the *artifacts* in *reality* (Definition 3☞36) as presented in Part 5☞37 of the ongoing example, which are used by stakeholders to jointly realize the *system under development* of the current project. Since the system under development usually does not exist as a single, explicit artifact, but represents the objective of the ongoing project and therefore implicitly exists, it is visualized as cloud in Figure 2.1. In general, nodes visualized as cloud indicate concepts which exist at least *implicitly* and might exist also *explicitly*, if they are manifested. The system under development consists of the artifacts (δ-relationships), therefore, the artifacts are abstracted parts of the system under development.

Modeling: Views represent Artifacts (called Concrete Renderings)

The right part (gray area) describes the elements for *modeling* the elements in reality: The artifacts, which are used in reality, are treated as *views* in modeling (μ-relationships), leading to views for requirements, for class diagrams and for Java source code. The other way around, views are visualized in reality called *concrete renderings* for requirements with Excel files, for UML class diagrams and for textual Java source code.

Complete View consists of Partial Views

Additionally, the system under development in reality can be completely described by a view in modeling ("System uD Description", "uD" is the shortcut for "under Development"). This view is a *complete view* for the system under development in contrast to *partial views* for artifacts. The partial views can be composed into the complete view (δ-relationships), even when the complete view is not made explicit.

Viewpoints determine Content of Views

In order to determine, which parts of the complete view are in the partial views, *viewpoints* are used, leading to viewpoints for requirements, for class diagrams and for Java source code. These viewpoints select (depicted as δ-relationships) some concepts of the system under development ("System uD Concepts"), which should be depicted in conforming views (χ-relationships). This (complete) viewpoint (cloud "System uD Concepts" in Figure 2.1) often exists not explicitly, but implicitly.

Up to now, the terminology with system under development, artifacts respectively concrete renderings, views and viewpoints targets the *conceptual* level. Multiple perspectives

are multiple views conforming to different viewpoints. As the term multi-perspective *modeling* suggests, views are realized by models (Section 1.1 ☞ 26) on more technical level, i. e. the information of the system under development which is selected to be represented by a view is encoded as model. Therefore, the views are depicted as models in Figure 2.1 ☞ 52. Accordingly, viewpoints encode the structure desired for the models of conforming views as metamodels and are depicted as metamodels in Figure 2.1 ☞ 52.

Views encode their Information as Models

Summarizing, a view can be seen as a *mapping* of one system under development and one viewpoint, leading to one model. This model represents those parts of the system under development which are indicated by the viewpoint, and in the structure which is determined by the metamodel of the viewpoint. Since a stakeholder requests a view, but is usually not used to work on the view's model directly, concrete renderings of the model are provided, which are determined by the viewpoint.

Views map one System and one Viewpoint to one Model with Concrete Renderings

All relationships between two nodes as depicted in Figure 2.1 ☞ 52 require synchronization in order to ensure these relationships. Note, that synchronization can follow relationships in transitive way, which is important in particular for δ-relationships, as deepened below. The impact of changes within nodes is analyzed regarding related nodes depending on the kind of relationship:

RepresentedBy μ describes the relationship between models and their systems (according to Definition 12 ☞ 59 following in Section 2.2.1 ☞ 59). Here, views represent parts of the system under development like the concrete requirements, class diagrams and source code.

μ-relationships require synchronization on technical level: Stakeholders usually do not directly work on the models of the views, but work on parts of the represented systems as concrete renderings of the models (Figure 2.22 ☞ 90). Therefore, the information about the system under development is the same, but it is presented with different techniques. This relationship is deepened in Section 2.2.1 ☞ 59.

Synchronize Views with represented (Parts of) Systems regarding technical Presentation

DecomposedIn δ describes the relation between a composite and one of its parts. Applied to Figure 2.1 ☞ 52, there are three cases of δ-relationships: First, the system under development consists of requirements, class diagrams and Java source code in reality. Second, the model-based description of the system under development (complete view) consists of the information of all (partial) views. Third, the concepts for the description of the system under development consist of the concepts of all viewpoints.

Looking at the second case for the involved models, partial views represent some information (as reduction) of the system under development. Therefore, changes in partial views must be transferred also into the description of the system under development (the complete view), otherwise the δ-relationships are hurt. The other way around, changes in the complete view must be reflected also in all its partial views to ensure δ. This relationship is deepened in Section 2.3 ☞ 71.

Keep Complete View as complete System Description and its partial Views consistent to each other

If information of the complete view which is part of multiple, overlapping partial views is changed, these changes affect multiple views: If the information is changed in one partial view, the complete view must be changed accordingly and then transitively also all other partial views containing the changed information, along the δ-relationships in the case of an explicit complete view. In the case of an implicit complete view, changes within one partial view must be *directly* propagated to other partial views.

ConformsTo χ describes the relationship between a viewpoint and its views, i. e. the view conforms to its viewpoint. According to IEEE (2011), each view is governed by exactly one viewpoint.

View conforms to its Viewpoint

Changing views must ensure, that they still fulfill the guidelines of their viewpoints. Changing viewpoints must ensure, that their views are changed accordingly without accidental information loss. As already discussed in Section 1.3.2☞43, the conformance of views (and their models) to their viewpoints (and their metamodels) must be ensured, but must be distinguished from ensuring consistency between models representing views here. This relationship is deepened in Section 2.2.2☞60.

δ between Models target semantic Consistency here

Since the objective of this thesis is the semantic consistency of different models (Section 1.3.1☞42), δ-relationships between partial views and the complete description of their system under development in Figure 2.1☞52 are the most important relationships here. Therefore, this Chapter 2☞51 focuses on semantic consistency between different models, which counts also for the investigated related approaches covering semantics in Chapter 3☞93.

2.1 Views and Viewpoints

Views for multi-perspective Modeling

The objective of using different views is to support the persons who are involved in the development of the system with exactly that information which they require. Since supporting multiple views is an established concept during modeling and development of complex systems (Section 1.1☞26), this section introduces the terminology for views, their viewpoints and involved persons. Additionally, this section is the foundation to investigate possible inconsistencies between views in Section 2.3☞71.

The terminology, clarified in this section, follows mostly the ISO Standard for Architecture Description 42010:2011 (IEEE, 2011). While IEEE (2011) focuses on the architecture domain, its definitions are generalized here to the description of any systems, including architectures. In order to summarize the introduced terms and their relationships, Figure 2.2 represents them as class diagram: Parts of the *system under development* are represented by *views* guided by *viewpoints* which address *concerns* of *stakeholders* interested in the system under development and using these views. This figure is extended in the following sections.

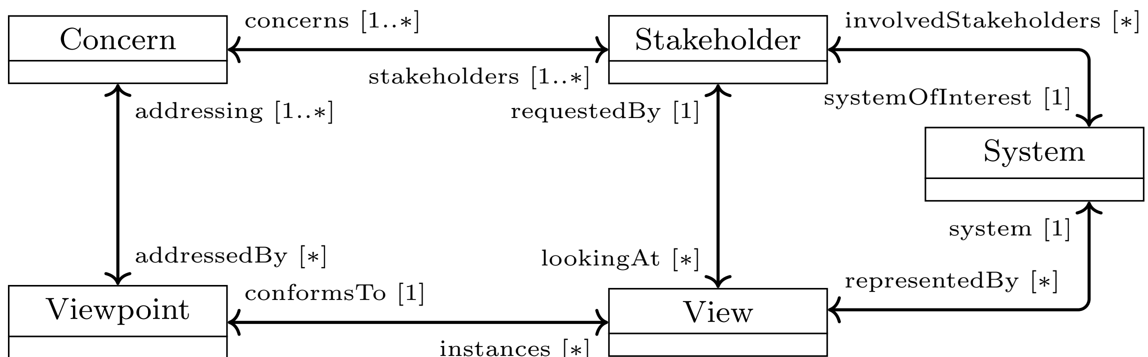


Figure 2.2: Concepts for Stakeholders, Systems and Views

Stakeholders of the System under Development

Persons involved in the system under development are *stakeholders* and introduced in Definition 8☞55:

Definition 8: Stakeholder

A stakeholder is an "*individual, team, organization, or classes thereof, having an interest in a system*" (IEEE, 2011) or is involved somehow in the development of a system.

Possible stakeholders of projects for system development include project manager, customers, developers, requirements engineers, testers, operators and end users, but also governments establishing laws. Stakeholders relevant for the new approach of this thesis as a concrete system are introduced in Section 2.4 ☞79. Note, that persons can act as different stakeholders for the system, i. e. stakeholders can be seen as roles taken up by persons. An interest of stakeholders in the system under development is called *concern*, introduced in Definition 9:

Concerns reflect Interests of Stakeholders

Definition 9: Concern

A concern is the "*interest in a system relevant to one or more of its stakeholders*" (IEEE, 2011).

Possible concerns of stakeholders for software system development include knowing the objectives and requirements of the system, defining the design of the system, checking if and where requirements are implemented in the source code, and testing the developed regarding the requirements.

Perspectives reflect these concerns by defining, which parts of systems are selected, and are named *viewpoint*s in Definition 10:

Definition 10: Viewpoint

A viewpoint defines "*the conventions for the construction, interpretation and use of [parts of systems] to frame specific system concerns*" (IEEE, 2011).

These conventions reflect one or more concerns of stakeholders which are involved in the current system under development. It is important, that viewpoints do not describe actual information of concrete systems, but specify more generically, which information of systems should be focused on and which information should be ignored, depending on the concerns which are addressed by the viewpoint. Therefore, viewpoints are usually applicable not only to the current system under development but also to other systems of similar kind. This allows the reuse of viewpoints for same concerns for similar, but different systems. Additionally, viewpoints are later used to specify the desired consistency in a generic way (Section 2.3 ☞71). As look-ahead, in order to explicitly define the concepts and their structure of the selected parts of the system, viewpoints use abstract vocabulary for this task in form of metamodels. Therefore, each viewpoint comes with a metamodel (Definition 13 ☞61), which is rendered as class diagram, as discussed in Section 2.2.2 ☞60.

Viewpoints reflect Concerns by selecting appropriate Concepts of Systems (realized as Metamodel)

Viewpoints rendered as Class Diagrams

Excursion: Related Work for Viewpoints

Originally, the term viewpoint was introduced by Finkelstein, Kramer and Goedicke (1990). Darke and Shanks (1996) present a survey and classification on viewpoint approaches in the field of requirements engineering. Atkinson and Tunjic (2014a) give hints for identifying orthogonal dimensions, which help to organize and align a high number of viewpoints according to some principles: Two examples for such orthogonal dimensions are components respectively sub systems in the system and the level of abstraction matching the concerns of involved stakeholders. These dimensions can be used to identify and develop viewpoints for each combination of one value per di-

selected Related Work for Viewpoints

mension, e. g. a viewpoint for the user management (value for dimension sub system) with focus on the data (value for dimension level of abstraction) for developing the data base, or a viewpoint for the web interface (value for dimension sub system) with the focus on the user interface (value for dimension level of abstraction) for usability studies, and so on.

Viewtype

Clements, Garlan et al. (2002) use the term *viewtype* with the following definition: "*A viewtype defines the element types and relationship types used to describe the [...] system from a particular perspective*" (Clements, Garlan et al., 2002, p. 18). This definition is very similar to the above definition for viewpoints, while Definition 10 ☞55 emphasizes the applicability of viewpoints to multiple systems according to multiple concerns more. Goldschmidt, Becker and Burger (2012) distinguish between viewpoints and viewtypes with slightly different understandings: Each viewpoint targets some[a] concerns by selecting concepts of the system under development on more conceptual level. Viewpoints define multiple viewtypes which are metamodels (Bruneliere, Burger et al., 2019) and which additionally define concrete syntaxes (Goldschmidt, Becker and Burger, 2012). In this thesis, only the term viewpoint is used like a mapping of concerns to a metamodel and to definitions for concrete syntaxes for views. Concrete syntaxes are discussed in following paragraphs.

Viewpoint vs Viewtype

[a] Goldschmidt, Becker and Burger (2012) define, that one viewpoint covers exactly one concern (and vice versa), while this strong 1-to-1 mapping is relaxed to "*a combination, partioning and/or restriction of concerns*" (Bruneliere, Burger et al., 2019) later.

Views show Parts of a particular System according to a Viewpoint

When a stakeholder looks with particular concerns at one concrete system, a viewpoint supporting her/his concerns determines, which parts of the system are shown to her/him. The shown result is named *view* in Definition 11:

Definition 11: **View**

A view represents one concrete "*system from the perspective of specific system concerns*" (IEEE, 2011) according to a viewpoint which supports these concerns.

It should be highlighted, that views whose inter-view consistency is discussed all have to represent parts of the *same* system. The other way around, a system can be represented by multiple views at the same time (Figure 2.2 ☞54), which must be consistent to the system under development and to each other. In particular, the whole system is completely described by its views. As look-ahead, each view comes with a model (Section 2.2.1 ☞59) to store its information in this thesis.

Views are visualized by Concrete Renderings

For the visualization of views, their viewpoints come with an arbitrary number of definitions for concrete syntaxes[1], as depicted in Figure 2.3 ☞57, which extends Figure 2.2 ☞54: These concrete syntax definitions enable to render each view (i. e. its model) into a "concrete rendering", if the particular view conforms to the viewpoint of the particular concrete syntax definition. With this design, viewpoints can determine concrete syntaxes for rendering views according to the concerns of the particular viewpoint. Concrete renderings can be any visualizations including graphics, diagrams, text and dedicated tools like editors for domain-specific languages (DSLs, Section 3.6.2 ☞137). Users request views and work on concrete renderings of these views. IEEE (2011) defines model kinds as "*conventions for a type of modelling*" (IEEE, 2011), which is unclear, but the given examples ("*data flow diagrams, class diagrams, Petri nets, balance sheets, organization charts and state transition models*" (IEEE, 2011)) coincide with the definitions for concrete syntaxes of viewpoints.

[1] Maro, Steghöfer et al. (2015) show an example from industry for this case, where a textual concrete syntax and a graphical concrete syntax cover the same metamodel.

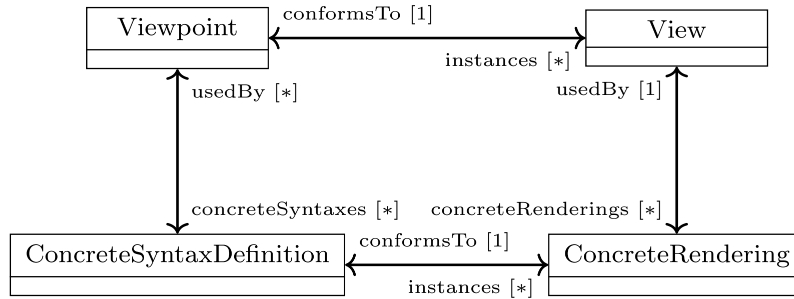


Figure 2.3: Concepts for Views and their Concrete Syntaxes

As default visualization which is always applicable, this thesis visualizes views by rendering them as object diagrams conforming to the corresponding viewpoint. Object diagrams are selected for rendering views here, since this notation is independent from the current domain and the current viewpoint. As an alternative, concrete renderings defined by DSLs provide a tailored and more specific visualization according to the particular domain and concerns, but needs to be developed for each viewpoint, which is not in the focus of this thesis.

Views rendered as Object Diagrams (by default)

Coming back to Figure 2.1 ☞ 52, *views* are depicted in its right part (gray area). Views model parts of the system under development in reality, leading to views for requirements, for class diagrams and for Java source code. Since the information about parts of the system under development is encoded as model, each view has one such model (see Figure 2.22 ☞ 90) and the views are depicted as models in Figure 2.1 ☞ 52. The system under development can be completely described as a whole with a view, which can be seen as a "complete view of the complete system". This view often does not explicitly exists and therefore is visualized as cloud "System uD Description" in Figure 2.1 ☞ 52. Compared with IEEE (2011), it refers to the "Architecture Description", which describes the architecture of the system under development as a whole.

Views in the Megamodel

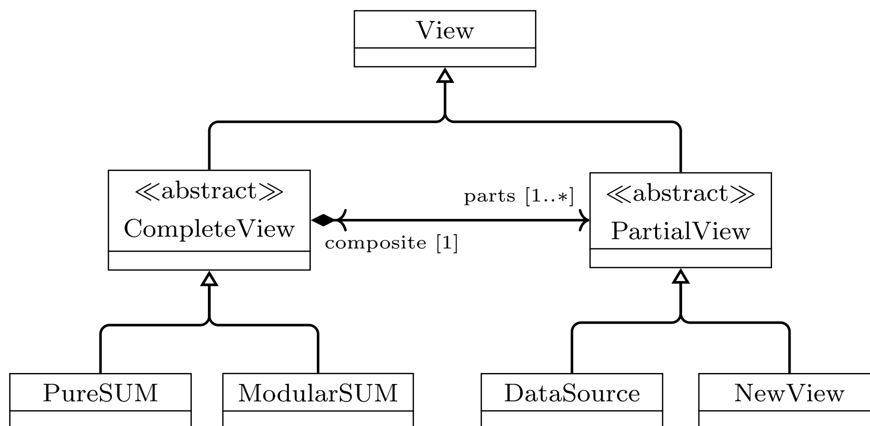


Figure 2.4: Kinds of Views in Multi-Perspective Modeling

Therefore, views can be distinguished into partial views and complete views, as depicted in Figure 2.4: According to Figure 2.1 ☞ 52, views representing parts of the system under development like requirements, class diagrams and Java source code are called *partial views*. According to Section 1.2 ☞ 31, partial views either are already existing as data sources or are newly derived as new views: *Data sources* have an already existing view and viewpoint which must be reused (Definition 4 ☞ 37), while *new views* and viewpoints are created by the approach for the first time (Definition 5 ☞ 40). In order to keep Figure 2.1 ☞ 52 short, it shows only the three data sources as introduced in Part 5 ☞ 37 of the ongoing example and not the

Partial vs Complete Views

new view as introduced in Part 6[☞40] of the ongoing example. Since the whole model-based description of the system under development provides a complete view on it, it is called *complete view*. The information within the complete view is composed of the information of all partial views, indicated by the composition in Figure 2.4[☞57]. As look-ahead, it can be explicitly realized as pure SUM or as modular SUM, as discussed in Section 3.4[☞120]. With these clarifications for views, Figure 2.4[☞57] complements Figure 2.22[☞90].

Since views respectively viewpoints are conceptually realized by models respectively metamodels, the foundations of modeling with models and metamodels are discussed in Section 2.2. The models encode the information of views, which should be kept consistent.

2.2 Modeling

Modeling for multiple Perspectives

Multi-perspective modeling was already motivated in Section 1.1[☞26] as foundation for developing software-intensive systems with models. The multiple perspectives are realized by viewpoints and conforming views. As views and viewpoints are the central concepts for developing systems, views and viewpoints must be realized, which is done by models and metamodels. Therefore, this section discusses foundations for modeling models and metamodels. The relationships between view(point)s and (meta)models is depicted in Figure 2.5, which is explained in the following sections in step-wise way.

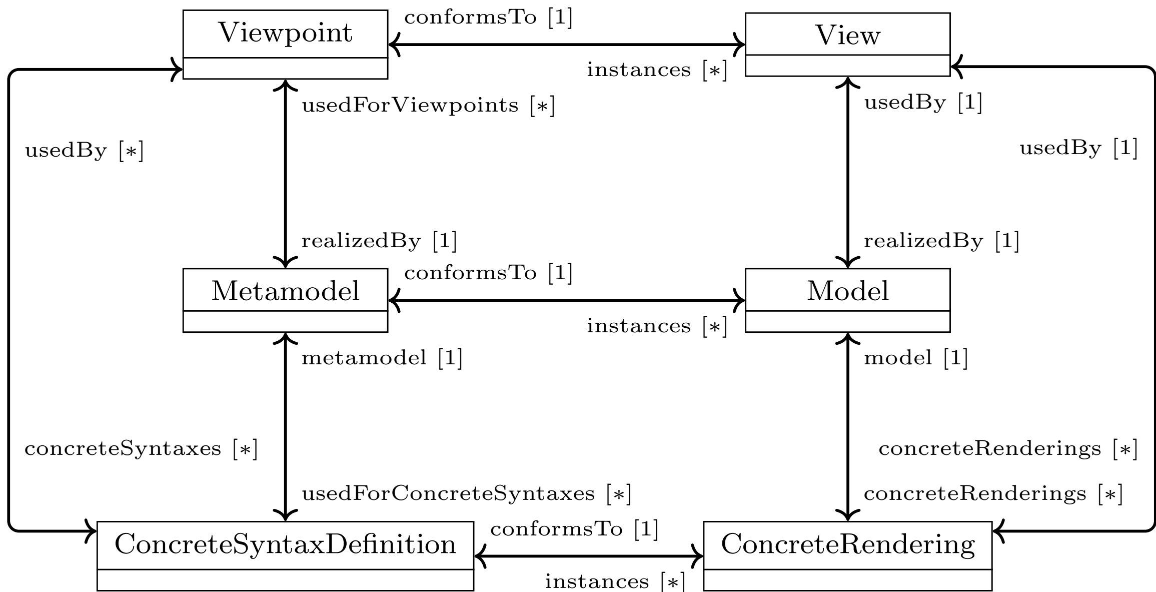


Figure 2.5: Concepts for Views and their Models

Models realize Views, Metamodels realize Viewpoints

As defined in Definition 11[☞56] in Section 2.1[☞54], views are used by users in order to be informed about parts of the system under development and to change them. As defined in Definition 10[☞55], viewpoints define which parts of a system under development are contained in the views and are often given by existing tools used in the current project.

Modeling is an Activity

Viewpoints and views are realized by metamodels and models, leading to the need to discuss some foundations of modeling: *Modeling* is an activity which creates, changes and manages models and treats models as first-order elements (Bézivin, 2006, p. 40) due to the principle, that "*everything is a model*" (Bézivin, 2005). When using modeling techniques for system development, various terms like "model-based engineering" and "model-driven development" occur:

- Jelschen (2024, p. 136f) works out, that there is no common understanding of the exact difference between the terms "model-driven X" and "model-based X": Brambilla, Cabot and Wimmer (2012, p. 9) and Pastor and Molina (2007, p. 41) both distinguish the terms model-driven and model-based regarding completeness of the used models and automation using model transformations, but place them on opposite ends of the spectrum, i. e. the definitions of terms are switched. Here, the terms model-driven X are used, since "driven sounds stronger than based" corresponding to the design decision, that all data including source code are realized as models.

Model-driven vs Model-based

- Model-driven engineering (MDE) uses modeling as key activity and involves models as central artifacts in engineering various systems along their whole life cycles.

MDE

- Model-driven development (MDD) restricts MDE to the pure development activities for systems.

MDD

- Model-driven software engineering/development (MDSE / MDSD) applies MDE / MDD techniques for engineering software systems.

MDSE / MDSD

- Model-driven architecture (MDA) is a MDSD approach of the Object Management Group (OMG) (Kleppe, Warmer and Bast, 2003; Brambilla, Cabot and Wimmer, 2012) using standards and tools of OMG and focusing on generating executable code from high-level models using chains of model transformations.

MDA

Therefore, definitions for models (Definition 12) and the related concepts metamodel (Definition 13☞61) and model transformation (Definition 14☞67), and discussions about technical spaces (Section 2.5☞84) are required, which are inspired by definitions of Jelschen (2024, pp. 132ff.).

2.2.1 Model

Models are the central artifacts of modeling. Models are defined using the definition of Jelschen (2024, pp. 132ff.), which bases on among others Stachowiak (1973), Skyttner (2005) and Hesse and Mayr (2008) in slightly simplified way.

> Definition 12: **Model**
>
> "*A model represents a part of a real system, reduced to serve a particular purpose.*" (Jelschen, 2024)

This definition condenses the three characteristics of models, i. e. first their description of *real systems*, second their *abstraction* i. e. representation of only some aspects of the systems by means for *reduction* and third their *purpose*, which determines the selection of aspects of the systems to represent. Models as representations of real systems help to deal with them as representatives, instead of working with the elements of the real systems directly, in particular for cases, when that is impossible, since the real system e. g. is not digital or does not yet exist.

Purpose determines the Reduction of the modeled System

Therefore, models are suited to store the information which shall be provided by views, since views describe parts of the system under development tailored to the concerns of its viewpoint (Figure 2.2☞54). While models store information for views, their visualizations for stakeholders is defined by the viewpoint as concrete syntax (Figure 2.5☞58): Executing such a definition for concrete syntax with the particular model of the view leads to a concrete rendering for the view. In this thesis, all models are visualized as object diagrams by default, as already discussed for views in Section 2.1☞54. The technical realization of models in memory is discussed in Section 2.5☞84.

Models store the Information of Views

Visualization of Models (default: Object Diagrams)

As preparation to define metamodels in Section 2.2.2, different kinds of models are investigated: Following Kühne (2006), the "model-of" relationships between models and their systems must be distinguished into "token-model-of" and "type-model-of". Such models can also be called token models or type models: *Token models* represent elements of the system in a one-to-one manner by creating one corresponding element in the model (but reduced according to the purpose of the model) for each element in the system. *Type models* classify the elements of the system into groups of elements with similar properties and creates one element in the model for each of these groups representing the "main nature" of the included elements of the system in a many-to-one manner. This can be used to describe elements in the system as instances of an element in the model as their type. This distinction corresponds to the terms "representedBy" for token models and "conformsTo" for type models in Bézivin (2005). Another important difference is, that representedBy is transitive, i. e. if A is represented by B and B is represented by C, then A is also represented by C, while conformsTo is not transitive, i. e. A does not conform to C, if A conforms to B and B conforms to C (Kühne, 2006).

Token Model: 1-to-1 transitive "represented by"

Type Model: n-to-1 not transitive "conforms to"

Ongoing Example, Part 8: Modeling ← List →

Modeling is applied to the ongoing example in Figure 2.1 ☞ 52 in form of μ-relationships: They describe the relationships between models and their represented systems. Here, views represent parts of the system under development like the concrete requirements, class diagrams and source code. As an example, the view for Java source code consists of a model (called Abstract Syntax Graph (ASG)) representing the Java source code under development in textual form. Since this representation is 1-to-1 according to token models, the μ-links are annotated with "representedBy" in Figure 2.1 ☞ 52.

Since this representation property of models comes also with a reduction, i. e. not everything of the system under development is reflected in the model, synchronization is needed on technical level: Stakeholders usually work not directly on the models of the views, but work on parts of the represented systems as concrete renderings of the models (Figure 2.5 ☞ 58). Therefore, the information about the system under development is the same, but it is presented with different techniques. Therefore, it is necessary to switch between different technical spaces (introduced in Definition 21 ☞ 84) of the view and its concrete renderings of the represented systems in reality. Additionally, *changes* in the concrete renderings of the represented systems must be propagated into the views and vice versa. Later on, this leads to the motivation for and the realization by adapters (Section 6.6.5 ☞ 226).

Synchronize Views with represented (Parts of) Systems regarding technical Presentation → Adapter

2.2.2 Metamodel

Models are systems, following Skyttner (2005, pp. 59–62), who distinguishes systems as "conceptual systems" from "concrete systems". Systems under development can be seen as concrete systems here and models as conceptual systems, while all are systems. This allows to create models which describe models as targeted systems. This can be used to conceptually specify the allowed concepts and their relationships in models by metamodels: By abstracting from individual details of all concrete elements of the model, each model conforms to a metamodel (Bézivin, 2005). This relationship between models and their metamodels is not called "instance of" (i. e. a model is an instance of its metamodel), since the term "instance of" is overloaded, in particular by object-orientation (Bézivin, 2005, p. 172). Therefore, "conforms to" is chosen as alternative. The statement, that a model conforms to its metamodel, indicates, that a metamodel is not only "*a model used to model modeling itself*" (Object Management Group, 2019, p. 3), often summarized as "a model for a model", but that the metamodel must be a type model for the model (Kühne, 2006).

modeling Models

Model conforms to Metamodel

Summarizing, metamodels are defined in Definition 13:

Definition 13: Metamodel

A model MM is called metamodel for the model M, if MM is used as type model for M.

Compared with classical data bases, if the model represents the instances, then the metamodel represents their schema. Another important aspect is the recursive behaviour of metamodels: Since a metamodel is also a model, it is possible to define a metamodel for a metamodel, which can be called a meta-metamodel, and so on. This emphasizes, that the designation of a metamodel is a role of a model regarding another model (Kühne, 2006). Note, that generalization is a completely different concept, since it relates model elements within the same meta level and is transitive (Kühne, 2006).

Recursion of Metamodels

To clarify the understandings of the central term metamodel, two different approaches for (meta-)modeling, the OMG model stack and Multi-Level Modeling (MLM), are compared regarding their understanding of the term metamodel in the following. In order to see applications instead, this excursion into theory can be skipped and applications of metamodels and models can be found afterwards in Part 9 ☞64 of the ongoing example.

Excursion: Metamodels as understood by OMG and Multi-Level Modeling

OMG Model Stack

The OMG defines their *model stack* with four levels, called M_0, M_1, M_2 and M_3, with conformsTo relationships between them, shown in Figure 2.6: The system with its real elements (M_0) is described by models in form of UML diagrams in various kinds (M_1), which are defined by the metamodel in form of the UML super structure (Object Management Group, 2017). The UML super structure (M_2) is defined by the meta-metamodel in form of the Meta Object Facility (MOF) (Object Management Group, 2019), which is defined with its own concepts (M_3).

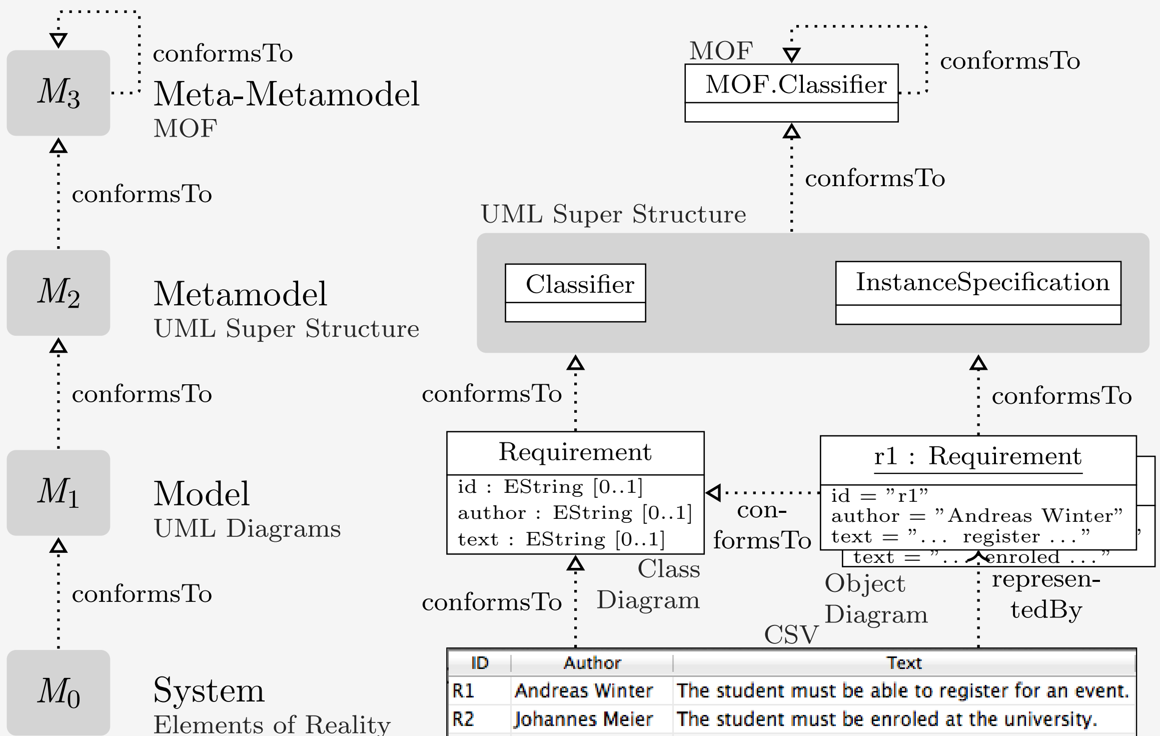


Figure 2.6: The OMG Model Stack (left) applied to represent the ongoing requirements (right)

Applied to requirements of the ongoing example, the current set of elicited requirements in the CSV file are located at M_0 and can be described with the class diagrams (left) and object diagrams (right) in UML at M_1. Comparing class diagram and object diagram regarding their relation to the real requirements shows, that the requirements conformTo the class diagram, since many requirements are described by one class (type-model-of), while each requirement on M_0 is representedBy one corresponding object in M_1 in a one-to-one manner (token-model-of). This is inconsistent and is mirrored by different statements about the relationship between M_0 and M_1 in literature (e. g. conformsTo in Hesse and Mayr (2008, p. 390) and representedBy in Bézivin (2005, p. 178)).

M_0 conforms to VS represented by M_1

Only the elements in M_2, e. g. the UML super structure, together are called *metamodel* by definition. This can be relaxed on the one hand and more formalized on the other hand by the definition of Hesse and Mayr (2008) for metamodels: A model C is called metamodel for model A, if there is another model B which conformsTo C and A conformsTo B. By allowing the real system to be A (the original definition requires a model for A) and taking the class diagram for B, this definition confirms, that the UML super structure (as C) is the metamodel (for the real system as A). As already discussed, that is not really true, if the object diagram is taken for B, since the requirements do not conformTo the object diagram. As benefit of the relaxed definition of Hesse and Mayr (2008), M_3 can be called a metamodel for M_1.

M_2 is the Metamodel (for M_0)

Between the requirements in the object diagram and the requirement in the class diagram, there is another conformsTo relationship, since multiple requirements are mapped to the single concept of requirements (type-model-of). But this introduces a *conflict with transitivity of conformsTo*: Requirements in the object diagram (as A) conformTo the requirement in the class diagram (as B), which conformsTo the UML super structure (as C). Since conformsTo must not be transitive, the transitive "shortcut" A conformsTo C must not exist, but the object diagram A conformsTo the UML super structure C, too.

Conflict with Transitivity

Due to these issues in the OMG model stack (Atkinson and Kühne, 2002b) and in order to "*reduc[e] accidental complexity in domain models*" (Atkinson and Kühne, 2008) in general, *multi-level modeling* (MLM) distinguishes two kinds of conformsTo relations (Atkinson and Kühne, 2003; Kühne, 2006): *Ontological* conformsTo, leading to the ontological levels O_i in Figure 2.7☞63, targets the contentwise conformsTo within the problem domain. *Linguistic* conformsTo, leading to the linguistic levels L_i in Figure 2.7☞63, targets the technical realization of (meta)models.

Multi-Level Modeling (MLM)

conformsTo: Ontological vs Linguistic

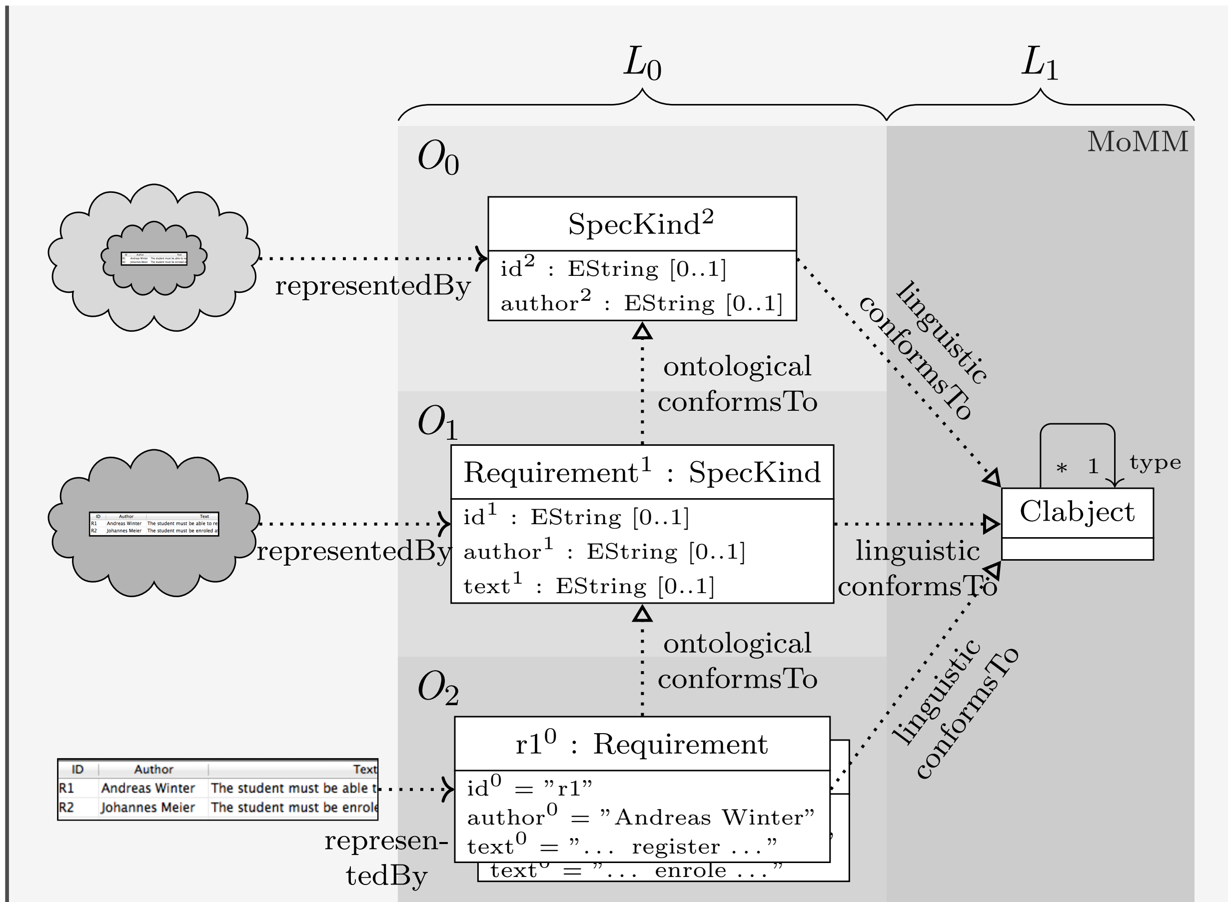


Figure 2.7: Multi-level modeling (MLM) applied to represent the ongoing requirements

Foundations of multi-level Modeling

The real requirements (bottom left) are represented in a one-by-one manner by objects in O_2 with the type `Requirement`, which is defined in O_1 to represent the concept (or idea) of requirements (middle left). Since there are different kinds, how functionality can be specified in system development, O_0 represents the concept of different concepts for specifying requirements (top left) by `SpecKind` (specification kind). Additionally to requirements, user stories are another way to formulate functionality, which could be stated as `UserStory`^{1} : `SpecKind` in O_1 (not part of the figure).

The potency of elements and their attributes (superscripted numbers) is reduced by one for each following ontological level and indicates at the value zero, that no more instantiations are possible for elements (e. g. r1^0) respectively that concrete values must be given for attributes (e. g. id^0 = "r1"). Note, that the understanding of potency and the numbering of ontological levels is reworked by Kühne (2018).

L_1 defines the Concepts of multi-level Models

Up to now, only content in form of concepts is described in the ontological levels O_i, called together one *multi-level model*, which is organized in the column L_0: To realize the complete multi-level model *technically*, a data structure is required, which is located in L_1 and can be seen as linguistic metamodel. `Clabject` (fusing the terms class + object) indicates the most central element of a linguistic metamodel for multiple metalevels (MoMM) (Atkinson and Kühne, 2001), sometimes also called model element or instance. Since the `type` of a clabject is again a `Clabject` (on higher ontological level), the MoMM can describe the whole multi-level model (linguistic conformsTo).

Resolve Conflicts with Transitivity of conformsTo

By organizing ontological and linguistic levels clearly separated as orthogonal concepts as visualized in Figure 2.7, conflicts with transitivity can be prevented, since the transitivity must not exist for the *same kind of conformsTo*. Compared with the OMG model stack in Figure 2.6☞ 61, conformsTo between class diagram and object diagram is ontological, while

conformsTo between the UML diagrams and the UML super structure is linguistic. These relationships between the involved (meta)models could be formalized with megamodels, too (Gašević, Kaviani and Hatala, 2007).

With MLM, it is possible to support an arbitrary number of ontological levels ($O_{0,1,2,\ldots}$). In contrast, UML supports only modeling with two ontological levels, e. g. with object diagrams and class diagrams (both in M_1), which refer to O_2 and O_1 in Figure 2.7☞63. The elements in all ontological levels O_i together form one *multi-level model*. Therefore, the term metamodel is not that important in MLM, but is formally defined by Kühne (2006) and is formulated simplified and similar to Hesse and Mayr (2008) as follows: A model C is called (ontological) metamodel for model A, if there is another model B which (ontological) conformsTo C and A (ontological) conformsTo B. Applying this definition to Figure 2.7☞63 results in the finding, that O_0 is a metamodel for O_2, since O_2 ontological conformsTo O_1 and O_1 ontological conformsTo O_0.

$O_0 + O_1 + O_2 + \ldots =$ multi-level Model

O_0 is Metamodel for O_2

The handling of metamodels, i. e. model elements on three or more linguistic levels with type-model-of relationships between them, would be out of the scope of this thesis, following MLM. Therefore, Definition 13☞61 requires *only one conformsTo relationship* between two models as precondition for calling one of them being a metamodel for the other model. Another finding when reflecting Definition 13☞61 is, that the mentioned conformsTo relationship can be concretized to target only ontological relationships. There are other approaches for multi-level modeling (Atkinson, Gerbig and Kühne, 2014) as the one sketched in Gonzalez-Perez and Henderson-Sellers (2008), but they are skipped here, since they are not required for the general understanding of modeling with multiple levels.

Compare Definition 13☞61 with OMG and MLM

Part 13☞90 of the ongoing example demonstrates, how the chosen definitions for metamodels and models are applied, in comparison to the understandings of OMG and MLM. To avoid the term meta-metamodel of the OMG in this thesis, technical spaces similar to L_1 in MLM are discussed in Section 2.5☞84 for technical realization of models and metamodels.

In this thesis, all metamodels are visualized as class diagrams. The technical realization of metamodels is discussed in Section 2.5☞84. To give a practical understanding of models and metamodels, examples for them are given for the ongoing example now:

Ongoing Example, Part 9: **Used Metamodels and Models** ← List →

In order to make clear, how the finally used metamodels and models for the ongoing example look like, they are shown now. For all data sources (as introduced in Part 5☞37 of the ongoing example), their metamodels and models are visualized here: Since the concepts and used data of data sources are already described, the metamodels and models are not discussed in detail here. Metamodels are visualized as class diagrams and models are visualized as object diagrams. EMF is used as technical space, as discussed in Section 2.5☞84 and Part 13☞90 of the ongoing example.

The metamodel for requirements is shown in Figure 2.8☞65: The `RequirementsSpecification` contains multiple `Requirements`, whose content is stored as `text`. The structure of the metamodel depends on the CSV format and its supporting adapter, which is introduced in Part 24☞276 of the ongoing example.

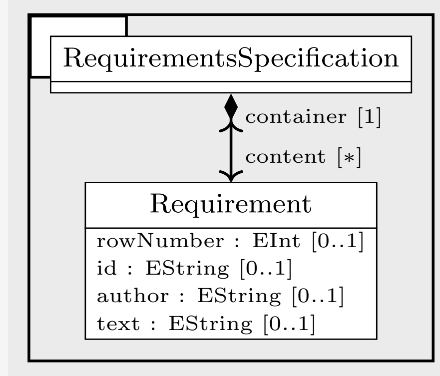


Figure 2.8: Metamodel for the data source Requirements

The model for requirements is shown in Figure 2.9 and contains the requirements `r1` and `r2` representing the two requirements in the data source.

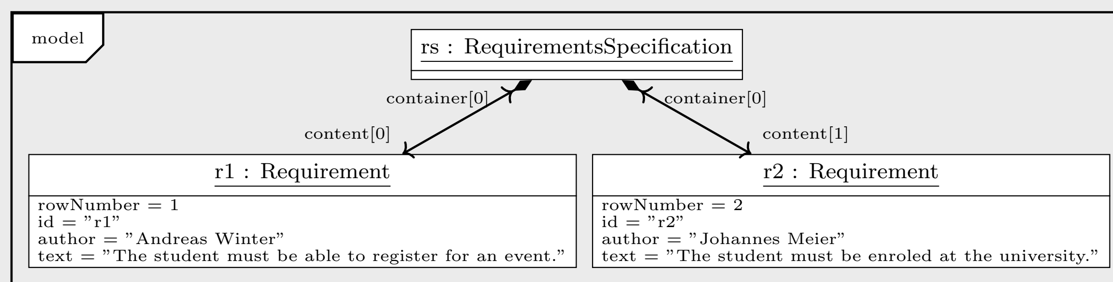


Figure 2.9: Model for the data source Requirements

The metamodel for Java is shown in Figure 2.10: The `JavaASG` represents the whole source code consisting of `ClassType`s, which have `Method`s. The call hierarchy of methods is modeled with the bidirectional association `calling` respectively `calledBy`.

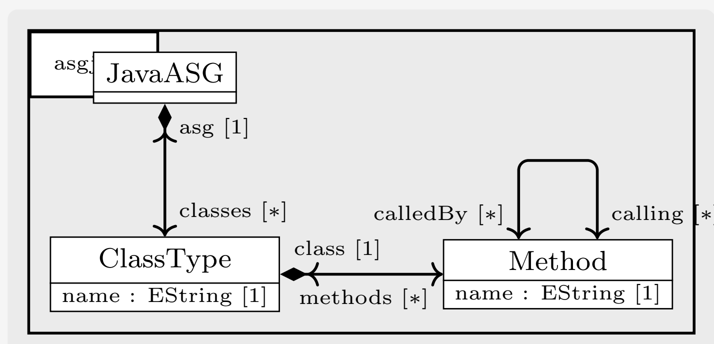


Figure 2.10: Metamodel for the data source Java

The model for Java is shown in Figure 2.11 ☞ 66 and contains the two classes "University" (`j1`) and "Student" (`j2`). Both have one method and the method "register" calls "start".

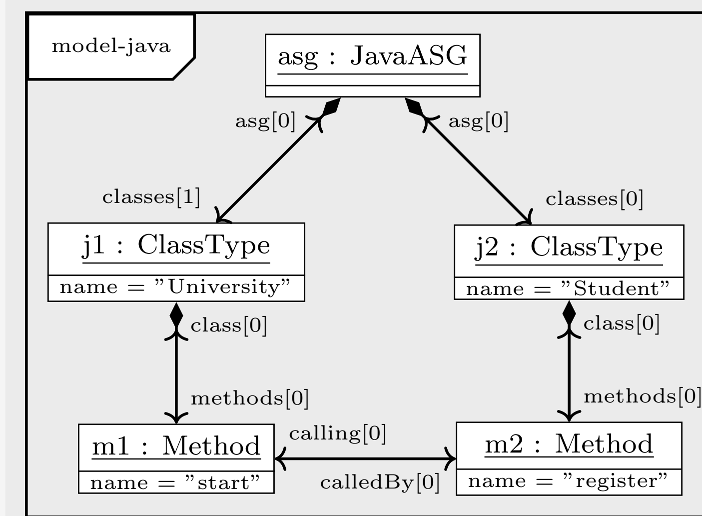


Figure 2.11: Model for the data source Java

The metamodel for UML class diagrams is shown in Figure 2.12: A `ClassDiagram` contains `Class`es, which contain unidirectional `Association`s. Associations have, among others, one type which is again a class.

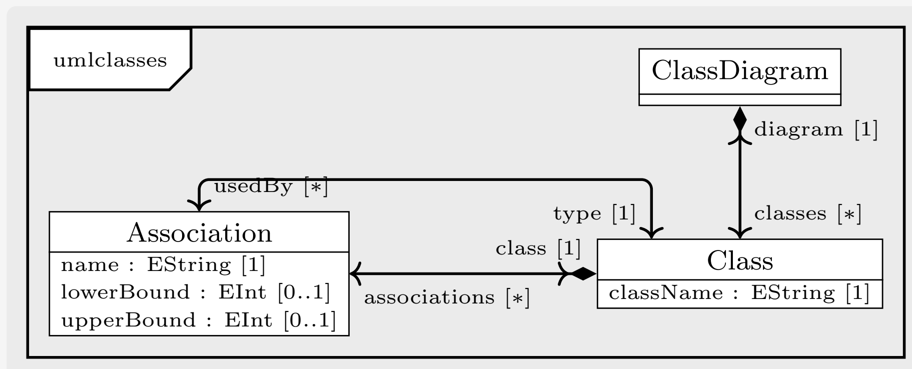


Figure 2.12: Metamodel for the data source UML

The model for UML class diagrams is shown in Figure 2.13. It models one class diagram (`uml`) with only one class "University" (`cd1`).

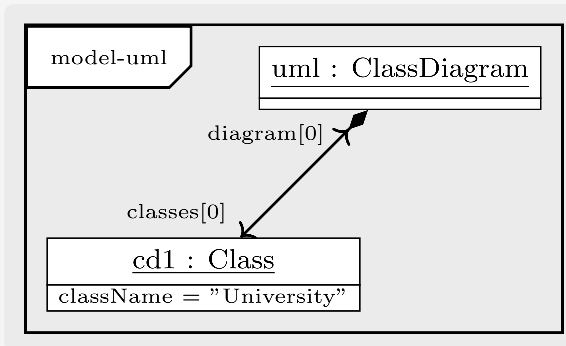


Figure 2.13: Model for the data source UML

These metamodels and models for data sources are used as starting point, as presented in the following parts of the ongoing example.

After defining models and their metamodels, these two concepts are compared regarding the objectives of this thesis (Section 1.3.1 ☞42): Objective of this thesis is to keep models (not their metamodels) consistent to each other, while these models conform to different metamodels. These metamodels are used for the *technical description and realization* of the desired consistency in order to be valid for all models which conform to these metamodels. With this idea, users can create any models using their viewpoint respectively metamodel and the consistency which is generically specified on level of the metamodel can be ensured automatically.

Consistency Preservation targets Models, but is defined on their Metamodels

This works only, if models really conform to their metamodels. Therefore, this relationship is important, is depicted as "conformsTo χ" in Figure 2.1 ☞52 and describes the relationships between a schema and its instances (Bézivin, Jouault and Valduriez, 2004), i. e. the instance (model) conforms to its schema (metamodel). This relationships between models and their metamodels must be ensured, when models or metamodels are changed: Changing models must ensure, that they still conform to their metamodels. Changing metamodels must ensure, that their models are changed accordingly, which points to the challenge of model co-evolution, which is discussed in Section 6.2.1 ☞193. As already discussed in Section 1.3.2 ☞43, the conformance of models to their metamodels must be ensured, but must be distinguished from ensuring consistency between models representing views here.

Model conforms to its Metamodel

Model Co-Evolution

In order to keep models synchronized with their systems, bidirectional model transformations are one possibility for the technical realization of synchronization in both directions (Stevens, 2018), i. e. update models as representatives of their updated (parts of the) system (according to the descriptive role of models) and update the systems under development according to updates in their models (according to the prescriptive role of models). This use case for model transformations in this thesis is covered by the concept of adapters (Section 6.6.5 ☞226).

Model Transformations for synchronizing Models and their Systems

In order to relate models to each other regarding their consistency, model transformations can be used to make these relations explicit and to maintain it, as deepened in Section 3.2 ☞99. This use case for model transformations is more important in this thesis due to its direct relation to consistency than the first use case. These discussions are deepened in Section 2.3 ☞71. But both use cases motivate the need for model transformations as introduced in the next Section 2.2.3.

Model Transformation for synchronizing Models with each other

2.2.3 Model Transformation

After defining the central terms model and metamodel, model transformations as important technique to work with models conforming to metamodels are introduced in Definition 14, similarly to Jelschen (2024), following definitions of Kleppe, Warmer and Bast (2003):

Definition 14: **Model Transformation**

A *model transformation* produces a target model out of a source model, following a given model transformation definition.

Model Transformation

In order to distinguish specification and execution, this definition for executed model transformation are accompanied with three additional terms (Kleppe, Warmer and Bast, 2003, pp. 23–26): A *model transformation definition* specifies unambiguously, how the source model is transformed into the target model, by using model transformation rules and controlling their application (Czarnecki and Helsen, 2006, p. 627). A *model transformation rule* specifies unambiguously, how single elements of the source model are transformed into

Model Transformation Definition

Model Transformation Rule

single elements in the target model. Model transformation rules are the "*smallest units of transformation*" (Czarnecki and Helsen, 2006, p. 627). A *model transformation engine* realizes a model transformation automatically by producing the target model out of the source model according to the given model transformation definition.

Model Transformation Engine

Kleppe, Warmer and Bast (2003) define these four terms without the prefix "model", e. g. transformation instead of model transformation. Since modeling concepts can be mapped to graph terminology for realization (Ehrig, Ermel et al., 2015b, p. 49), the generic definitions of Kleppe, Warmer and Bast (2003) could be applied also to graph transformations. Here, the prefix "model" is added to emphasize the context of modeling.

Engine vs Tool

Kleppe, Warmer and Bast (2003) use the term transformation tool, while model transformation engine is preferred here, since tools could contain additional parts like a graphical user interface, next to the model transformation engine. The term engine is used also in other work like Czarnecki and Helsen (2006).

Model Transformation (Definition)

While the term model transformation (the execution) refers to the execution of a model transformation definition (the specification), model transformation is often used as abbreviation for model transformation definition, since the context shows, if the execution phase or the specification phase is discussed. Therefore, this thesis uses this abbreviation, too.

Specifications on Metamodels, Executions on conforming Models

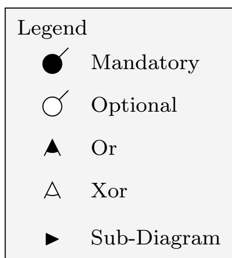
The specifications of model transformation definitions and model transformation rules are done on metamodel level, e. g. on the source metamodel, to be directly applicable to all source models which conform to this source metamodel.

Input: all kinds of Models, e. g. Models, Metamodels, Model Transformations, . . .

Model transformations can also use metamodels as source input and produce models or metamodels as target output, since metamodels are also models. Corresponding model transformation definitions are specified on the metamodel of these metamodels then. Additionally, model transformation (definitions) can be source model or target model of model transformations, since model transformations are also models (Bézivin, 2005; Bézivin, Büttner et al., 2006). Therefore, each approach for model transformations provides a metamodel describing the concepts which can be used for model transformation definitions.

generic Surveys

There is a wide range of model transformation approaches, as shown by surveys of Jakumeit, Buchwald et al. (2014) and Kahani, Bagherzadeh et al. (2019). Model transformation approaches can be classified regarding different properties, as done by Czarnecki and Helsen (2006) in form of feature models for functional criteria. Mens and Van Gorp (2006) propose additional quality criteria for model transformations. Classifications which are required for this thesis are adapted from Czarnecki and Helsen (2006), depicted in Figure 2.14 and introduced now:



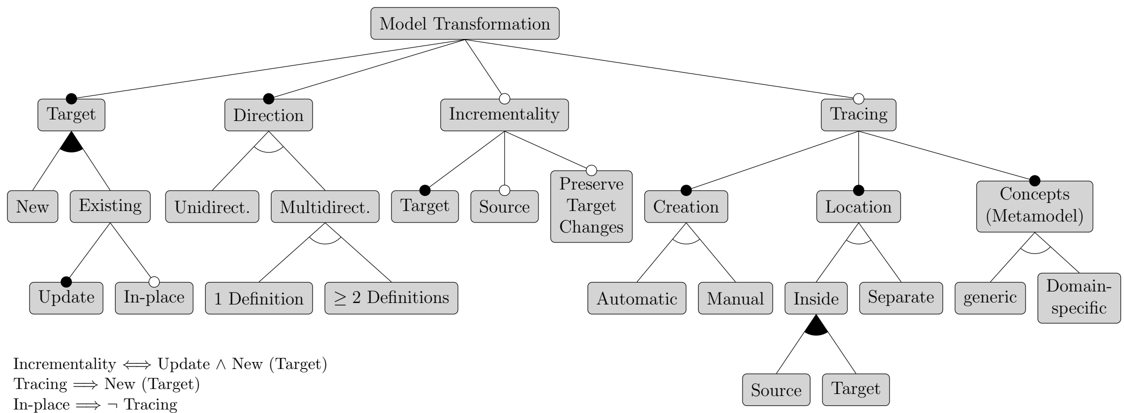


Figure 2.14: Feature Model for Model Transformations (adapted and extended from Czarnecki and Helsen (2006))

- Model transformation work with one or more models: Each model can be either read or written or both, referring to e. g. source models, target models or models which

are transformed in-place. As an example, model merging has two or more source models and creates one target model (Mens and Van Gorp, 2006). The *target* model can be created *newly* or is already *existing*, since the source model is transformed *in-place* into the target model or since the target model of an out-place model transformation already exists (due to previous activities, e. g. a previous execution of the transformation which created the target model newly) and will be *updated* only. Model transformations whose involved models all conform to the same metamodel are called *endogenous* and *exogenous* otherwise (Mens and Van Gorp, 2006).

new vs updated Target

transform Source in-place

endogenous vs exogenous

These classifications for the involved models are important for this thesis, since models of existing data sources must be reused and updated on the one hand, while new models must be created for new views.

- *Unidirectional* model transformations always use the source model to provide the target model (one and same direction). *Multidirectional* model transformations (MX) can switch the roles of source and target models to support multiple directions, e. g. *bidirectional* model transformations (BX) as special case of multidirectional model transformations can transform the source model into the target model and the target model into the source model (two directions). Multidirectional model transformations can be defined by providing one set of (multidirectional) model transformation rules which cover multiple directions within each model transformation rule or by providing one set of (unidirectional) model transformation rules for each direction. In literature, bidirectional model transformations are often understood to have only *one definition* for both directions (Abou-Saleh, Cheney et al., 2018).

  unidirectional vs bidirectional

  Bidirectionality is important for this thesis, since information of models of existing data sources must be transformed into models of new views, while changed information in these models for new views must be transferred back into the models of the data sources.

- *Incrementality* aims to benefit from known changes in models instead of complete transformations in batch-mode and comprises three different aspects:

  Incrementality transforms only changed Elements again

  - *Target incrementality* enables to update an existing target model, when running the model transformation with an updated source model again. This feature corresponds to the already used term *change propagation* of user changes in the source model.
  - *Source incrementality* enables to execute only those model transformation rules again, which use updated elements of the source model. After an impact analysis of the user changes in the source model, only the identified model transformation rules are executed to update the target model accordingly. This can improve the performance, in particular for huge source models.
  - *Preservation of user changes in the target model* deals with the challenge of user changes in the target model: When running the model transformation again with an updated source model, the target model must be updated accordingly, but additional user changes in the target model should be preserved, too.

  Incrementality is important for this thesis, since it allows to keep only some information in models consistent, while all other information is kept unchanged, in contrast to complete model transformations in batch-mode which tend to recreate such models in order to replace the previous model which deletes information not covered by the model transformation. Incremental execution must be distinguished from *lazy* execution (Tisi, Martínez et al., 2011), which updates model elements in delayed way at the moment, when the model elements are requested.

Traces between Source and Target

- *Tracing* adds trace links between source elements of the source model and corresponding target elements of the target model into the model transformation, according to the model transformation rules. Traces can be created *automatically* or *manually* by writing requests for traces within the model transformation rules. Traces can be stored *inside* the involved (source or target) models, e. g. by using dedicated UML profiles (Vanhooff and Berbers, 2005), or outside in a *separate* model conforming to a trace metamodel (Schwarz, Ebert and Winter, 2010). The concepts of traces (which are not covered by Czarnecki and Helsen (2006), but discussed by Hidaka, Tisi et al. (2016) and by Samimi-Dehkordi, Zamani and Kolahdouz-Rahimi (2018)), e. g. the trace metamodel in case of a separate storage location, can be *generic*, e. g. the same trace metamodel is used for any source and target metamodels, or *domain-specific*, e. g. the trace metamodel depends on the particular source or target metamodels.

  Trace links are important for this thesis, since trace links indicate corresponding elements in source and target models, which is usually required for incrementality.

specific Surveys

Additionally, there are surveys focusing only on classes of model transformations with special properties, e. g. Kusel, Etzlstorfer et al. (2013) for incremental model transformations, Hildebrandt, Lambers et al. (2013) for Triple Graph Grammars (TGGs) as concrete model transformation approach (Schürr and Klar, 2008) or Stevens (2008) and Anjorin, Buchmann et al. (2020) for bidirectional model transformations.

The big picture for terminology in form of a megamodel in Figure 2.1 ☞52 does not show model transformations directly, although Bézivin, Jouault and Valduriez (2004) explicitly define model transformation as possible relationships between models: τ describes the relationship between a source model which is transformed into a target model by a model transformation. While they provide no conceptual findings for consistency in Figure 2.1 ☞52, model transformations can be used to technically ensure consistency:

Projections by Transformations

If the complete view ("System uD Description") is realized as explicit model, e. g. as SUM as it will be introduced in Section 3.4 ☞120, this model could be transformed into models for partial views by removing all information which is not part of this view, according to the δ-relationship. This leads directly to the idea of projectional management of views (Section 3.5 ☞121).

Update partial Views regarding overlapping Information

If there is no such complete view, parts of the information from one partial view could be generated from parts of another partial view. Such model transformations could update the model of the target view in case of changes of overlapping information in the model of the source view, described as synthetic management of views (Section 3.3 ☞108). Changing the target view makes it inconsistent to the source view, as long as there is no inverse transformation to update also the source view. In the ongoing example, classes in UML can be transformed into classes in Java, but without methods, since they are not described in UML. But renaming classes in UML requires to rename the corresponding classes in Java and vice versa.

Model Transformations for bridging Technical Spaces

Additionally, transformations can be used to realize μ-relationships in form of bridges between different technical spaces. Summarizing, definitions and classifications of model transformations are important, since they represent important related approaches for model consistency in Section 3.3 ☞108 and they are used as part of the designed solution in Section 6.1 ☞185.

Using multiple models as views conforming to metamodels as viewpoints introduces the potential for inconsistency between them, as already motivated in Section 1.1 ☞26. Therefore, the next Section 2.3 ☞71 provides some foundations for understanding consistency in more detail.

2.3 Consistency

When developing the system under development with multiple views conforming to different viewpoints, inconsistencies can occur between those views, which describe overlapping and related parts of this system under development. Since all views together describe the system under development in its entirety (Definition 2☞32), these views always have some dependencies leading to possible inconsistencies, as discussed in Section 1.1☞26. Therefore, this section continues to investigate the foundations of consistency. Consistency of views is already defined in Definition 2☞32 in Section 1.2.1☞31 and its main message is repeated here for readability: One or more views are consistent, if these views describe parts of the same system under development without semantic contradictions within a particular project.

Objective: Deepen the Understanding of Consistency

This understanding for consistency is concretized here and substantiated by analyzing the involved views of a system, their relationships and their impacts to consistency. This investigation is done along the ongoing example in form of the megamodel in Figure 2.1☞52. DecomposedIn-relationships δ describe the relations between a composite and one of its parts (Favre and Nguyen, 2005). Applied to multi-view modeling according to Figure 2.1☞52, there are three cases of δ-relationships: First, the system under development consists of requirements, class diagrams and Java source code in reality. Second, the model-based description of the system under development (complete view) consists of the information of all (partial) views. Third, the concepts for the description of the system under development consists of the concepts of all viewpoints.

δ-Relationships

Looking at the second case for the involved models, partial views represent some information (as reduction) of the complete view for the complete description of the system under development. Therefore, changes in partial views must be transferred also into the complete view, otherwise the δ-relationships are hurt. The other way around, changes in the complete view must be reflected also in all its partial views to ensure δ.

Keep complete View and its partial Views consistent to each other ...

Since a partial view is a *reduction* of the complete view in the sense of the *abstraction* property of models (Section 2.2.1☞59), the *purpose* of the partial view, i. e. the concerns targeted by its viewpoint, determines, which information of the complete view is selected to be part of the partial view. Therefore, δ-relationships depend on the purpose of (partial) views here and require to *ensure (semantic) consistency* between the models of the partial views in order to model the same system under development in coherent way. If the complete view is made explicit, synchronization between partial views and their complete view are sufficient, otherwise, the partial views are synchronized directly with each other, as discussed in Section 3.2☞99.

... depending on the Purpose i. e. Semantics of partial Views

After clarifying, that the semantic consistency between different views is the main challenge of this thesis, now some activities around consistency are analyzed. The Definition 2☞32 for consistency describes a state, as visualized in Figure 2.15: Views are consistent to each other (state consistent), then a user changes one view (transaction User Changes) by changing its model, which results in model differences depicted as ${}^{\text{User}}\Delta$. Afterwards,

Consistency is a State

${}^{\text{User}}\Delta$ introduce Inconsistencies

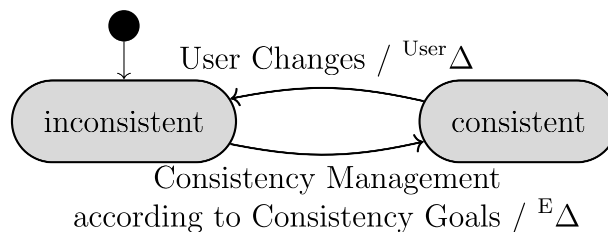


Figure 2.15: Terminology for Consistency

the views are inconsistent to each other (state inconsistent). In order to make the system

consistent again, this transition from the inconsistent state to the consistent state is realized by consistency management:

Consistency Management

$^{E}\Delta$ fixes Inconsistencies

Consistency management comprises the activities which ensure consistency according to guidelines, which are called consistency goals in the following Definition 15, by managing inconsistencies, including detecting dependencies between views corresponding to the consistency goals, detecting inconsistencies and fixing inconsistencies (Spanoudakis and Zisman, 2001). In order to transfer an inconsistent state to a consistent state, inconsistencies are detected and fixed, which results in model differences depicted as $^{E}\Delta$ (E stands for execution). The concepts $^{User}\Delta$ and $^{E}\Delta$ allow an explicit designation of these manual and automated changes. These symbolic notations are introduced here, since they are required for formal visualizations for the design in Chapter 6☞185. Technically, they are model differences, which are technically realized by difference operations (see Section 6.7☞227). Initially, the system is expected to be inconsistent (initial state in Figure 2.15☞71), since reused views might be inconsistent due to manual consistency management before (see Section 1.2.2☞36).

Since the Definition 2☞32 for consistency does not define possible contradictions between views, it does not help to determine, if given views are consistent to each other or not, i. e. if state consistent or if state inconsistent holds in Figure 2.15☞71. Therefore, Definition 15 allows to formulate conditions for consistency:

Definition 15: Consistency Goal

A consistency goal formulates a relation between elements of one ore more viewpoints. If this relation holds between corresponding elements of conforming views, these views are called consistent, otherwise, they are called inconsistent, regarding this consistency goal.

Consistency Goals formulate Conditions for Consistency

Since the viewpoints and their relations for consistency are project-specific (see page 35), each consistency goal is specific for that project, too: Consistency goals make these project-specific consistency challenges explicit and can be seen as special requirements for the desired consistency for the current project, i. e. the consistency is defined by consistency goals as its parts (Figure 2.17☞73).

Consistency Goals clarify Dependencies between Views

Consistency goals should clarify dependencies between elements of different views, since they are the origin for possible inconsistencies. Possible kinds of dependencies are identified on page 33 as redundancies, explicit links and constraints. Each consistency goal makes one of such dependencies between views explicit, so that it can be checked in order to determine, if the dependency is fulfilled or not.

Consistency Goals describe Semantics to hold between depending Information of any Views

This relationship between views and consistency goals is depicted in Figure 2.16☞73 as extension of Figure 2.4☞57: Consistency goals explicitly describe goals for the consistency of multiple views which comprise information which semantically depend on each other. If a consistency goal is linked with only one view, it describes its intra-model consistency. In general, consistency goals can target any views: If there is no complete view, consistency must hold only between partial views. If there is an explicit complete view, consistency must be ensured between the partial views and its complete view, too. From the perspective of stakeholders, they expect consistency goals to hold for partial views (only), since stakeholders usually work only with them and do not care about a possible complete view.

Consistency Goals are formulated on Viewpoints and checked on Views

Note, that consistency goals are formulated in terms of elements of the viewpoints, but must hold for corresponding elements of conforming views. Therefore, consistency goals can be evaluated like constraints for all conforming views in order to determine, whether the views are consistent or inconsistent. If all relations of all consistency goals for a project and its system under development hold, the views of this system are called consistent (otherwise, they are called inconsistent) to each other as well as to the system.

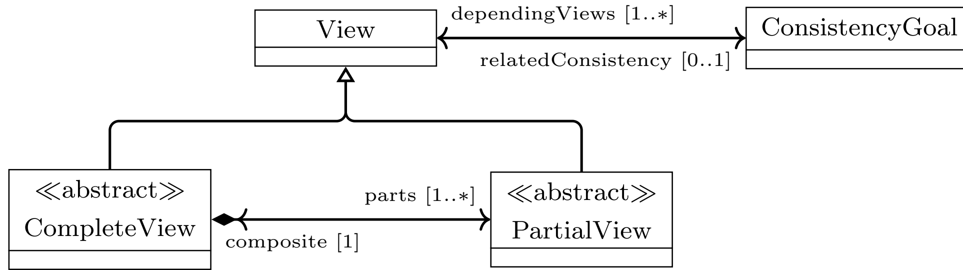


Figure 2.16: Concepts for Views and their Consistency

Instead of consistency goal, Reder and Egyed (2012) call it "design rule" and use expressions to describe the consistency formally, while consistency goal emphasizes the purpose of consistency. Dijkman, Quartel and van Sinderen (2008) call it "consistency rule", but this term is used differently here, as defined below.

alternative Terminology

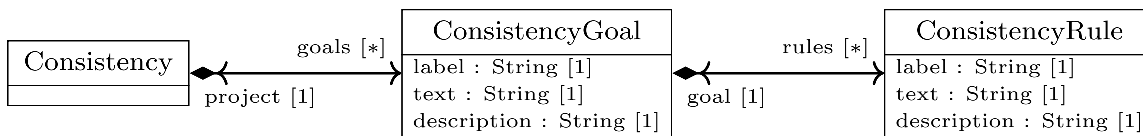


Figure 2.17: Concepts for Consistency

While Definition 15 ☞72 allows to determine, if consistency is reached or not, it is still unclear, *what* to do in order to fix inconsistencies: Inconsistent views must be changed in order to get updated views which are consistent to each other afterwards. Usually, an occurred inconsistency can be fixed in *multiple* ways, i. e. there are multiple possible fixes to repair a single inconsistency. An example is found in the following Part 10 of the ongoing example:

There are *multiple* possible Fixes

Ongoing Example, Part 10: Multiple possible Fixes ← List →

When an inconsistency occurred, usually, there is not exactly one possible fix for the inconsistency, but there are multiple possible fixes in general. As an example, the same class is represented in source code and in class diagrams: After the user renamed a class A in the class diagram to B, there are multiple ways to eliminate the occurred inconsistency, e. g.

- by creating a new class B in the source code (with or without deleting A in the source code),
- by renaming the class A in the source code to B (the chosen fix in Consistency Rule C 2 c ☞77)
- or even by deleting the renamed class B in the class diagram.

To fix the introduced inconsistency, one fix of this (incomplete) list of possible fixes must be chosen.

In general, there are multiple possible fixes for inconsistencies (Reder and Egyed, 2012). This observation can be found not only in practice but also in theoretical way: Hettel, Lawley and Raymond (2008) give another explicit example and formalize the need for a selection strategy, when defining round-trip engineering for change propagation (more details are given in Section 3.3.1 ☞108). Formalizing consistency as relation, as it is done in the following, formally shows, *why* multiple possible solutions occur. Another motivation for formalizing consistency is, that it shows the need to change also the source view in

order to fix inconsistencies. Both findings are mapped to provided functionality of related approaches in Chapter 3 ☞93.

Stevens (2010) formalizes consistency as mathematical relation $R \subseteq L(S) \times L(T)$ between source models $s \in L(S)$ and target models $t \in L(T)$ with S and T as their metamodels and $L(S)$ and $L(T)$ their induced languages as sets of all conforming models:

Consistency mathematically formalized as Relation

$$\forall s \in L(S), t \in L(T) : R(s,t) \iff s \text{ and } t \text{ are consistent to each other} \tag{2.1}$$

Informally, for each possible pair of one source model and one target model it is checked, if they are consistent to each other: If they are consistent, this pair is stored respectively marked by the relation R. If they are not consistent, this pair is not part of the relation R. In the following example in Figure 2.18, there are five combinations of five possible source models and of six possible target models which indicate consistent models, i. e. $(s_1,t_1),(s_3,t_2),(s_4,t_2),(s_5,t_5),(s_5,t_6) \in R$. Note, that the assignment of two models as source model respectively target model is artificial here and could be switched without impact.

Exemplary Consistency Relation

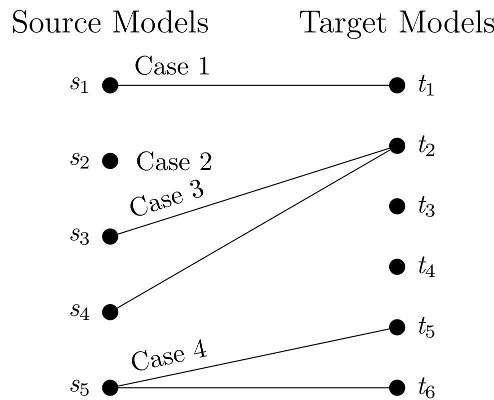


Figure 2.18: Exemplary Consistency Relation between source and target models

When looking only at the source models (in general, the same counts when switching source and target), there are four different cases, how they could be related as consistent to target models:

Cases of related Models

- In Case 1, a source model is related to exactly one target model which is related to no further source models, e. g. (s_1,t_1) in Figure 2.18.
- In Case 2, a source model is related to no target model, e. g. s_2 in Figure 2.18.
- In Case 3, a source model is related to exactly one target model which is related to two or more source models, e. g. $(s_3,t_2),(s_4,t_2)$ in Figure 2.18.
- In Case 4, a source model is related to two or more target models, e. g. $(s_5,t_5),(s_5,t_6)$ in Figure 2.18.

With these distinctions, the following results regarding the selection of possible consistent target models for a particular source model can be achieved:

Selecting consistent models

- If only Case 1 occurs in R, R is bijective and s and t determine each other completely and contain the same information with different structures (Stevens, 2008). In that bijective setting, there is only one possible fix for an inconsistent source model respectively target model without the need for selection.

- If Case 2 occurs in R, there is no related target model to be consistent with the source model. Since there is no target model, the current source model must be changed to come to another source model which have one or more related target models. With other words, the current source model is the root of the current inconsistent state and must be fixed itself. In practice, this Case 2 can occur, when the current source model does not fulfill intra-model consistency.

Change (also) the Source Model (again)

- If Case 2 does not occur in R, R can be written as function $f : L(S) \longrightarrow L(T)$: If Case 4 occurs in R, f must select one of the possible target models, otherwise f cannot be established. If Case 3 does not occur, f is called injective. If R is bijective, f is bijective, too, and there is an inverse function $f^{-1} : L(T) \longrightarrow L(S)$ for f (Stevens, 2008).

  As look ahead, functions f and f^{-1} (if existing) can be realized as (unidirectional) model transformations (see Section 2.2.3☞67). If f^{-1} exists, since f is bijective, f can be written as bidirectional model transformation and f and f^{-1} can be executed without further specifications. If f^{-1} does not exist, a (unidirectional or bidirectional) model transformation for the inverse direction must cope with the challenge to select one of multiple possible results for the desired consistency (Section 3.2☞99).

- Case 4 shows the need to select from multiple possible target models which are consistent to the current source model. Therefore, a strategy for the selection of the desired fix i. e. consistent target model is required. Such selection is not required for the Cases 1 and 3, since there is exactly one related target model.

Summarizing, relations help to formalize consistency. Depending on the particular consistency relation, a selection of one of the possible target models to be consistent to the current source model is required in general. This counts in particular for cases, where the source models contain less information than the target models, since the additional information in the target model is unknown in the source model and result in ambiguities. Another important finding is, that not only a related target model can be selected for the current source model (which remains unchanged), but also the current source model could be changed, so that another (or even any) target model related as consistent to the changed source model can be determined.

Selection of related (Source or Target) Models

Since the resolution of inconsistencies is not unique, the *selection of one of these possible fixes* for application is required (Spanoudakis and Zisman, 2001): It is important, that it is not sufficient for selected fixes to make views consistent to each other, the views must be consistent to the system under development, too. In particular, changes by users in one view must be reflected in the other views in order to *update* the system under development *accordingly*. Therefore, the fix to select must fit to the purpose of the system under development and to the changes of the users. This makes also the selection of fixes project-specific. To specify the desired fixes of inconsistencies, Definition 16 is required to guide the selection of possible fixes:

Selection of one Fix required

> Definition 16: **Consistency Rule**
>
> A consistency rule provides a strategy to ensure the consistency defined by its consistency goal.

Consistency rules are specific for a consistency goal (Figure 2.17☞73) and concretize them in order to operationalize them: This is done by providing strategies, how the consistency is ensured, which is determined by the consistency goals. This can be done by defining the desired degree of automation (e. g. manual, automated) or the general strategy (e. g. heuristics), by defining required reactions on occurred changes or by defining special cases

Consistency Rules formulate Strategies for fixing Inconsistencies

for the consistency. In contrast, consistency goals describe the desired state of consistency in more generic way.

Related Work for Consistency Specification

Kramer (2017, p. 57) distinguishes consistency check specification and consistency enforcement specification for consistency specification: The consistency enforcement specification specializes the consistency check specification by actions to realize consistency, additionally to the check, if consistency is fulfilled. Here, the consistency rules are designed to complement their consistency goals by concretizing the desired transition from the inconsistent state to a consistent state. Kramer (2017, pp. 106ff.) provides also some formal definitions for consistency using the term consistency rule, which corresponds to the term consistency goal in this thesis. To annotate consistency checking constraints with repair rules to fix inconsistencies detected with such constraints is a strategy also found in related approaches like Stünkel, König et al. (2018), which are presented with more details in Chapter 3 ☞93.

This motivates to concretize Requirement R 1 (Model Consistency) ☞154 with the following Requirement R 1.2 ☞155:

Requirement R 1.2: Generic Consistency Goals

The approach must support arbitrary consistency goals concretized by consistency rules.

This requirement will be chosen in Chapter 4 ☞153 in order to support project-specific consistency challenges in form of concrete consistency goals and their consistency rules.

Consistency formulated in natural Language

The consistency can be formulated in formal way, e. g. as demonstrated by Diskin, König and Lawford (2018) or with OCL as by Dijkman, Quartel and van Sinderen (2008) and Egyed, Zeman et al. (2018). Here, all consistency goals and consistency rules are formulated as text in natural language, since they should be used for discussions with stakeholders (Section 2.4 ☞79), who usually have no formal knowledge for that. Each consistency goal and each consistency rule is represented by one sentence, complemented with a longer description for motivation and description, as demonstrated for the ongoing example:

Ongoing Example, Part 11: **Consistency Goals and Rules** ← List →

Now, the consistency issues, which are described only roughly up to now, are summarized in form of consistency goals now. To realize the consistency goals, some concretizing consistency rules are added.

Consistency Goal **C 1**

Requirements must be linked with their fulfilling Java methods.

This consistency goal summarizes the first consistency issue in the ongoing example. It links requirements from the requirements data source explicitly with those methods from the Java data source, which fulfill the requirements.

Consistency Rule **C 1 a** for C 1

Links between requirements and fulfilling methods are added manually.

Since the identification of methods which fulfill requirements should be done manually, no automation is described here.

Consistency Rule C 1 b — for C 1

If a requirement or a method is deleted, all its links must be deleted automatically, but not the other linked element.

After removing a requirement or a method, all direct traceability links must be removed, too. The element at the other end of the link is kept. This is important, since these deletions can be done independently from the traceability, e.g. by removing the elements in their original data sources.

Consistency Goal C 2

All classes must be represented always in Java source code, but not necessarily in the UML class diagram.

The classes in UML are a subset of the classes in Java. This allows to keep UML on a higher level showing only classes which are relevant for the architecture.

Consistency Rule C 2 a — for C 2

A new class in UML must be created also in Java, but a new class in Java is not added to UML.

If the new class in UML is already existing in Java, which is possible due to C 2, these two classes are identified as same, but nothing more happens.

Consistency Rule C 2 b — for C 2

A deleted class in Java must be deleted also in UML, but a class which is deleted in UML remains in Java.

In the end, each class in Java can be shown or hidden in UML, which counts also for deletion of classes.

Consistency Rule C 2 c — for C 2

A renamed class in UML must be renamed also in Java and vice versa, if the class is also represented in UML.

Since the same class is represented in UML and Java, it must have the same name in both representations.

Consistency Goal C 3

Each association in UML must have exactly one method which provides its values (getter).

Since associations in UML are usually realized as private attributes in Java, a public getter-method is required to retrieve the values of that attribute. By convention, the name of the getter starts with the prefix get and ends with the name of the method whose first letter is a capital. If there is no getter, a new getter will be created explicitly for the association. Note, that associations are part of ClassDiagram and methods are part of Java: Due to C 2, all classes of UML defining associations are part of the Java source

code, so creating getters is always possible.

Consistency Rule **C 3 a**	for C 3
If an association is renamed, its getter must be renamed accordingly.	

Otherwise the names of association and getter do not match the convention anymore. The special case, that the renamed getter conflicts with another already existing method, is ignored here for simplicity.

Consistency Rule **C 3 b**	for C 3
If a method which is used as getter for an association is renamed, this association must be renamed accordingly.	

Otherwise the names of association and getter do not match the convention anymore. The special case, that the renamed association conflicts with another newly created association, is ignored here for simplicity.

Consistency Rule **C 3 c**	for C 3
If a method which is used as getter for an association is removed, a new getter is created for this association.	

A new getter is created, otherwise the still existing association has no getter anymore, which hurts C 3.

Consistency Rule **C 3 d**	for C 3
If an association is removed, its getter is removed, too.	

Since the purpose of getter methods is only to provide the value of the association, the getter is no longer used, when its association is removed.

Figure 2.19 presents an overview of all consistency goals, annotated along the edges[a]. The nodes in the graphic represent the data sources in this application.

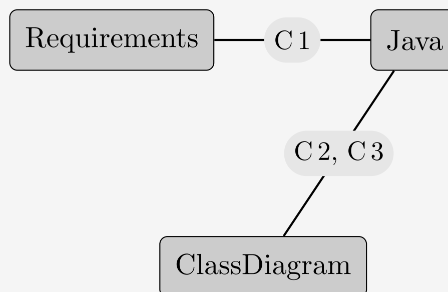


Figure 2.19: Overview of Consistency Goals in the ongoing Example

Since all involved data sources have models, Figure 2.19 could be treated also as macromodel with consistency goals as used type for relations (Salay, Mylopoulos and Easterbrook, 2009; Stevens, 2017), since megamodels have a fixed set of relation kinds between models (Favre and Nguyen, 2005).

[a]Hyperlinks at the consistency goals allow to jump to their introductions.

Usually, it is sufficient to specify consistency goals for views representing data sources only and not for new views, since new views provide only some already existing information of the data sources in a different way, but do not introduce new information. But this reused information is already targeted by the consistency goals for the views representing data sources. Therefore, Part 11 ☞76 of the ongoing example presents consistency goals and consistency rules for data sources only and not for new views.

Consistency Goals for Data Sources only

This formulation of the desired consistency in terms of consistency goals and consistency rules helps to discuss it with stakeholders and to clarify their roles in consistency management (Section 2.4).

2.4 Stakeholders

The management of inconsistencies (Spanoudakis and Zisman, 2001) involves four groups of stakeholders, i. e. users, methodologists, platform specialists and adapter providers, which are motivated and clarified in this section. Later on, these stakeholders help to adjust the design of the new approach (Chapter 6 ☞185) and its application (Chapter 12 ☞455) to their skills. Examples for these groups of stakeholders in the software development domain are collected in Part 12 ☞83 of the ongoing example.

Stakeholders for Consistency Management

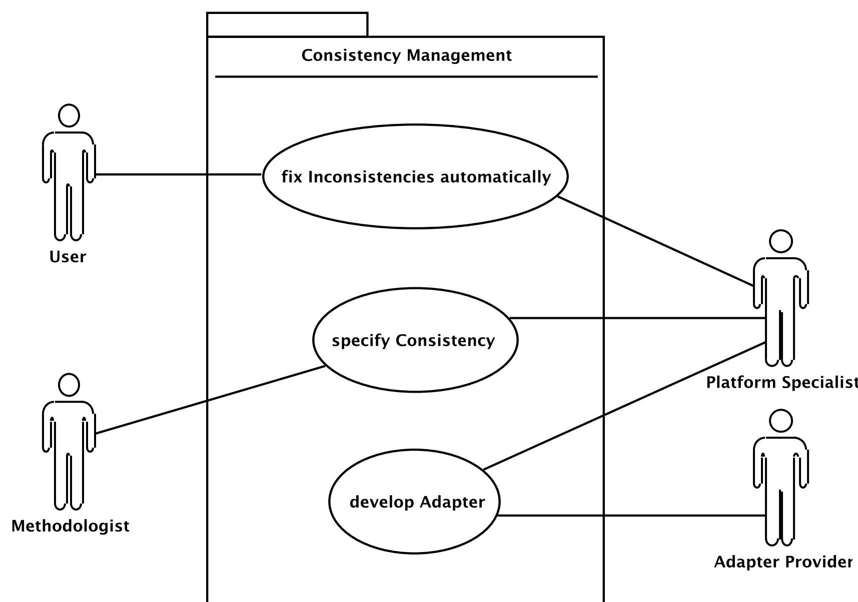


Figure 2.20: Use Cases of Consistency Management

In order to motivate the proposed stakeholders, the use cases for consistency management are discussed and depicted in Figure 2.20, in order to derive involved stakeholders for consistency management:

Use Cases for Consistency Management

Fix Inconsistencies automatically This use case automatically fixes possible inconsistencies in views, after one view was manually changed by a stakeholder involved in the current development project. The stakeholders using and changing views manually and requiring automated fixes afterwards by triggering this use case are called users. To automate such fixes for manually introduced inconsistencies within this use case is the main objective of this thesis (Section 1.3.1 ☞42).

Specify Consistency The consistency to ensure in a project is specified during this use case by explicitly formulating consistency goals and consistency rules (Section 2.3 ☞71).

Objective of this use case is to establish explicit specifications for the desired consistency in order to automatically fix inconsistencies in the previous use case.

Technical Space

Adapter

Develop Adapter Since different views are technically realized with different tools, environment and data formats (Section 1.2.2[☞36]) or different concrete syntaxes (Section 2.1[☞54]), the information encoded by these views must be mapped to the same technical representation in order to work with them in a uniform way. These technical representations are called *technical spaces* and are concretized in Section 2.5[☞84]. This bidirectional mapping between the technical spaces is done by *adapters*, which are developed during this use case.

Frequency of Use Cases

It makes sense to distinguish the introduced use cases in this way, since the functionality of these use cases is needed with different frequency: For each project, the use case for specifying consistency is done only *once* and these specifications are *often* used for executing the use case for fixing inconsistencies after each use of a view. A new adapter is developed only once for each new technical space to support, since adapters can be reused for all projects.

User

Derived Stakeholders

Methodologist

Platform Specialist

Adapter Provider

The first use case to automatically fix inconsistencies is triggered by stakeholders of the system development after they manually changed their views. These stakeholders are grouped under the term *users* of consistency management (Section 2.4.1). Since specifying consistency for the second use case requires knowledge about all views and the domain of the system and users usually do not have this knowledge, another stakeholder is responsible for realizing this use case, called *methodologist* (Section 2.4.2[☞81]). Since fixing inconsistencies as first use case should be automated, a realizing conceptual approach and supporting software system must be developed: This requires researchers called *platform specialists* as additional group of stakeholders, since they have knowledge about consistency challenges independent from the application domain, while this knowledge cannot be requested by users and methodologists focusing on single projects. These three stakeholders, users, methodologists and platform specialists, are shortly mentioned by Meier, Werner et al. (2020) already, but are elaborated in the following sections. An additional, fourth stakeholder is dealing with the technical heterogeneity of views and is identified by the third use case to develop adapters: In order to bridge different technical spaces of different views, the *adapter provider* develops adapters, which can be used for all following projects by methodologists.

Use Cases guided by Platform Specialists

All three use cases in Figure 2.20[☞79] are not actively executed, but supported by platform specialists: Since platform specialists develop the approach for consistency management and its implementing framework, they provide guidelines and APIs how to specify consistency for methodologists, they determine the automated execution of fixing inconsistencies for users and specify the supported technical space, for which adapter providers have to implement adapters from other technical spaces.

A forth use case ("Initialize SU(M)M") is introduced in Section 5.2.3[☞176], but not shown in Figure 2.20[☞79], since it is not required for managing inconsistencies in general, but depends on design choices.

2.4.1 User

Users are the stakeholders, who benefit from ensuring inter-model consistency automatically, as defined in Definition 17:

Definition 17: User (Stakeholder)

Users read and write single views and expect them to be consistent to all other views before and after their manual changes.

Users are all stakeholders of a project which use one or more of its views. Before using a view, they expect, that views presented to them are consistent with the other views. Using a view includes reading information and writing information and is done often via tools with fixed viewpoints or with concrete syntax. In case of writing, the user causes changes in the view in order to update the underlying system. Since only the current view is changed, it might become inconsistent to the other views. Therefore, the users expect, that the views are made consistent automatically after their manual changes, i. e. their changes are propagated accordingly into all other views. Summarizing, users execute the following activities:

Users use Views and want automatic Fixes for Inconsistencies

- users read and write single views
- users request automated fixes for possible inconsistencies after their changes

Meier, Werner et al. (2020) call this stakeholder "developer", which is fine in the area of software development, but is not generalizable to domains outside of software development. Even within software development, there are stakeholders falling in the group of users like requirements engineers and software architects, who are using views like requirements specifications and UML diagrams and expect them to be consistent to each other. Therefore, the developer is renamed to user here. In the context of views representing classical models like UML models, the user is sometimes called "designer", e. g. by Demuth, Lopez-Herrejon and Egyed (2015). Here, the term user is used, since it is more general for arbitrary application domains than the mentioned alternatives.

related Terminology

2.4.2 Methodologist

Methodologists are the stakeholders, who automate the consistency for a concrete project using an approach for consistency preservation, as defined in Definition 18:

> Definition 18: **Methodologist (Stakeholder)**
>
> Methodologists apply approaches for consistency preservation in order to realize the automated consistency preservation desired by users.

Initially, methodologists identify and specify the consistency goals and their consistency rules for the current project, together with involved users and perhaps further domain experts. Note, that the understanding of consistency might be subjective depending on the particular users and further stakeholders (Branco, Xiong et al., 2014, p. 933). The identified consistency rules are realized technically using the provided concepts of the chosen approach in order to ensure the consistency goals automatically. Therefore, methodologists need knowledge about meta-modeling, since consistency goals and consistency rules are defined on the viewpoints respectively metamodels, and about the application domain, which could be supported by domain experts or users. This contains also legal issues as precondition for data integration, which are out of the scope of this thesis (Section 1.3.2 ☞43). Additionally, methodologists have to unify the technical spaces of different viewpoints by using adapters. Summarizing, methodologists execute the following activities:

Methodologists realize Consistency Management

- methodologists identify the desired consistency goals within projects
- methodologists specify consistency rules for the collected consistency goals
- methodologists realize consistency goals and their consistency rules using an approach and framework for consistency management
- methodologists use adapters to bridge different technical spaces of different viewpoints

related Terminology

When identifying overlaps of heterogeneous views in collaborative way, Bennani, El Hamlaoui et al. (2018) propose a similar role having knowledge about the domain, its semantics and meta-modeling, called "semantics expert" there. In a following paper (El Hamlaoui, Bennani et al., 2019), a similar role is usually called "expert" (and once "integrator expert"). Vara Larsen, DeAntoni et al. (2015) call the stakeholder responsible for coordinating different models to each other as "integrator". Nentwich, Emmerich and Finkelstein (2003) call the stakeholder to select and customize possible fixes for detected inconsistencies as "repair administrator". Here, the term methodologist is used, since it was coined and agreed upon along with other SUM approaches in Meier, Klare et al. (2019). Additionally, these findings show, that the role of the methodologist is identified as senseful by related work.

User vs Methodologist

Users and methodologists must be distinguished, since users usually know only their own views and have no knowledge about the other views. But the knowledge about inter-view consistency issues is required to solve them, which requires to have methodologists. The other main difference between these two groups of stakeholders is, that users work with their views *often*, while methodologists configure the automation of consistency only *once*.

2.4.3 Platform Specialist

Platform specialists are the stakeholders, who develop generic approaches for consistency preservation in multi-view environments, as defined in Definition 19:

Definition 19: **Platform Specialist (Stakeholder)**

Platform specialists design approaches for consistency management and implement frameworks which support methodologists during their application for concrete consistency problems in projects.

Platform Specialists develop Approaches for Consistency Management

Platform specialists solve classes of consistency problems with conceptual approaches and provide technical frameworks, languages or libraries in order to support methodologists during the application those approaches. The technical support in form of a framework defines at least one technical space, for which adapters as bridges to other technical spaces are required. Summarizing, platform specialists execute the following activities:

- platform specialists design approaches for consistency management
- platform specialists implement such approaches as framework

related Terminology

Meier, Werner et al. (2020) introduce the platform specialist shortly. The word platform can be seen as aggregation of the approach and its implementing framework. As developer of both, platform specialists are specialists for their platforms, i. e. approaches with supporting frameworks. An alternative term for platform specialist could be "researcher". Here, the term platform specialist is used, since it was coined and agreed upon along with other SUM approaches in Meier, Klare et al. (2019).

Methodologist vs Platform Specialist

Methodologists and Platform Specialists must be distinguished, since methodologists apply consistency approaches for each project, since the consistency is specific for the current project, while it is sufficient for platform specialists to develop a generic approach only once. By providing reusable frameworks, platform specialists can help methodologists to save effort.

2.4.4 Adapter Provider

Adapter providers are the stakeholders, who support different technical spaces, i. e. the formats and tools used by users, as defined in Definition 20☞83:

Definition 20: Adapter Provider (Stakeholder)

Adapter providers develop techniques to automatically transform data used by users into the technical spaces used by approaches for consistency management and vice versa.

Adapter Providers ensure automatic Data Transformation between Views and Consistency Management Approaches

Users often use predefined tools, DSLs and data formats for their views, called technical spaces, whose models must be handled by the framework. Therefore, the framework determines at least one technical spaces to be supported by the framework. This selection is discussed in Section 2.5 ☞84 for the new approach of this thesis. Adapter providers develop transformations as bridges between technical spaces of viewpoints used by users and the technical space of the framework of the approach for consistency management. These transformations are bundled as adapters (see Section 6.6.5 ☞226). Summarizing, adapter providers execute the following activities:

- adapter providers develop bidirectional bridges between two different technical spaces

Alternatively, this work could be done by methodologists, if the technical space to support is project-specific. An example is the adapter for XTEXT, which is developed by the methodologist as specialization of the EMF adapter in the application for rights management in Chapter 9 ☞283.

Platform Specialist vs Adapter Provider

Platform specialists and adapter providers must be distinguished, since platform specialists do not know all possible technical spaces and tools whose data should be handled. In particular, data formats like DSLs can be developed explicitly for the current project and must be supported, too, which can not be handled in advance by platform specialists. To support new adapters by adapter providers, platform specialists define mechanisms for developing new adapters in the approach and a corresponding API in the framework.

Ongoing Example, Part 12: Stakeholders ← List →

Mapped to the running example, the four groups of stakeholders can be identified, too, and concretized with involved persons:

User Users are requirements engineers, software architects, programmers and project managers, who use and change only their views (usually they use its representation with concrete syntax for that) and benefit from the automated updating of the other views. End users of the developed software system for university management like students and lecturers are no users in this context, since there is no view to support their needs in this restricted example. In real projects, users want to get the final software for using it, therefore, such a view for the final product after deployment is useful.

Methodologist The methodologist is a person, who has knowledge about the software development project and its desired consistency challenges and has meta-modeling skills, like an expert for quality assurance in software development of the company or of a consulting company.

Platform Specialist is the author of this thesis, since he developed the approach MoConseMI and its realizing framework.

Adapter Provider Since the used formats CSV and EXCEL are provided together with the framework, the platform specialist is also the adapter provider here, who is the author of this thesis.

The design of the new approach (Chapter 6 ☞185) takes these stakeholders into account and supports their concerns by providing tailored ways for application (Chapter 12 ☞455). Section 14.3.1.3 ☞493 discusses the required skills of stakeholders as summary.

As discussed above, views are used by users and viewpoints are used by methodologists. To realize models technically in order to automatically work with them in software tools like model transformation engines, in particular used by adapter providers, Section 2.5 presents the concept of technical spaces, discusses some existing technical spaces and decides to use EMF in this thesis finally.

2.5 Technical Spaces

Technical Spaces provide Techniques and Tooling to realize Models

Models and metamodels are often realized as graphs. Accordingly, model transformations can be realized by graph transformations (Taentzer, Ehrig et al., 2005). From technical perspective, tools using models like model transformation engines require, that the involved models, metamodels and model transformation definitions follow the same technical foundations. They are summarized as *technical spaces*, as defined in Definition 21:

Definition 21: Technical Space

"*A technical space is a model management framework accompanied by a set of tools that operate on the models definable within the framework.*" (Bézivin and Kurtev, 2005)

more than Modeling

More informal, "*the intuitive meaning behind a technical space is a certain technology*" (Bézivin and Kurtev, 2005) to technically realize models including supporting tooling like serialization and deserialization of models and metamodels. Since everything is a model, technical spaces as defined above can realize not only models, but also programming languages and data bases (Bézivin and Kurtev, 2005), ontologies (Kurtev, Bézivin and Akcsit, 2002) or other data like CSV files. In the end, each view is realized by a technical space.

Modeling Space

Djuric, Gaševic and Devedžic (2006) introduce the term *modeling space* as "*a modeling architecture defined by a particular meta-metamodel*" (Djuric, Gaševic and Devedžic, 2006, p. 132). The meta-metamodel like MOF refers to the central concepts which can be used in metamodels. Compared with multi-level modeling, modeling spaces refer to the particular concepts defined in L_1. Djuric, Gaševic and Devedžic (2006, p. 140) use modeling spaces to concretize technical spaces: Objective of a technical space is to realize one modeling space. During that realization and the supply of tooling for this modeling space, further modeling spaces can be used. As an example, MOF is the modeling space of the MDA technical space, that uses also EBNF as modeling space to support programming languages like Java for the generated source code.

Technical Space = 1 realized Modeling Space + n used Modeling Spaces + Tooling

In this thesis, the term technical space is used, since also the particular realization of the main modeling space including existing tooling is reused. Note, that the similar term "model space" is sometimes differently used for other things by different authors, e. g. to distinguish artifacts in the modeling world from artifacts in the programming world (Angyal, Lengyel and Charaf, 2008) or to define a graph of all models conforming to the same metamodel as nodes with model differences between each pair of these models as edges (Diskin, Gholizadeh et al., 2016). The next Section 2.5.1 sketches some existing technical spaces to realize models and metamodels technically.

2.5.1 Related Work: Technical Spaces

sketch existing Technical Spaces

This section sketches some of the existing technical spaces usable to technically realize models and metamodels. Objectives are neither completeness nor a reliable comparison, but to show some existing alternatives. The existence of different technical spaces motivates adapters (Section 6.6.5 ☞226) as bridges between different technical spaces. The

presented selection is restricted to technical spaces which realize models (and not programming languages, as an example) and focuses on technical spaces which are targeted by other approaches for model consistency or metamodel evolution, presented in Chapter 3 ☞93.

Modeling Space	Technical Space
MOF	MDA
ECore	EMF
KM3	KM3
TGraph	JGraLab

- The Object Management Group introduced the Meta Object Facility (MOF) 2.0 (Object Management Group, 2019) to realize (meta)models for their MDA initiative. MOF can be distinguished into the Complete MOF (CMOF) containing all concepts and Essential MOF (EMOF) containing only a subset of concepts. Wachsmuth (2007) presents metamodel adaptations for the complete MOF 2.0.

MOF

- The Eclipse Modeling Framework (EMF) is a Java-based framework to enable modeling with a bunch of tools in the frame of the Eclipse IDE. Within EMF, ECore describes the possible concepts in EMF metamodels and is very similar to EMOF (Steinberg, Budinsky et al., 2009). Differences between EMOF and ECore are discussed by Kramer (2017, pp. 26–28) together with simplified metamodels for the concepts of EMOF and ECore. In terms of Djuric, Gaševic and Devedžic (2006), ECore is the modeling space, which is realized and accompanied by, among others, XMI serialization and Java source code generation within the technical space EMF. EMF is used by multiple approaches, including Gruschko, Kolovos and Paige (2007) for model conformance and Vitruvius (see Section 3.5.2 ☞126) for model consistency. For describing the evolution of metamodels, Vermolen, Wachsmuth and Visser (2012) use a simplified subset of ECore, without enumerations and packages, among others.

ECore

- The Kernel MetaMetaModel (KM3) is a technical space with the purpose to define metamodels (Jouault and Bézivin, 2006). It contains similar concepts like ECore, but not all concepts of ECore. Next to the meta-metamodel defining the concepts of metamodels, KM3 comes with a textual DSL and transformations to bridge KM3 with the technical spaces MOF and ECore. KM3 is used by Cicchetti, Di Ruscio et al. (2008) for model co-evolution and is supported directly by the transformation language ATL (Jouault, Allilaire et al., 2008).

KM3

- The TGraphs approach allows to realize models and metamodels in form of typed, attributed and ordered nodes and edges and is implemented with Java in JGraLab with additional tools for serialization and transformation (Ebert, Riediger and Winter, 2008). TGraphs are used for managing traceability between artifacts of software development (Schwarz, Ebert and Winter, 2010).

TGraph

more Technical Spaces

There are also more formal technical spaces, e. g. *colored petri nets* used for consistency of dynamic UML diagrams (Shinkawa, 2006) or *hypergraphs with constraints* used as intermediate data structure for bridging technical spaces (McBrien and Poulovassilis, 1999). Other technical spaces focus on realizing graphs (as model) without explicit and user-defined graph-schema (as metamodel) like JGraphT (Michail, Kinable et al., 2020). Another variation is to restrict graphs to *trees* as data structure, as in Harmony (Foster, Greenwald et al., 2007). Other research areas enrich modeling spaces with additional concepts like roles for role-based modeling (Kühn, Böhme et al., 2015) or multiple meta-levels for multi-level modeling (Atkinson and Kühne, 2001). More technical spaces can be found in Jelschen (2024, p. 143f), in surveys on model transformations like Jakumeit, Buchwald et al. (2014) and Kahani, Bagherzadeh et al. (2019) and in surveys on workbenches for domain-specific languages like Erdweg, Storm et al. (2013).

Demarcation: UML Profiles

Unrelated to technical spaces is the concept of UML profiles (Pardillo, 2010): While MOF (at M3) allows to model metamodels like the UML super structure (Object Management Group, 2017) (at M2), which is used by developers to model activity diagrams and

state machines (at M1), the profile mechanism is provided by the UML super structure (at M2) in order to support developers to define UML profiles (at M1). Profiles are used by developers to annotate, extend and constrain elements which are modeled in core UML diagrams (at M1). Among others, there are profiles for performance testing (Bernardino, Rodrigues and Zorzo, 2016) and developing hardware and software systems in form of the SysML (Wolny, Mazak et al., 2020), on which other profiles can be defined, e. g. for requirements (Maschotta, Wichmann et al., 2019). Atkinson and Kühne (2002a) discuss some hassles with UML profiles and propose to use ideas of strict multi-level modeling to improve and clarify the profile mechanism. On the other hand, Mallet, Lagarde et al. (2010) realize multi-level models as UML profiles. While profiles are not usable as technical space, since they are located on the metamodel level and not on the meta-metamodel level, the suitability of profiles for combining existing view(point)s is reviewed in Section 3.5.4[☞131].

The next section Section 2.5.2 argues, why EMF is chosen as technical space for this thesis. The concept of adapters as designed in Section 6.6.5[☞226] conceptually allows to support more technical spaces than EMF.

2.5.2 Descision: EMF

de-facto Standard

After presenting some existing technical spaces in Section 2.5.1[☞84], this section motivates, why EMF is used to realize models and metamodels in this thesis. EMF and ECore are selected for this thesis, since EMF is a *de-facto standard* for modeling. Following Ehrig, Ermel et al. (2015b, p. 52), EMF became a standard technology for modeling languages.

lots of Tools support EMF

EMF inspired a huge bunch of tools supporting ECore for (meta)models: Already in the frame of Eclipse, Canovas Izquierdo, Cosentino and Cabot (2017) count 55 modeling projects in 2017, whose degrees of maturity are similar to those of non-modeling Eclipse projects in general. This includes, among others, model transformations like Henshin (Strüber, Born et al., 2017) and work benches for domain-specific languages (DSLs) like Xtext (Bettini, 2013) for textual DSLs and Sirius (Viyovic, Maksimovic and Perisic, 2014) for graphical DSLs. In the field of model transformation approaches, Kahani, Bagherzadeh et al. (2019, p. 2372) report, that EMF is the most supported technical space. Additionally, the underlying Eclipse framework eases tool integration (Mohagheghi, Gilani et al., 2013b, p. 633) for plugins of these approaches.

Implementation reuses EMF-based Tools

Focusing on the specifics of the approach of this thesis, choosing EMF is beneficial, too: According to own statement, EMF is suited for detailed data integration (Steinberg, Budinsky et al., 2009, p. 38). Tools of the technical space EMF are realized for the implementation like (de)serialization of (meta)model elements and their identifiers. The technical representation of models reuses parts of the Edapt project (see details in Section 6.2.1[☞193]), which uses EMF. In the application of Chapter 9[☞283], the EMF-based tool Xtext (Bettini, 2013) is reused to convert grammar-based text into (meta)models directly usable for consistency issues.

increasing (re)use of EMF-based Models

Finally, the use of EMF is increasing in academia and industry (Babur, Cleophas et al., 2018): Since the reuse of already existing artifacts in terms of models and metamodels is one of the central challenges of this thesis (see Section 1.2.2[☞36]), EMF is a good choice, since many models and metamodels are realized with EMF.

Summarizing, EMF is chosen as technical space due to its wide use and benefits from reusing existing EMF-based tooling. Therefore, EMF is described with more details in the following Section 2.5.3[☞87].

While EMF is used as technical space for the approach and its implementation, the wide range of existing technical spaces and their use in practice show the need for another requirement:

Requirement R 4: Technical Spaces

The approach must support views realized in different technical spaces.

Since each view could be realized with a different technical space, it must be possible to reuse models encoded in different technical spaces. Since it is impossible to realize all technical spaces beforehand (Section 2.4.4 ☞82), the adapter provider is introduced as important stakeholder role in Definition 20 ☞83, who is able to support new technical spaces. To support adapter providers, the approach must support a mechanism to bridge technical spaces. Later on, this is realized by the concept of adapters, presented in Section 6.6.5 ☞226. As an example, Part 24 ☞276 of the ongoing example shows, how the technical space CSV is supported.

bridge Technical Spaces → Adapters

2.5.3 Foundations of EMF

Since EMF is used as technical space (Section 2.5.2 ☞86), this section presents those concepts and features of EMF, which are targeted by the consistency management or used later by the implementation of the new approach. Some more specific features which are only supported by adapters are introduced during their implementation in Section 8.4 ☞271. The main reference for this section is Steinberg, Budinsky et al. (2009).

ECore as meta-metamodel used in EMF contains several concepts for defining metamodels. The concepts which are supported by the approach, are shown in Figure 2.21 ☞88, hiding all other concepts[2].

Concepts of EMF for Metamodels

All elements provided by ECore start with the letter "E" by convention. All elements of metamodels defined in ECore are organized in `EPackage`s, which can be nested. Main elements, i. e. `EClassifiers`, are `EClass`es representing classes, `EDataType`s representing data types like `String` or `double` and `EEnum`s representing enumerations. Note, that `EEnum` inherits from `EDataType` and not directly from `EClassifier`. Enumerations contain multiple `EEnumLiteral`s having a `name`, a `value` and an optional `literal`. Classes are either `abstract` or non-abstract and can have multiple super classes (and multiple sub classes accordingly).

EClassifier: EClass, EDataType, EEnum

Features of classes are generalized as `EStructuralFeatures` having a `name`, a lower bound and an upper bound. There are attributes in form of `EAttribute`s with an `EDataType` (or an `EEnum`) as `type`. `EReferences` can be compared with unidirectional UML associations, since they have one `EClass` as `type` and allow to navigate only from its containing class to its type. To enable navigation for a `EReference` also from its type to its containing class, it must be combined with another `EReference` using the `opposite` attribute. This `EReference` has the containing class of the first `EReference` as type and the type of the first `EReference` as class. In this way, bidirectional associations are realized by combining two unidirectional `EReference`s as pair.

EStructuralFeature: EAttribute, EReference

Additionally, an `EReference` can be marked as `containment`: Compared with UML, the visualization of a containment reference looks like a composition, in contrast to "normal" non-containment references looking like usual associations[3]. But its impact for the conforming models is slightly different than in UML: Each object within a model realized with EMF, i. e. an instance of an `EClass` represented as `EObject`, must be contained exactly once within another object, except for a root object. This containment is realized by a link conforming to a `EReference` which is marked as `containment` in the metamodel. Therefore,

Containment Tree

[2] Figure 2.21 ☞88 does not mirror the implementation of these concepts in EMF perfectly, e. g. the types of attributes and references are modeled slightly different and a `ENamedElement` generalizing the `name` attribute is ignored, but represents the concepts in conceptual way. The concrete syntax of Figure 2.21 ☞88 uses the concepts of ECore for metamodels.

[3] There is no comparable concept for UML aggregations in ECore.

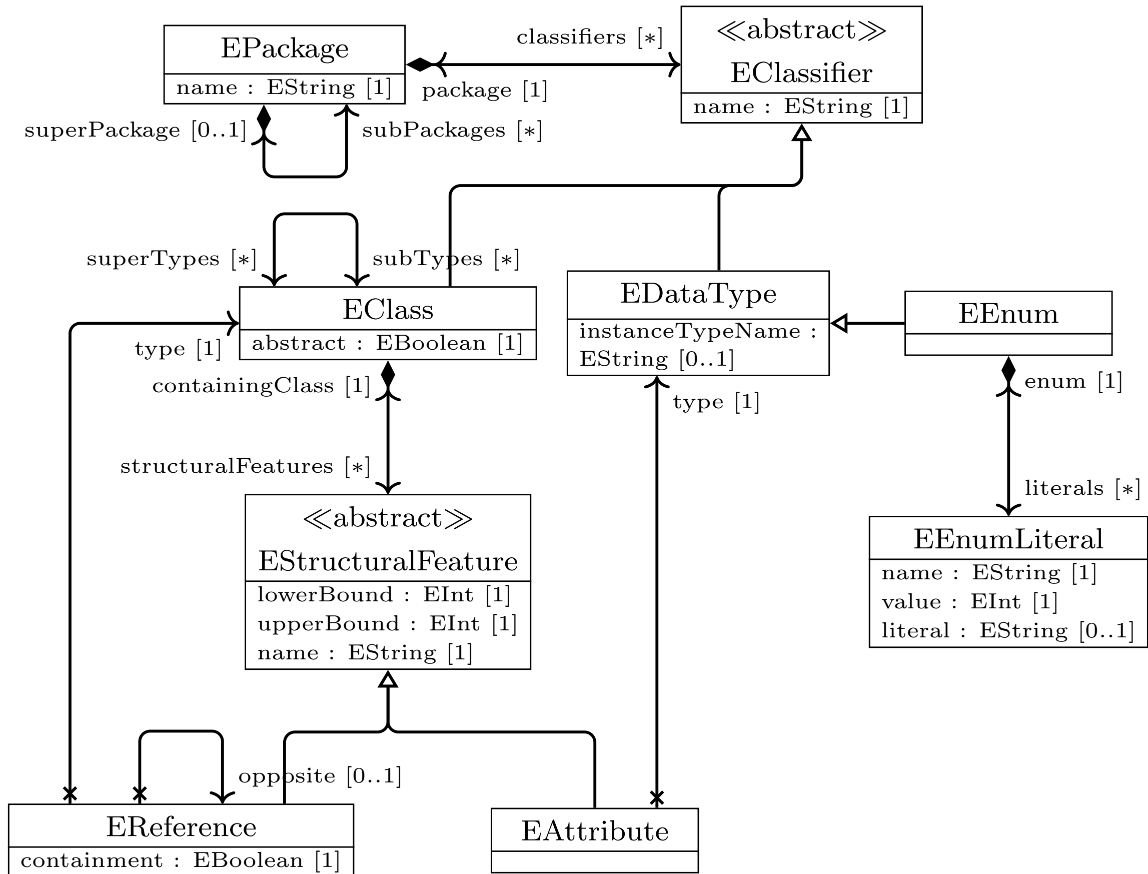


Figure 2.21: Relevant Concepts of ECore

objects form a tree in terms of containment (links) in the model, while these containment references must be provided in the metamodel accordingly. This containment tree is used by EMF for (de)serialization as XMI, since one (or more) root objects are associated to a file (`EResource`) and the root objects and all their recursively contained objects are stored in that file. Additionally, if an object is removed from the model, all its contained objects (the "children") are removed from the model, too. In the case of multiple root objects, they can be associated to different files in order to spread the model over multiple files, which can improve the management of huge models (Jahed, Bagherzadeh and Dingel, 2021).

XMI-IDs

By default, EMF needs no unique identifiers for objects in the model, since they are represented as mesh of objects in memory and a hierarchical key for identifying objects is used for (de)serialization. But as an alternative strategy for object identification, EMF supports to assign textual identifiers (`String`) to objects in the model. During (de)serialization with XMI, these identifiers are used and stored with `xmi:id="myID"` in the resulting XMI files. The in-memory representation of objects does not contain a method like `object.setID("myID")`, but the identifiers are stored within XMI files (`XMIResource` is a special `Resource`) with `file.setID(object, "myID")`. Additionally, it is possible to assign XMI-IDs also to all elements of metamodels. Since the design of the new approach needs unique and stable identifiers (Section 6.6.4 ☞225), this concept of XMI-IDs is used to store identifiers.

static vs dynamic EMF

Models and metamodels can be used with EMF in static or in dynamic way: In *static* mode, Java source code is generated from defined metamodels, which represents the classifiers and their features in Java, e. g. by containing one Java class for each `EClass`. This static Java source code is used at runtime to represent objects of EMF models as Java objects with Java class as type which was generated from the corresponding `EClass`. This

static mode eases programming in Java with the data structures defined in the metamodel, but requires the regeneration of code, if the metamodel evolves. In *dynamic* mode, no code generation is required, since metamodel and model are both represented as mesh of objects at runtime: The classes of the metamodel are represented as instances of `EClass` and the objects of the model are represented as instances of `EObject`[4]. Both Java classes `EClass` and `EObject` are provided by EMF and can be reused without further adaptations. Since the metamodels must be adapted (Section 6.2[☞192]), the implementation uses dynamic EMF later on, but supports also static EMF for data sources (Section 8.4.2[☞273]). Part 13[☞90] of the ongoing example shows, how dynamic EMF is applied to realize parts of the metamodel for requirements.

Motivation for supporting these Features

The motivation for selecting these features are mainly to enable modeling with the main features of UML class diagrams (all `EClassifiers`, all `EStructuralFeatures`) and to enable grouping of elements with `EPackages` in order to simplify the management of huge metamodels (Section 13.3.3.3[☞476]). The amount of supported EMF features corresponds with the amount of developed operators, as presented in Chapter 7[☞241], since they directly work with and change the features of models and metamodels.

some not supported EMF Features

Some popular features of EMF are explicitly not supported, since they are not required for modeling in general, but could be supported later as future work: This includes the concept of proxies when dealing with models which are spread across multiple files. Non-containment links to objects stored in another file initially refer to a proxy object, which is resolved to the real object in the other file, if it is explicitly accessed the first time. This lazy-loading improves performance. Another performance improvement when dealing with huge models could be partial loading, as done by Wei, Kolovos et al. (2016).

Class and Object Diagrams reflect Characteristics of ECore

EMF as described with these concepts and features is used as technical space for this thesis. Therefore, the conventions of ECore are used in the graphics of this thesis for metamodels and models. As an example, `EInt` is shown as data type in diagrams instead of `int` and links within metamodels with filled diamonds are looking like UML compositions, but are containment references in the sense of ECore.

2.6 Summary

Summary of Terminology

Views conforming to viewpoints are suited to provide stakeholders with information about the current system under development tailored to their concerns which are reflected by the viewpoints (Section 2.1[☞54]). Since using multiple views can introduce inconsistencies, the desired consistency in particular projects must be clarified, which is done by introducing the terms consistency goal, which provide single conditions for consistency, and consistency rule, which provide strategies to ensure single consistency goals by fixing corresponding inconsistencies (Section 2.3[☞71]). The involved stakeholders in the process of consistency management are clarified as *users* using their known views in the usual way and expecting automated fixes for occurred inconsistencies, as *methodologists* specifying and realizing these fixes for particular projects, as *platform specialists* developing approaches for consistency management and as *adapter providers* realizing the technical integration of views into consistency management (Section 2.4[☞79]).

Consistency Goals target overlapping Semantics of different Views

Since views are represented as models, modeling terminology is introduced in Section 2.2[☞58], in particular models, whose structures and allowed concepts are determined by metamodels. Metamodels enable to formulate consistency goals and consistency rules once in a general way, which can be used to ensure consistency at any time for all models

[4] This is a strongly simplified description: In reality, special implementations of `EClass` and `EObject` are used, e. g. `DynamicEObjectImpl` having more super classes. Additionally, `EObject` and `EClass` are interfaces in the Java source code of EMF and `EClass` is an (indirect) sub interface of `EObject`.

conforming to these metamodels. Summarizing, consistency goals formulate conditions for semantically overlapping information of different views (Figure 2.16 ☞ 73) on the level of their viewpoints.

Views come with Models and Concrete Renderings

This relationships between terminology are summarized in Figure 2.22. Most important is the understanding of views, as they reduce the whole information of one system under development according to one viewpoint (realizing the concerns of stakeholders) into one model, which is visualized with concrete renderings according to the defined concrete syntaxes of the particular viewpoint. In Figure 2.22, the system of a model is the system of the view of this model (`(Model.usedBy).system`).

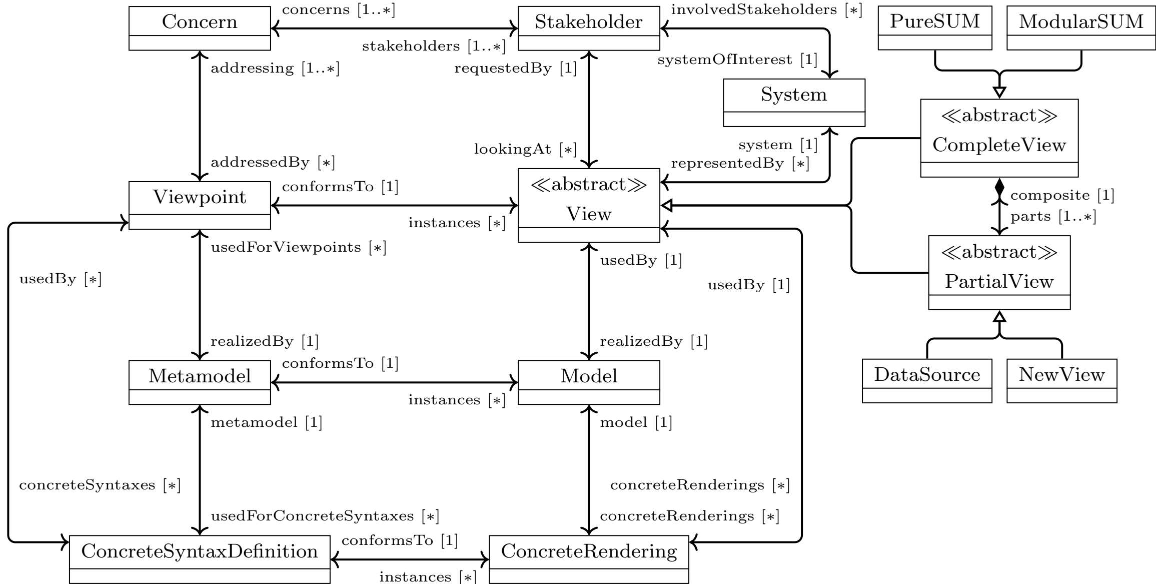


Figure 2.22: Concepts for Stakeholders, Views, Models and Concrete Syntaxes

Important is the distinction between different kinds of views: When discussing the consistency between views, usually *partial views* are discussed, which represent parts of the system under development. Partial views are distinguished into data sources to reuse (Section 1.2.2 ☞ 36) and new views (Section 1.2.3 ☞ 39). *Complete views* represent the whole system under development and contain the information of all partial views on the system under development. Complete views exist at least implicitly, but might be explicitly realized depending on the approach for ensuring consistency, as investigated in Chapter 3 ☞ 93.

Model Transformations and EMF for technical Realization

Model transformations (Section 2.2.3 ☞ 67) enable to work with models in a structured way. In order to technically realize models and metamodels, technical spaces are required and reviewed in Section 2.5 ☞ 84. Finally, the choice of EMF as technical space for this thesis is motivated.

To illustrate the theoretic concepts of the system under development, views and their realization with models and technical spaces given in this Chapter 2 ☞ 51, they are applied to the partial view for the used requirements in Part 13 of the ongoing example:

Ongoing Example, Part 13: **Concepts of Modeling** ← List →

Figure 2.23 ☞ 91 summarizes the modeling concepts as understood and used in this thesis, in contrast to OMG and MLM, arranged in four columns.

The left column (without background color) contains the data for the requirements view(point) of the system under development. Note, that they contain the amount of elicited requirements (one row for each instance) together with an implicit schema (the header row). Part 24 ☞ 276 of the ongoing example discusses that finding in more detail.

Schema and instances are represented by metamodel and model in the neighbored second column as class `Requirement` and object `r1` (`r2` is mostly hidden). The model (ontologically) conforms to the metamodel and correspond to O_1 respectively O_2 in multi-level modeling (see Figure 2.7☞63 in the excursion above). In the OMG model stack, these parts are *both* located in M_1 (see Figure 2.6☞61 in the excursion above).

The other two columns depict the technical realization of the concepts in the second column. The forth column shows the technical space at development time, which is used in the third column to represent the models and metamodels in the second column at runtime: All elements in the metamodel, in particular the class `Requirement`, (linguistically) conform to the `EClass` in the forth column in many-to-one manner. `EClass` is part of the source code of the ECore project and is included as library into the implementation of the framework (Section 6.6.2☞222). At runtime (third column), each metamodel element is represented by one object with (linguistic) type `EClass` in one-to-one manner.

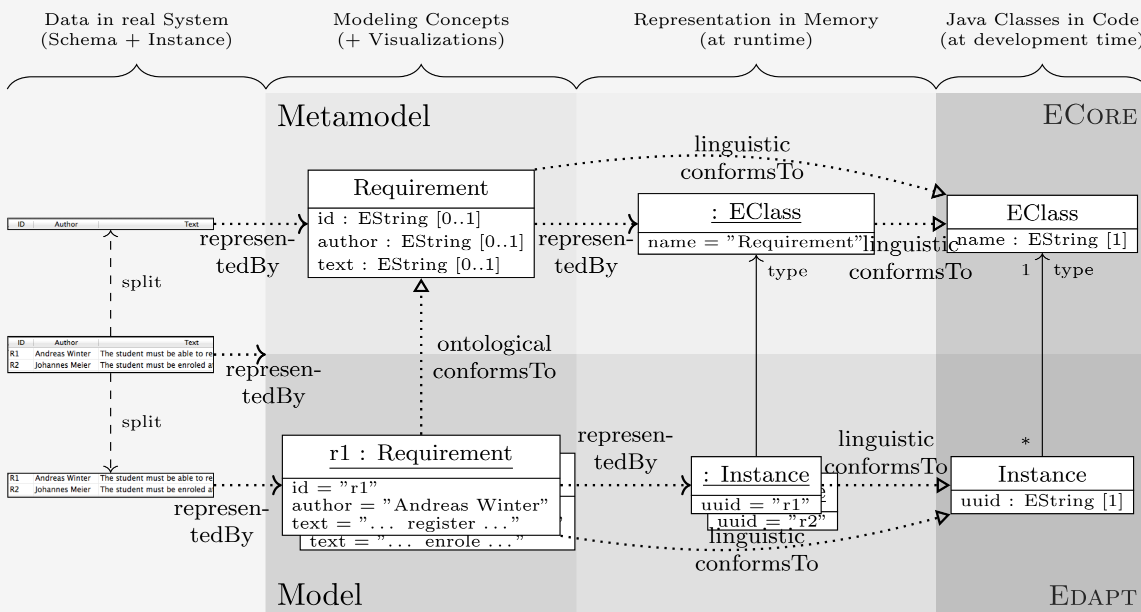


Figure 2.23: Modeling concepts of MoConseMI applied to represent the ongoing requirements

On model level, all elements of the model, in particular all requirements like `r1` (`r2` is mostly hidden), (linguistically) conform to the `Instance` in the forth column in many-to-one manner. `Instance` is part of the source code of the EDapt project and is included as library into the implementation of the framework (Section 6.6.3☞223). At runtime (third column), each model element is represented by one object with (linguistic) type `Instance` in one-to-one manner.

The ontological type in the third and forth columns for the technical realization is realized by the `type` association between the classes `Instance` and `EClass` (forth column) respectively the `type` links between the `Instance`-objects and the `EClass`-objects (third column).

As a preview, the adapters as designed in Section 6.6.5☞226 take the data (left column) at runtime as input and transform them into a mesh of `EClass`-objects for the metamodel *and* into a mesh of `Instance`-objects for the model (third column). These two meshes are input for the first operator.

This terminology and its application establish also relationships between models, views, and so on, which are discussed in this section as megamodels. The lesson learned from dis-

Lessons learned from Megamodeling

cussing them with megamodeling is, that there are (consistency) relationships as challenges to be solved in multi-perspective modeling. In detail, the findings of Figure 2.1 ☞ 52 are the following ones:

- Both relationships μ and δ use the means of abstraction in order to provide reduced models and therefore require synchronization effort, but vary regarding *what* is reduced: μ represents mainly the same information with different renderings in different technical spaces without semantic differences, leading to reduced visual or technical details. δ remains within the same technical space, but reduces information according to semantic purposes.
- When changing models (or metamodels), the *conformance* between models and their metamodels must be ensured, but this must be distinguished from *consistency* between different models.
- Model transformations τ provide means to realize some of the mentioned relationships between models. Therefore, they are introduced in Section 2.2.3 ☞ 67 and investigated as related approaches in Section 3.3 ☞ 108.

Side note: Use of Megamodels

Megamodels are not only used for theoretic discussions about general relations between different models, but are also applied in practice, e. g. for managing big data analyses (Ceri, Valle et al., 2012), software process lines (Simmonds, Perovich et al., 2015) and traceability issues (Seibel, Neumann and Giese, 2010).

In such applications, the megamodels usually do not contain the "elements of the real world" directly, but contain models as representatives for them instead. Since such megamodels are new views on the whole system with additional consistency relations to all other views, megamodels are not used for the realization in this thesis, but only for discussions. Summarizing, "*[m]egamodelling is better seen as a mental discipline than as a technology*" (Stevens, 2017).

After discussing terminology and basics for view(point)s, consistency, modeling and technical spaces, they serve as foundations to analyze existing approaches in Chapter 3 ☞ 93. In particular, the wide range of available technical spaces as sketched in Section 2.5.1 ☞ 84 requires to analyze them regarding approaches for consistency focusing on a particular technical space, which is done in Section 3.6 ☞ 135.

Chapter 3

Related Work

After defining *ensuring of consistency between multiple views* as main objective of this thesis (Chapter 1☞25) and clarifying the terms for consistency and modeling (Chapter 2☞51), this section identifies and discusses existing related approaches for ensuring consistency between views. Objective of the investigations in this section is to learn about related approaches, their characteristics and their suitability for ensuring inter-model consistency. The result of these investigations will be in Section 3.7☞146, that the investigated related approaches fulfill some requirements for ensuring consistency, but not all. This motivates the need for a new approach called MoConseMI (Part III☞163).

Compare related Approaches

Therefore, related approaches are shortly described with their contributions for consistency management and are compared to the requirements, which are directly derived from the challenges for ensuring consistency in Section 1.3.3☞46 and therefore are on high-level:

Requirements for ensuring Consistency

High-level Requirements

- **R 1** Changes in one model have to be propagated into all related models. (**Model Consistency**)
- **R 2** The approach must allow to reuse existing artifacts. (**Reuse existing Artifacts**)
- **R 3** The approach must allow to define new view(point)s. (**Define new View(point)s**)

Since these requirements are on a high level, another contribution of this section are improved requirements: When investigating related approaches, some more challenges for ensuring consistency are found, which are depicted as sub-requirements. Additionally, some more requirements are found, which should be fulfilled by approaches and their realizations on technical level. All requirements are motivated and collected in Chapter 4☞153 as summary.

Another Contribution: Improved Requirements

Due to the huge amount of related approaches in different research areas, this section first applies some strategies to identify and select related approaches for investigation and second applies some strategies to compare these related approaches with requirements. The following *strategies to identify and select related approaches* for investigation are applied:

Selection of related Approaches to investigate

- Section 3.1☞94 identifies criteria to classify the functional objectives of related approaches, i. e. the supported levels of heterogeneity of data to keep consistent, multi-directionality and involved stakeholders during the fix of found inconsistencies. These criteria are used to clarify the focus of selected approaches. These criteria are no direct requirements, but are partially derived from requirements.

Outline is Research Area-oriented

- The following research areas with related approaches are selected for investigation: Since the focus of this thesis is on modeling, related approaches in the modeling domain are investigated (Section 3.3[☞108] – Section 3.5[☞121]). Since software engineering is an application domain of modeling, UML (Section 3.6.1[☞136]) and DSLs (Section 3.6.2[☞137]) are discussed as representatives for languages for engineering, which are realized with modeling techniques. Due to the long history and wide usage of information systems, consistency in terms of data bases (Section 3.6.3[☞139]) and ontologies (Section 3.6.4[☞142]) is discussed. As an example outside of computing science with strong need for managing lots of data, data consistency within enterprises is discussed in Section 3.6.5[☞144], leading to a comprehensive consideration of inter-model consistency in a broad application domain. Additionally, these research areas cover modeling within software engineering, software engineering and information systems within computing science and outside of computing science.
- To show the broadness of existing approaches for managing consistency *beyond these previously selected research areas*, some sections point to some more related approaches outside these research areas, but these are not discussed in detail. These sections are Section 3.5[☞121] and Section 3.6[☞135].

Compare related Approaches with Requirements

The following *strategies to compare related approaches* with requirements are applied:

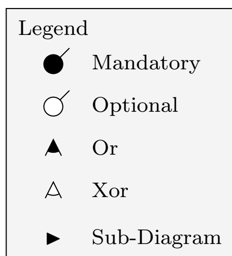
- Related approaches with similar characteristics are grouped together and compared with requirements in group-wise way.
- Not each approach is compared with each requirement, but only the most important requirements are compared, focusing on not fulfilled requirements as limitations of the presented approaches.

Recurring generic Techniques before concrete related Approaches

Since lots of related approaches use similar techniques with different purposes, different frequency or different forms, such recurring generic techniques are described in Section 3.2[☞99], before concrete related approaches are discussed. This allows to focus on the individual characteristics of related approaches with short references to the used generic techniques.

3.1 Criteria for Classification

In order to evaluate existing approaches in a structured way, some early criteria to classify approaches are identified. All criteria are functional objectives of related approaches and are orthogonal to each other. More motivations for these criteria are given directly when discussing them in detail. They are visualized as a feature model in Figure 3.1. The main features, which are numbered, are explained in the following five sections.



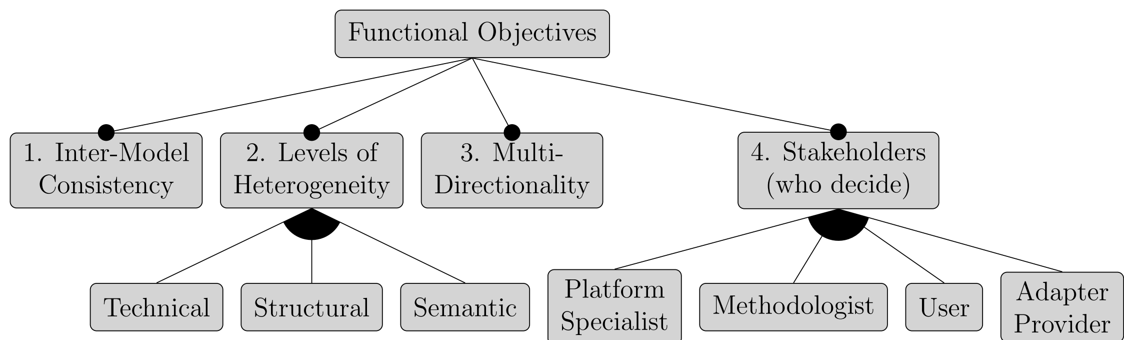


Figure 3.1: Feature Model for classifying functional Objectives of related Approaches

3.1.1 Inter-Model Consistency

The *first criterion* “Inter-Model Consistency” in Figure 3.1 ☞ 94 makes clear, that related approaches must target ensuring consistency between different models conforming to different metamodels. Therefore, this criterion is mandatory. This criterion is already motivated and established in Section 2.3 ☞ 71 and bases on Requirement R 1 (Model Consistency) ☞ 154.

Ensure Consistency between different Models

3.1.2 Levels of Heterogeneity

Since heterogeneity of the involved views and their models is one major problem for consistency management as explained in Section 1.1 ☞ 26, possible *levels of heterogeneity* are analyzed as *second criterion* in order to focus the analysis of related approaches regarding the levels of heterogeneity at which inconsistencies might occur which are fixed by the related approaches. Related approaches must overcome at least one of these levels to be relevant here, therefore, this criterion is mandatory. The three relevant levels of heterogeneity are introduced and discussed in the following:

Levels of Heterogeneity:

Technical heterogeneity covers different representations of the same information in terms of technical aspects like different exchange formats, encodings, or separator signs. The concrete syntax of a view and its embedding into tooling falls into this category of heterogeneity. Additionally, technical heterogeneity occurs, if different data sources use different concepts to describe metamodels, like object-oriented, relational or XML, referring to different modeling spaces, as presented in Section 2.5 ☞ 84. The technical realizations of views are summarized as technical spaces (Section 2.5 ☞ 84).

different technical Representations

Structural heterogeneity covers the aspect, that the same concepts can be described by different metamodels. An example are the different strategies to resolve multiple inheritance into single inheritance.

different Metamodels for same Concepts

Semantic heterogeneity targets conflicts regarding the meanings of different concepts leading to information: Information occurs by interpretation of the available data. These interpretations depend on the context, i. e. the particular project. Doan, Halevy and Ives (2012, p. 92f) give two examples for semantic heterogeneity in details, first different scales of values, e. g. for temperature in Celsius and Fahrenheit or currency in Euro and Dollar, and second the mapping of different names for the same element, e. g. “MDA”, “Model Driven Architecture” and “OMG MDA” refer to the same modeling initiative.

different Meanings of Concepts

Data × Interpretation → Information

In the following, the levels of heterogeneity are applied to the ongoing example to make them clear and to give some concrete examples:

Ongoing Example, Part 14: **Levels of Heterogeneity** ← List →

The data sources of the ongoing example provide challenges regarding the three levels of heterogeneity as presented above:

Technical Since the three data sources are presented to users with different concrete syntax, there is technical heterogeneity, e. g. requirements are realized in CSV format, while Java and class diagrams are realized directly with EMF. The use of different signs for separators in the CSV format is also technical heterogeneity (Leser and Naumann, 2007). But different row numbers for EXCEL (starting with 1) and CSV (starting with 0) is not technical, but semantic heterogeneity.

Structural In the ongoing example, the concepts which are described twice, are classes representing for Java and class diagrams: In the metamodels for both data sources, classes are described by one (meta-)class, but these two classes have different names

(`ClassType` in Java, `Class` in class diagrams), which can be treated as tiny structural difference. Associations and methods are different concepts, which occur only once, and therefore represent no structural heterogeneity. Going beyond this small development project, the designation of classes to be either abstract or non-abstract, could be realized differently: as simple boolean attribute, as attribute with an explicit enumeration having the literal abstract and non-abstract or as sub classes. Examples for different metamodels for the same information are existing in form of different Abstract Syntax Graphs (ASGs) for Java, among others, JaMoPP (Heidenreich, Johannes et al., 2009) and an JavaASG basing on the Java Development Tools (JDT) of Eclipse (Meyer, 2016).

Semantic All consistency goals which are described in Part 11 ☞76 of the ongoing example fall into this category of semantic heterogeneity: Consistency Goal C 2 ☞77 targets the different meanings of the concept for classes in Java and ClassDiagram by requiring, that all classes must be in Java, but in ClassDiagram, some classes might be missing.

Main Focus: semantic + structural Heterogeneity, Adapters: technical Heterogeneity

Summarizing, these levels of heterogeneity help to emphasize the main objective for related approaches to investigate: Ensuring consistency between models regarding *semantic* issues represented by consistency goals is the most important level of heterogeneity here. This fits to Definition 2 ☞32, which requires semantic agreement of models for consistency. To overcome this semantic heterogeneity, the other types of heterogeneity must be solved, too: *Structural* heterogeneity is important, since overlapping concepts are usually realized with different metamodels. In database management, interoperability of heterogeneous data is one of the oldest and most important problems requiring significant amounts of time in practice (Bernstein and Melnik, 2007). The new approach of this thesis focuses on *semantic and structural heterogeneity*, while technical heterogeneity is overcome by the concept of adapters. This criterion is named “Levels of Heterogeneity” in Figure 3.1 ☞94.

alternative Classifications for Data Integration

In the context of data integration in the data base area, Leser and Naumann (2007, pp. 60–78) classify six levels of heterogeneity: Technical heterogeneity as introduced above is distinguished into technical, syntactical and data model heterogeneity in order for a more fine-grain classification and to address the way, how to access and manage the desired data in terms of communication protocols and query languages. Structural heterogeneity as introduced above is distinguished into structural and schematic heterogeneity in order to reflect specific challenges for data base queries. Semantic heterogeneity is defined in coincident way. Since the groups of heterogeneity of Leser and Naumann (2007) fit to the proposed levels of heterogeneity above in general, this shows, that they are reasonable and do not cover specific details of data bases.

In the context of tool integration, Thomas and Nejmeh (1992) extend the classification of Wasserman (1990) (cf. Section 1.3.2 ☞43) regarding different properties of data integration. This classification targets data integration from the perspective of tool integration. Therefore, this classification is not taken as main classification here, but reviewed in contrast to the own classification above. Thomas and Nejmeh (1992) distinguish involved data into persistent data, e. g. the involved data of the system under development, and non-persistent data, e. g. data to synchronize tools running in parallel at runtime. Technical heterogeneity is covered by the terms interoperability (for persistent data) and data exchange (for non-persistent data). Structural and semantic heterogeneity are covered by the terms data consistency (for persistent data) and synchronization (for non-persistent data). The distinction between persistent and non-persistent data is not necessary here, since only persistent data must be kept consistent with other views, while non-persistent data might be used to manage single views at runtime, e. g. to realize the concrete syntax, but are independent from other views, since the stakeholders use their views independently from

other views. Summarizing, the classification of Thomas and Nejmeh (1992) is structured differently, but emphasizes again the semantic consistency between data of different views. The survey of Darke and Shanks (1996) for viewpoint approaches emphasizes the handling of semantic conflicts, too.

De Lara, Guerra and Vangheluwe (2006) distinguish syntactic and semantic consistency: Semantic consistency corresponds to the presented classification for overcoming semantic heterogeneity, while syntactic consistency refers to the abstract syntax, which corresponds to structural heterogeneity here and emphasizes the suitability of the introduced levels of heterogeneity. Additionally, de Lara, Guerra and Vangheluwe (2006) distinguish semantic consistency into static semantic consistency and dynamic semantic consistency, referring to consistency of models describing static or dynamic aspects of the system under development. This distinction is not necessary here, since models with any purposes are targeted here.

3.1.3 Multi-Directionality

As already motivated in Section 1.2 ☞31, targeted by Requirement R 1 (Model Consistency) ☞154 and named as *third criterion* "Multi-Directionality" here, changes in each view must be propagated into all other related views: In general, each view might be changed by the user leading to model changes, which are propagated to other views and each view might receive model changes originated from other views. Therefore, all related approaches must support change propagation in *multiple directions* between multiple involved views. Therefore, this criterion is mandatory. While this criterion is called symmetric organizational dominance by Diskin, Gholizadeh et al. (2016), it is named "Multi-Directionality" in Figure 3.1 ☞94, since this term emphasizes the different directions more.

Multi-Directionality

3.1.4 Stakeholders who decide

After identifying with the previous criteria, for *what* existing approaches need to be evaluated, i. e. the semantic consistency between heterogeneous data in all directions, as *fourth criterion* now the stakeholders of Section 2.4 ☞79 are evaluated regarding their involvement of finding fixes for inconsistencies, i. e. *who* decides about fixes for inconsistency. These involvements can be seen as concerns of the stakeholders regarding (in)consistencies. In order to fix inconsistencies, fixes must be identified and selected, which makes this criterion mandatory.

Which stakeholders decide?

Platform Specialists decide by integrating their decisions directly into the developed approach. Such decisions are fixed and must be used for all projects and applications domains. Examples are hard-coded heuristics used in graph repair (Sandmann and Habel, 2019) or bidirectional transformations like least change (Abou-Saleh, Cheney et al., 2018).

Methodologists should decide on inconsistencies which are project-specific (and which cannot be handled by platform specialists) and which automatically provide fixes for such inconsistencies (which should not be decided again and again by users).

Users should decide on possible fixes for inconsistencies, when there is no unique solution for the particular situation or there is no automatable solution. Users should not decide, if the desired unique solution can be found in an automatic way.

Adapter Providers provide adapters to bridge technical spaces in order to support existing tools, DSLs and data formats, but are not directly involved in consistency issues. Therefore, adapter providers are usually not discussed for consistency issues anymore.

Demuth, Lopez-Herrejon and Egyed (2015, pp. 580–582) summarize fixes by platform specialists and methodologists as "automated fixing" and motivate its need with large numbers of inconsistencies. Additionally, they propose "guided fixing" to support inconsistencies resolved by users with providing possible fixes and information about depending consistency goals. This shows, that the discussed involvement of stakeholders is in line with other research. Involved stakeholders for fixing inconsistencies are analyzed for the ongoing example now:

Ongoing Example, Part 15: **Which Stakeholders decide?** ← List →

In the project of the ongoing example, there are different consistency issues (Part 11 ☞76 of the ongoing example), which must be ensured. If one of these consistency goals is hurt, possible fixes for the resulting inconsistencies must be identified and applied by someone. Now this box discusses, which stakeholders should be responsible for identifying such fixes:

Platform Specialists have only very little chances to provide useful fixes for the consistency goals here, since they are project-specific, while platform specialists design approaches for managing consistency in general way to be applicable for diverse projects and application domains. A small example can be identified for Consistency Goal C 1 ☞76 nevertheless: If there is a traceability link L between requirement R and Java method M and R or M is deleted, then the traceability link L must be deleted, too (corresponding to Consistency Rule C 1 b ☞77). This issue is a generic one and must be solved automatically by approaches, if it is technically realized with an association connecting the classes `Requirement` and `Method` with each other, due to modeling foundations, since links cannot exist without their connected objects. Therefore, this case targets also structures (and not only semantics) and can be solved by platform specialists.

Methodologists should identify and select fixes for inconsistencies according to all consistency goals (except for the cases discussed for the other stakeholders), since the consistency goals are project-specific, which excludes platform specialists, since they can fix only inconsistencies which are valid for all kinds of projects. These consistency goals could be managed by users of course, but since they are *automatable*, users would have to decide on similar inconsistencies again and again, which adds accidental complexity to users. Instead, methodologists should decide once, how these inconsistencies should be fixed.

Users should decide on possible fixes for inconsistencies, when there is no unique solution for the particular situation or there is no automatable solution: This is the case for the Consistency Goal C 1 ☞76, since in the ongoing project, there is no heuristic to decide, if a method fulfills a requirement or not. Instead, the developers should decide during the implementation, if there is traceability between the developed methods and the requirements (corresponding to Consistency Rule C 1 a ☞76). Developers fall into the category of users.

Adapter Providers are not directly involved in ensuring consistency, but provide bridges between technical spaces, e. g. for the CSV format of the requirements specification.

Focus on automatable and project-specific Consistency to be decided by Methodologists

These investigations show, that the particular consistency goals and consistency rules determine, which stakeholders are responsible for fixing violations of the current consistency goal respectively consistency rule. Since this thesis focuses on consistency issues which are automatable and project-specific, related approaches which support decisions of *methodologists* are most interesting. The ongoing example emphasizes this focus, since its consistency goals are mostly automatable and project-specific to be decided by the methodologist. This

criterion is named "Stakeholders (who decide)" in Figure 3.1 ☞ 94.

Another result of these investigations are relations between stakeholders and levels of heterogeneity: Users use only the technical spaces of their views, but expect automated management of semantic heterogeneity. Methodologists overcome semantic and structural heterogenities according consistency goals which are desired by users. Therefore, consistency goals and consistency rules are coming from the concerns of users, but are formulated and realized by methodologists. Since adapter providers (only) overcome technical heterogeneity, methodologists do not care about technical heterogeneity anymore. Platform specialists provide means for methodologists and adapter providers to overcome all three levels of heterogeneity.

Stakeholders × Levels of Heterogeneity

3.1.5 Summary

Figure 3.1 ☞ 94 summarizes the introduced orthogonal criteria for classifications with their following particular purposes:

Inter-Model Consistency is a required feature for approaches for ensuring consistency, determining the focus on *consistency* challenges between *different models*. This includes both checking consistency and fixing found inconsistencies and conforms to Requirement R 1 (Model Consistency) ☞ 154.

Level of heterogeneity of the data to keep consistent The classification regarding levels of heterogeneity helps to restrict the related work to present approaches which target *semantic consistency*. Since structural heterogeneity is also important, since modeling always involves metamodels which can be different, Section 6.2.1 ☞ 193 reviews related work to deal with structural differences of metamodels, too.

Multi-Directionality is a required feature for approaches for ensuring consistency, as discussed above. Therefore, the focus is on related approaches which support multi-directionality, while other approaches are only sketched, if at all.

Stakeholders who decide on fixes for inconsistencies The classification regarding the stakeholders who can decide how to fix inconsistencies help to evaluate existing approaches, since they often support users or platform specialists, but rarely methodologists.

Before reviewing existing approaches in detail, if they fulfill these functional criteria ("what?"), the next Section 3.2 introduces some general techniques which are used by lots of approaches in order to realize consistency management ("how?").

3.2 Overall Realization Techniques

After identifying functional objectives of related approaches in Section 3.1 ☞ 94, e. g. *what* to ensure and *who* decides on fixes for inconsistencies between data, objective of this section is to identify some overall techniques, *how* to ensure consistency by providing possible fixes for inconsistencies. The first reason for discussing these techniques is, that they show some design choices of the technical solution space for possible approaches. The second reason is, that these techniques are used by various related approaches for consistency management. Therefore, they are introduced only once now in general way. All design choices are orthogonal to each other. Some of these design choices are explicitly discussed by publications, others are included, since they occur in several investigated related approaches. More motivations for these design choices are given directly when discussing them in detail. With this selection, this list of technical design choices is not complete, since techniques used

Objective: introduce technical Design Choices and generic Techniques for realizing Consistency

only by single related approaches might not be included. The design choices are visualized as a feature model in Figure 3.2. Since most techniques are supporting and can be used by approaches, most of the features are optional. Techniques which represent design choices which must be decided are marked as mandatory features. The main features, which are numbered, are explained in the following four sections.

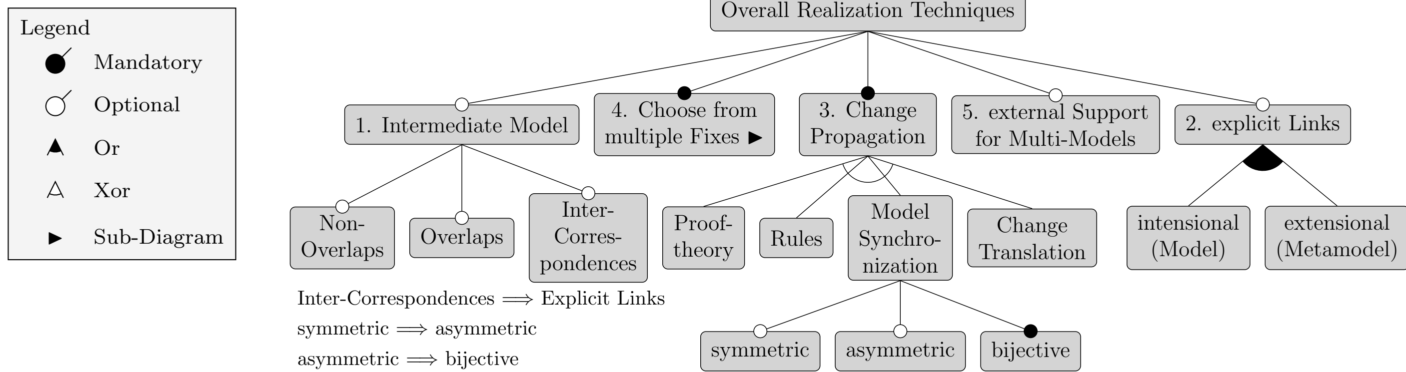


Figure 3.2: Design Choices for technical Realization

3.2.1 Intermediate Model

intermediate Model to store (additional) Information

Main motivation for approaches to use an optional "Intermediate Model" is to store additional information, which are not part of all existing models. The sub-features in Figure 3.2 concretize the content which is stored in the intermediate model, i. e. elements which are related to inter-model consistency issues ("Overlaps") in contrast to elements which are only relevant for exactly one model ("Non-Overlaps", also discussed in Section 13.3.3.2 [☞476]) and explicit links connecting different models ("Inter-Correspondences", as discussed below in the following).

Synthetic vs Projectional

Whether approaches use an intermediate model or not usually depends on, how the approaches manage interrelated models: The IEEE standard 42010 for architecture description (IEEE, 2011) distinguishes approaches for managing viewpoints and their views into synthetic and projectional approaches. *Synthetic* approaches manage interrelations between views directly in a pair-wise manner. *Projectional* approaches introduce a new intermediate structure (called "repository" in IEEE (2011)) and synchronize views only with the intermediate structure. Interrelations between two views are managed indirectly via the intermediate structure as step in between.

Side Discussion: Definitions for Consistency revised

The distinction into synthetic and projectional approaches could be an explanation, why existing definitions in the related work for consistency, as discussed for the Definition 2 [☞32], might miss the strong role of the system: Perhaps, they are focused on synthetic approaches, where the whole system is assembled by the collection of involved views. Therefore, they define the consistency of views to each other. In projectional thinking, the consistency of views to their underlying system is more important, since these relations transitively allow to argue on the consistency of views to each other, too.

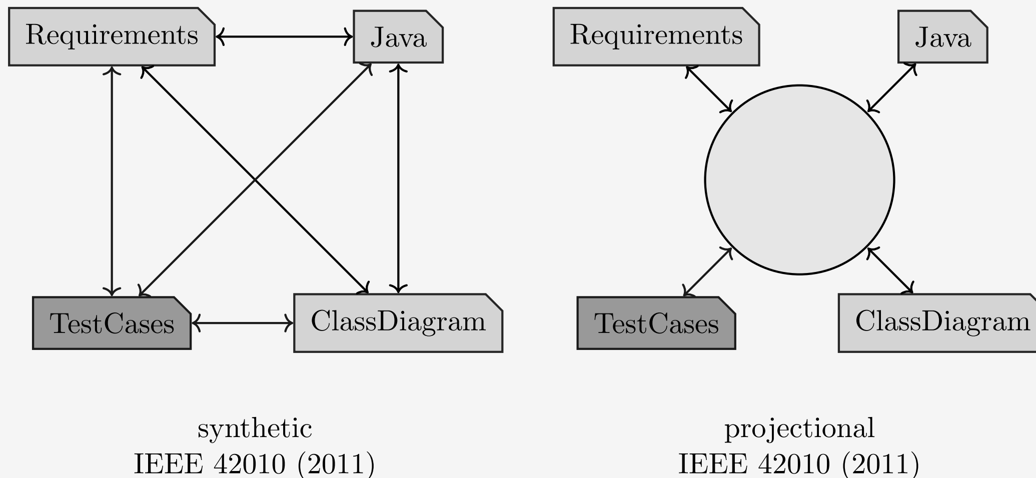
By design, synthetic approaches require to manage a square number of interrelations between all views (Feldmann, Wimmer et al., 2016; Atkinson, Gerbig and Tunjic, 2013a), while projectional approaches need a linear number of interrelations between the views and

the intermediate structure, which is realized as *intermediate model*. If views are inconsistent to each other, the intermediate model of projectional approaches can be used as single point-of-truth to decide conflicts, while synthetic approaches require an "order of importance" or other strategies to decide conflicts.

Synthetic: $\frac{n\cdot(n-1)}{2}$
Projectional: n
(n is the Number of involved Views)

Ongoing Example, Part 16: **Synthetic vs Projectional** ← List →

The following Figure 3.3 applies the presented classification of IEEE (2011) for viewpoint combination to the ongoing example.



synthetic
IEEE 42010 (2011)

projectional
IEEE 42010 (2011)

Figure 3.3: Classification of synthetic vs projectional approaches for the ongoing example

In synthetic approaches (left side), all three data sources for requirements, Java and UML class diagrams (marked in light gray) are synchronized directly with each other. This leads to square effort in general, while in practice the data sources are not fully-meshed. This counts also for the ongoing example, since consistency goals do not target requirements and UML class diagrams directly, as shown in Figure 2.19☞78. In projectional approaches (right side), the data sources are not synchronized directly, but via an intermediate structure.

After adding a fourth data source (marked in dark gray) like test cases, synthetic approaches require to check the interrelations of the new data source to all existing ones (linear effort), while projectional approaches need to interrelate the new data source only with the intermediate structure (constant effort).

Figure 3.2☞100 reflects synthetic and projectional approaches on technical level by the optional feature for an "Intermediate Model", which is used by projectional approaches. It is the natural location to store correspondences between models ("Inter-Correspondences", as deepened in Section 3.2.2) and to store the overlaps of models in order to provide a single point-of-truth ("Overlaps"). Additionally, the intermediate model could contain the information which is relevant only for exactly one model ("Non-Overlaps"). If non-overlaps are contained or not is also discussed in Section 13.3.3.2☞476.

3.2.2 Explicit Links

Often, *explicit links* are used to establish relations between involved (meta)models explicitly. Such links can occur in three terminologies, with similar technical realization, but used for different purpose:

Kinds of explicit Links:

- Such links can be called *correspondencies* connecting two or more elements of different models (IEEE, 2011; Bézivin, Bouzitouna et al., 2006), *intensional* on model level

Correspondencies

or *extensional* on metamodel level as in Romero, Jaén and Vallecillo (2009). If correspondences relate same elements in different models with each other, they can be used for comparing and merging of those models and as an alternative for persistent identifiers (Selonen and Kettunen, 2007).

Model Weaving

- Another kind of explicit links is represented by *model weaving*, "*which consists of establishing correspondences with semantic meaning between model elements*" (Del Fabro, Bézivin et al., 2005). They are stored in a weaving model conforming to a weaving metamodel. Weaving models can also be used to generate model transformations (Del Fabro and Valduriez, 2009; Del Fabro and Jouault, 2005). In particular, weaving is used to compose different models into a single model (as multi-to-one model transformation), e. g. for weaving aspects into a base model in aspect-oriented modeling (Jézéquel, 2008).

Traceability Links

- Traceability approaches use also explicit *traceability links* in order to trace the historic evolution of concepts across different artifacts and development steps, e. g. from requirements over architecture to source code. Traceability links can be stored in a traceability model conforming to a traceability metamodel (Schwarz, Ebert and Winter, 2010). Broy (2018) formalizes traceability and their connected artifacts with logical expressions.

Explicit links are created and maintained by, among others, humans in manual way or automatically by model matching (Bézivin, Bouzitouna et al., 2006) or by model transformation (see the tracing feature in Figure 2.14☞68). Explicit links are used for, among others, understanding relations between models like traceability, as support for propagating changes or even for the formalization and representation of consistency as in Diskin, Xiong et al. (2011). If explicit links are used, they must be maintained regarding changes in the linked models to remain usable for these tasks. Explicit links as depicted in the feature model of Figure 3.2☞100 cover all kinds of explicit links, i. e. correspondences, traceability and model weaving.

3.2.3 Change Propagation

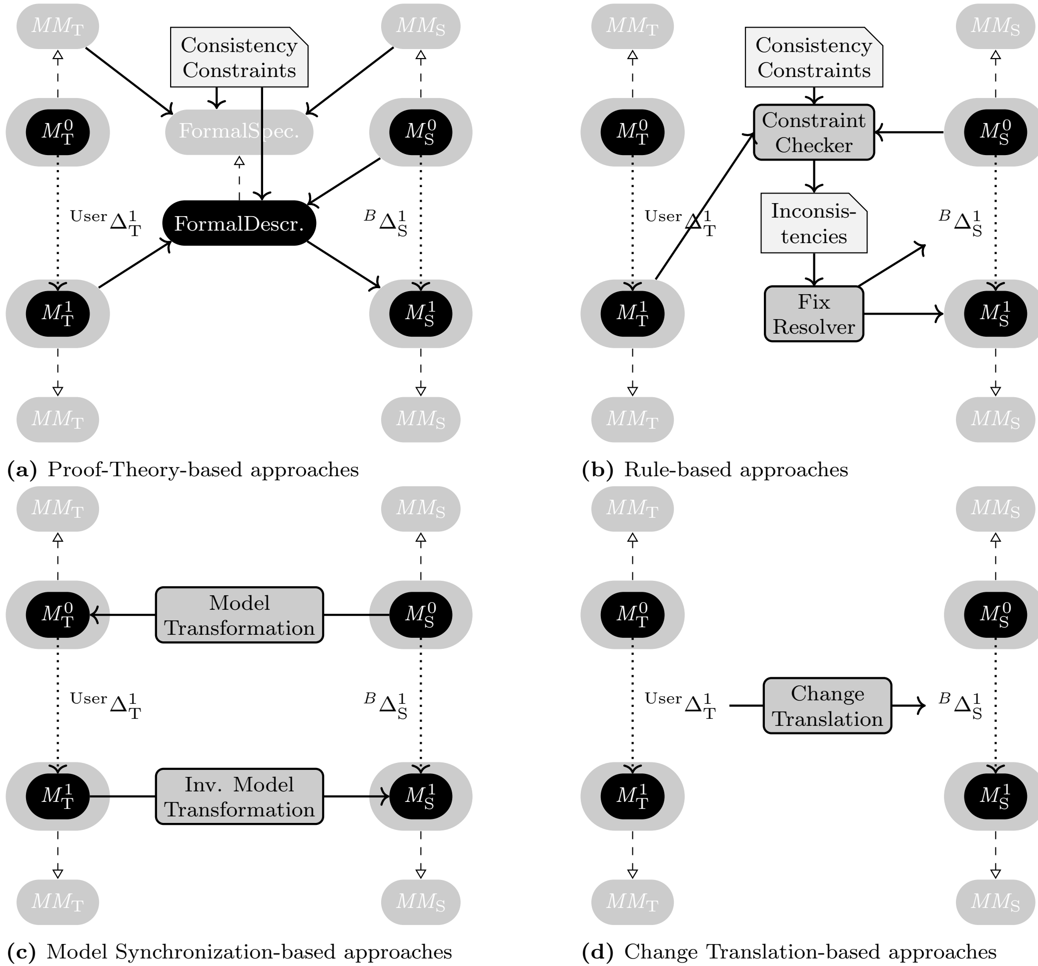
This criteria classifies techniques *how* they ensure consistency by identifying and applying fixes for inconsistencies, which subsumes checking for (in)consistency. The other way around, approaches for checking of consistency without possibilities to fix found inconsistencies are neglected, according to Requirement R 1 (Model Consistency)☞154. Since approaches must ensure consistency to be relevant, this feature "Change Propagation" is mandatory in Figure 3.2☞100.

Realize Change Propagation by …

Approaches for managing consistency, which in particular *propagate changes* into all affected models, are classified by Feldmann, Herzig et al. (2015a) into proof-theory-based, rule-based and (model) synchronization-based approaches. The fourth category of change translation-based approaches is not mentioned by them and added here, since such approaches are identified which do not fit into the three other categories. The general concepts of these four categories of approaches are depicted in Figure 3.4☞103.

… Proof-Theory-based Approaches, …

- *Proof-theory-based approaches* (Figure 3.4a☞103) transform the involved models into formal, well-defined descriptions as expressions or theoretic models (called semantic views by Guerra and de Lara (2006)), e. g. into first-order logic (Finkelstein, Gabbay et al., 1993) or communicating sequential processes (Engels, Heckel et al., 2002). Gabmeyer, Kaufmann et al. (2019) present different formal verification techniques. These formal specifications must be selected or created for the involved metamodels in order to handle the constraints. Additionally, the transformations from models



(a) Proof-Theory-based approaches

(b) Rule-based approaches

(c) Model Synchronization-based approaches

(d) Change Translation-based approaches

Figure 3.4: Conceptual designs of different approaches for consistency management

into formal descriptions require additional computation effort and decrease the performance. Therefore, proof-theory-based approaches are not investigated in detail here, only some approaches are sketched to get an impression for them.

- *Rule-based approaches* (Figure 3.4b) explicitly establish rules in form of constraints whose successful evaluations indicate consistency (by positive constraints) or inconsistency (by negative constraints). The used constraint checkers might be incremental taking the changes of the user into account. The strategies to find fixes for inconsistencies depend on the particular approach and comprise complementing rules with additional resolution rules and automatically calculating possible fixes for selection by users or heuristics, e. g. by state space exploration (Feldmann, Herzig et al., 2015a). Depending on the particular approach, fixes can be whole models or model differences. Since the rules are evaluated on the current models, no transformations into formal descriptions are needed, which reduces effort at development time and at runtime. Instead, the used constraint approaches must support constraints targeting multiple models (for workarounds, see below), preferred in incremental way (Diskin and König, 2016). Rule-based approaches are investigated in the following.

. . . Rule-based Approaches, . . .

- *Model synchronization-based approaches* (Figure 3.4c) use out-place, exogeneous model transformations to define how elements of the first model are related to elements of the second model (Feldmann, Herzig et al., 2015a). Feldmann, Herzig et al. (2015a) called them “synchronization-based”, while they are called “model synchronization-based” here, since this term emphases the use of model transforma-

. . . Model Synchronization-based Approaches or . . .

tions, relates directly to the corresponding research area of model synchronization and is an usual intent of model transformations as investigated by Lúcio, Amrani et al. (2016, p. 655). Since model transformations are able to create consistent counter-parts in the second model for model elements in the first model, model synchronization-based approaches directly include capabilities for fixing inconsistencies in a natural way: The changed model plays the role of the source model and the related models to update are the target models for such model transformations. Since the roles of changed and related models can be switched, bidirectional model transformations are important to enable synchronization in both directions.

bijective vs asymmetric vs symmetric

Since model transformations can automatically create only information in the target model which is available in the source model, Figure 3.5 compares the possible kinds, how information encoded in a source model (S, always right) and in a target model (T, always left) can overlap (Diskin, Gholizadeh et al., 2016): In the *bijective case*, source model and target model contain the same information, only their structure might be different. The bijective case rarely occurs and often makes less sense (Stevens, 2010), in particular here, since different views conforming to different viewpoints are usually tailored to different concerns requiring different information. In the *asymmetric case*, all information of the target model is contained in the source model, while the source model has more information which is not transformed to the target model. The asymmetric case is interesting, when new views are derived from a base model. In the *symmetric case*, some information from the source model is transformed into the target model, while source model and target model have additional "private" information which is not involved in the model transformation. The symmetric case is important to keep two models consistent to each other, which are not completely transformable from each other. These three kinds are complete, since model transformations are not applicable in cases without information overlap ($T \cap S = \varnothing$) and the asymmetric case counts also for switched source and target models. This classification is mostly orthogonal to the kinds of dependencies and therefore specific for approaches which use model transformations: Consistency can cover information which is contained in both models like redundancies on the one hand. On the other hand, consistency can relate information which is contained only in one model to information which is contained only in the other model, e. g. by constraints or explicit links. In the latter case, these explicit links can be seen as additional information which is contained in none of the two models. The expressiveness of model synchronization-based approaches depends on the supported kinds of information overlap, as depicted as features for model synchronization-based approaches in Figure 3.2 ☞100.

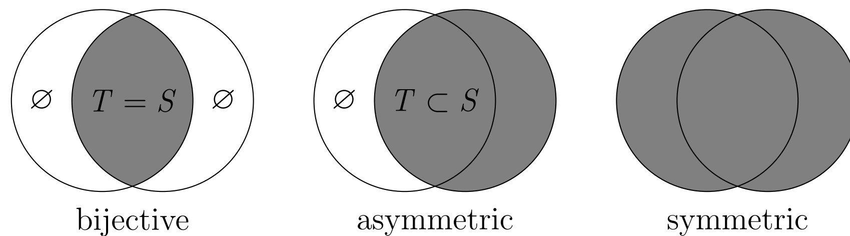


Figure 3.5: Kinds of information overlaps between source models (S, always right) and target models (T, always left) as Venn diagrams

In non-bijective cases, i. e. in asymmetric and symmetric cases, the non-overlapping information which is not covered by model transformations must be preserved, otherwise model transformations become lossy (in terms of Kurtev (2008, p. 383)). To keep

manual changes in the target models, incrementality-related features of model transformations can be exploited to be lossless (see Figure 2.14☞68). Additionally, incrementality can improve the performance of model transformations. Therefore, incrementality is a crucial dimension of model synchronization-based approaches (Diskin, Gholizadeh et al., 2016). Model synchronization-based approaches are investigated in the following.

- *Change Translation-based approaches* (Figure 3.4d☞103) translate changes in the first model directly into corresponding changes for the second model: Instead of using model transformation, updates in form of changes made in the first model are directly propagated by converting them into changes which are applicable for the other models in order to update them according to the updates in the first model. Often this change translation can improve the performance, since the amount of particular model changes is usually much lower than the amount of all model elements. Therefore, change translation-based approaches are investigated in the following.

... Change Translation-based Approaches.

In the context of consistent UML diagrams, Knapp and Mossakowski (2018) use a different, but not contradicting classification by emphasizing rule-based approaches in general: Model-synchronization-based approaches are called “heterogeneous transformation” approaches, while proof-theory-based approaches are distinguished into “system model” approaches and “universal logic” approaches: System model approaches use one uniform language to describe all semantic aspects of the different views, like xUML and fUML for UML. Universal logic approaches use one uniform formal technique into which all semantic aspects of the different views are converted, like transition systems to cover all semantics of UML. This distinction of proof-theory-based approaches is also done by Usman, Nadeem et al. (2008). Similar to Feldmann, Herzig et al. (2015a), change translation-based approaches are not covered by these two classifications, but Knapp and Mossakowski (2018) introduce “dynamic meta-modeling” as additional category, whose approaches extend the metamodels of the views with semantic information, which results in a integrated metamodel for all viewpoints. This last category is very similar to the chosen approach in Chapter 5☞163, but is subsumed under model synchronization-based approaches here. Approaches basing on “*human-centered collaborative exploration*” (Spanoudakis and Zisman, 2001) are not investigated in this thesis, since they cannot be automated due to their involvement of human stakeholders into the identification of inconsistencies.

different Classifications

Snoeck, Michiels and Dedene (2003) distinguish three strategies for managing consistency: *Consistency by analysis* takes the models with user changes and searches for inconsistencies by checking constraints. *Consistency by monitoring* monitors all models and immediately rejects changes which introduce inconsistencies. *Consistency by construction* generates corresponding consistent elements for related models from changed elements. This last strategy can be distinguished into passive, i. e. related models are informed about changes and construct corresponding elements themselves, and active, i. e. the changed model creates corresponding elements for all other models (Haesen and Snoeck, 2005). Compared with the classification above, proof-theory-based and rule-based approaches manage consistency by analysis, while model synchronization-based and change translation-based approaches manage consistency by (active) construction. Consistency by monitoring is not relevant here, since it would require technical facilities for ongoing monitoring, which might require the technical adaption of existing tools. Additionally, immediately fixing inconsistencies is not useful in practice (Stevens, 2017). Since existing tools should be kept as they are, this strategy is not applicable for this thesis. Instead of immediately handling each change, bundled model changes at explicitly defined points in time are used for managing consistency.

Lämmel (2016) call model synchronization-based approaches as *co-transformation* and change translation-based approaches as *co-transformation with delta*, since he identified

these kinds of transformations as generalized patterns which occur not only in model consistency settings.

In the context of model synchronization-based approaches, when formalizing round-tripping properties, Hettel, Lawley and Raymond (2008) use the term “change translation” not for the direct translation of changes as defined here, but for the generic change propagation.

Hearnden, Lawley and Raymond (2006) call the strategy of using (only unidirectional) model transformations as *re-transformation* and change translation as *live transformation*.

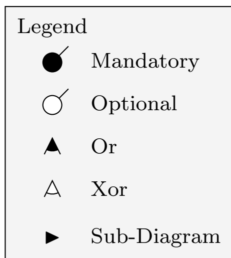
valide Approaches with each other

If there is support for both model synchronization and change translations, both approaches can be applied in order to evaluate each other, e. g. the resulting model of the backward transformation must be the same as the model which is the result of applying the translated changes to the initial source model (Hearnden, Lawley and Raymond, 2006). This strategy can be generalized by applying each combination of two of these approaches in order to validate them.

3.2.4 Choose from multiple Fixes

Design Choices for choosing Fixes for Inconsistencies

Since there are multiple possible fixes for each inconsistency in general, as found in Section 2.3☞71, approaches for ensuring consistency often provide multiple of the possible solutions to fix an occurred inconsistency. Therefore, design choices to select one of multiple possible fixes are mandatory and are discussed along Figure 3.6, which complements Figure 3.2☞100. The presented classification summarizes the results of a survey for model



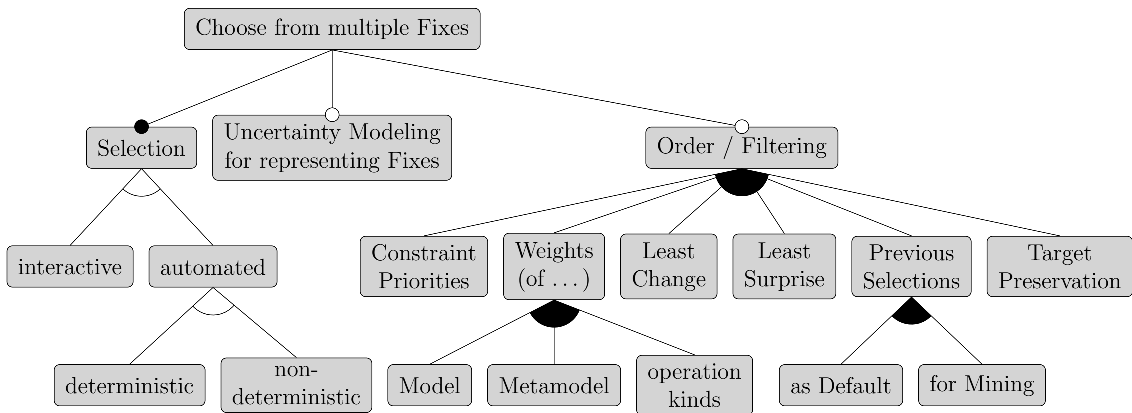


Figure 3.6: Design Choices for selecting one of multiple possible Fixes

repair approaches (Macedo, Jorge and Cunha, 2017, p. 629f) and of design space analyses for model synchronization (Antkiewicz and Czarnecki, 2008, p. 38), but clearly separates the particular selection from heuristics for ordering and filtering fixes before. While the special cases of having some meta-data for heuristics is ignored here, all other features from literature including least change and least surprise which are important for BX (Cheney, Gibbons et al., 2017) are covered here: The particular *selection* of one solution ...

Selection: automated vs interactive

- ... can be *interactive*, i. e. one of the stakeholders introduced in Section 2.4☞79 decides for each occurred inconsistency. With growing number of decisions and possible fixes, appropriate tool support (Mussbacher, Combemale et al., 2020) for deciding is useful.
- Otherwise, the selection is *automated* without involved humans by an algorithm, which can be either *non-deterministic* or *deterministic*, i. e. the decision can be predicted. Note, that deterministic selections are possible, even with multiple possible fixes for inconsistency (Stevens, 2018, p. 7).

Hybrid selections using automation where possible and interactive elsewhere are possible, too, as suggested e. g. for model merging (Dam, Egyed et al., 2016). Non-deterministic selection is used e. g. by graph repair (Sandmann and Habel, 2019), a proof-theory-based approach using typed graphs, graph grammars, graph formulas and graph programs in order to repair graphs regarding graph constraints to fulfill. This graph repair approach can be used also for meta-modeling with ECore (Sandmann, 2020).

Since three different stakeholders are involved in ensuring consistency according to Section 3.1☞94, the distinction between automated and interactive can be done for each stakeholder: As an example, the platform specialist can decide not to automate some inconsistencies, since their fixes might be project-specific. Now the methodologist has to decide and chooses to provide an automation, since the particular inconsistencies can be solved automatically for the particular project setting. Otherwise, the user had to provide an automated or interactive strategy for fixing. Section 3.1☞94 already discussed, that approaches which allow methodologists to manually determine fixes once are more helpful here.

Selection by different Stakeholders

To support these selections, the amount of possible fixes can be *ordered and/or filtered* before the selection by using and combining various single heuristics, as listed now. Some more ideas for combinations and heuristics are given by Macedo, Jorge and Cunha (2017, p. 617).

Ordering/Filtering with various Heuristics

- Since some fixes might not fix all occurred inconsistencies but only a subset, fixes can be ordered according to *priorities of corresponding consistency constraints.*
- The presented fixes can be *weighted* e. g. with cost functions according to the affected parts of the *model* and of the *metamodel* and to the *kinds of change operations* used in the fixes like create, change, delete. Such weights could be used to realize higher-level heuristics including least change, least surprise and target preservation:
- *Target preservation* ensures that parts of the target model are preserved, e. g. by increasing the costs for deletions in the target model.
- *Least change* uses metrics to minimize the amount of change operations in fixes, i. e. least change ensures, that the inconsistency is fixed, but nothing more.
- *Least surprise* prefers fixes which minimize unexpected disruptions in the models, but which are not always of minimal size (Cheney, Gibbons et al., 2017).
- Finally, *previous selections* can be reused as source for *default* fixes and for *mining* fixes with techniques like machine learning with an example in Barriga, Mandow et al. (2020).

In order to make different fixes and their resulting models explicit, *uncertainty modeling* (Troya, Moreno et al., 2021) can be used (feature “Uncertainty Modeling for representing Fixes” in Figure 3.6☞106): Since the details of the final models after fixing inconsistencies are unclear, the case of design uncertainty can be used to model all possible options explicitly. This can be done by specifying confidence values for all models elements like objects and links (with strategies of occurrence uncertainty) and for all primitive values of slots for attributes (with strategies of measurement uncertainty). For UML and OCL, Bertoa, Burgueño et al. (2020) extend the primitive data types (e. g. `UBoolean` for `Boolean`) with an additional `Real` value in the range $[0, 1]$ indicating the confidence value of the modeled primitive (here: boolean) value. These confidence values can reflect also the order of fixes determined by the used heuristics. Most approaches do not explicitly use such techniques or only argue using uncertainty terminology. As an example for an approach which explicitly involves uncertainty, Salay, Gorzny and Chechik (2013) model uncertainty of modeled elements directly within the models with special annotations. Basing on that explicit uncertainty information, propagation of changes affects also uncertainty information and can

modeling Uncertainty

be triggered also by changed uncertainty values. Since uncertainty of models can also be defined as a set of possible concrete models without uncertainty as done by Salay and Chechik (2015), uncertainty modeling is comparable with other research areas like software product lines, that use variability modeling (Pol'la, Buccella and Cechich, 2021) to make different possible model alternatives explicit.

3.2.5 External Support for Multi-Models

supporting Multi-Models

Other supporting techniques are required, when a modeling technique must cope with multiple models, but the available tools supports only a single model (optional feature "external support for Multi-Models" in Figure 3.2☞100): As workaround, approaches without multi-model-support can be used after nesting the involved models into one container model (Macedo, Jorge and Cunha, 2017, p. 622). An extended version of this workaround is to match same objects and links in the models and merge them in order to unify the models (König and Diskin, 2017). Further approaches directly lead to the question, how views can be composed into a single model, as discussed in Section 3.5.4☞131. Use cases which require multi-model-support include constraint checking in rule-based approaches and model transformations for new views in synthetic settings.

3.2.6 Summary

Outline for the following concrete Approaches

All presented techniques, the various kinds of explicit links and the strategies for change propagation including selection of the final fix, can be used for synthetic and projectional approaches in general. The classification regarding synthetic and projectional approaches is used for structuring the rest of this section: Section 3.3 describes synthetic approaches, Section 3.4☞120 introduces the intermediate model called SUM as prerequisite for the following projectional approaches in Section 3.5☞121, Section 3.6☞135 contains less generic approaches in different application domains and Section 3.7☞146 evaluates the presented related approaches and summarizes the lessons learned from these analyses and evaluations.

3.3 Synthetic Approaches

The main characteristic of synthetic approaches is, that they establish direct relations between pairs of models. If an intermediate model is used, it stores only explicit links between (usually two) models, but usually no overlaps or other parts of the models. Section 3.3.1 analyzes approaches for keeping models in this setting consistent (according to Requirement R 1 (Model Consistency)☞154), while Section 3.3.2☞119 reviews, how new view(point)s can be defined on top of multiple existing source (meta)models (according to Requirement R 3 (Define new View(point)s)☞156), since synthetic approaches usually are restricted to fulfill only one of these two requirements.

3.3.1 Synthetic Consistency Preservation

Synthetic approaches require lots of direct relations between the views for their direct change propagation. In the extreme case of spatial separation of the involved models, peer-to-peer approaches like in global software development (Mukherjee, Kovacevic et al., 2008) fall into the category of synthetic approaches. The projectional VITRUVIUS approach using synthetic techniques is presented in Section 3.5.2☞126. The change propagation along these relations can be realized by *proof-theory-based*, by *rule-based*, by *model synchronization-based* or by *change translation-based* techniques, for which examples are presented in the following paragraphs in this order.

The *proof-theory-based approaches* are only shortly investigated here, due to their restricted scalability, as discussed above. Pinna Puissant, Van Der Straeten and Mens (2015) determine possible fixes for previously identified inconsistencies using regression planning. Since this automated planning is implemented with logic-programming in PROLOG, this approach falls into the category of proof-theory. Since there might be multiple generated fixes called plans, their selection and order can be controlled using customized cost functions. In earlier works, Van Der Straeten, Mens et al. (2003) use description logics, a decidable subset of first-order predicate logic.

Proof-Theory-based Approaches

planning Fixes with PROLOG

Another proof-theory-based approach using PROLOG to support multiple repair plans is presented by Almeida da Silva, Mougenot et al. (2010): Based on the sequence of model differences which represent the current model (Blanc, Mougenot et al., 2009), a depth-first tree search algorithm with depth-limitation and back-tracking identifies repair plans for model differences which introduced inconsistencies. Afterwards, the users manually select one of these generated repair plans.

PROLOG-based searching for Repair Plans on Model Differences

Summarizing proof-theory-based approaches, for which some examples are sketched here, they require transformations of models into formal specifications: While these approaches allow to proof consistency or to derive fixes for inconsistencies on the formal descriptions (Requirement R 1 (Model Consistency) ☞ 154), e. g. by using SAT solvers like in ALLOY (Jackson, 2019), it is complex and requires high effort to define the required formal specifications and to create the required transformations[5] leading to limitations in practice, since they cannot be automatically created in each case due to different semantics of metamodels and desired consistency constraints. This counts also for extensions, when a new viewpoint with additional semantics (Requirement R 3 (Define new View(point)s) ☞ 156) leads to extensions of the already established formal specifications (Knapp and Mossakowski, 2018, p. 47). Therefore, at least for mechatronic manufacturing system, "*ensuring the completeness of such a formal system and, by that, proving the full consistency of models of mechatronic manufacturing systems is difficult if not impossible*" (Feldmann, Herzig et al., 2015a, p. 163). Additionally, the transformations from models into formal descriptions require additional computation effort and decrease the performance. This counts also for the used solvers themselves (Macedo and Cunha, 2013, p. 310).

Summary of Proof-Theory-based Approaches

The *rule-based approaches* differ from each other regarding the ways to check for inconsistency and to determine corresponding fixes. Some examples for such rule-based approach are presented now: Nentwich, Emmerich et al. (2003) introduce XLINKIT for incremental checking of constraints defined in restricted first-order logic on multiple XML documents using XPATH for navigation within XML. For detected inconsistencies, i. e. expressions which are evaluated to false, possible repair actions are automatically detected in order to fulfill these hurt expressions. These repair actions might be restricted and commented by methodologists and are presented to users afterwards, who manually select one of these solutions (Nentwich, Emmerich and Finkelstein, 2003). In order to increase the understanding of found inconsistencies, the evaluation creates and presents hyper-links between the concrete elements in the XML documents which are involved in the current evaluation.

Rule-based Approaches

XLINKIT: first-order Logic Constraints on XML for checking, identify Repairs by fulfilling hurt Constraints

Another example for a rule-based approach is provided by Egyed, Zeman et al. (2018): Project-specific consistency rules are specified with OCL and incrementally executed in order to detect inconsistencies. Found inconsistencies are not automatically fixed, but possible fixes are generated, arranged in form of a tree (Reder and Egyed, 2012) and presented to the users for guidance. To overcome tool boundaries, information which is relevant for consistency is extracted from each tool and provided as model with metamodel at a dedicated server application called DESIGNSPACE (Demuth, Riedl-Ehrenleitner et al., 2015). These models are explicitly linked with each other. The resulting links are used and checked

Rule-based Consistency checking with OCL based on Links

[5]Examples for such transformations are UML class diagrams with OCL constraints (Cunha, Garis and Riesco, 2015) and UML state machines (Garis, Paiva et al., 2012) transformed into ALLOY.

by the OCL constraints. This approach is successfully applied in the area of production automation (Demuth, Kretschmer et al., 2016).

EVL Constraints with imperative Fixes

The Epsilon Validation Language (EVL) (Kolovos, Paige and Polack, 2009) is a DSL with tool support within the Epsilon framework (Paige, Kolovos et al., 2009) and allows to specify and evaluate constraints which might depend on each other on multiple models. Additionally, each constraint can be complemented with an arbitrary number of fixes, which can be selected by users to be executed, when the constraint is hurt at runtime. These fixes (keyword `fix`) contain imperative actions in EOL which must fix detected inconsistencies in terms of hurt constraints. "*EOL is the core DSL in* Epsilon*, providing OCL-like model navigation and modification facilities*" (Paige, Kolovos et al., 2009, p. 164). EVL can be executed in distributed and parallel way (Madani, Kolovos and Paige, 2021), but not in incremental way.

Users select one of the provided Quick-Fixes (generated by Constraint Satisfaction) for Inconsistencies (found by Graph Pattern Matching)

Hegedus, Horvath et al. (2011) support users to select fixes for inconsistencies in form of "quick fixes" inspired from auto-completion of IDEs for programming languages: They use incremental graph pattern matching to identify inconsistencies and a heuristics-guided traversal algorithm with backtracking and cycle detection for state-space exploration, both directly on the current model for better performance. The approach aims to enable users to specify rules for inconsistencies by their own, since corresponding graph patterns can be added (and removed) in flexible way.

Summary of Rule-based Approaches

Summarizing rule-based approaches, they all evaluate constraints for consistency directly on the current (inconsistent) models in similar way using techniques like OCL, first-order logic or pattern matching. The strategies to fix found inconsistencies in form of hurt constraints vary in terms of finding repairs and selecting the final repair, which will be applied in order to fix the model in-place. The final selection is usually done by users, while methodologists sometimes can predefine, restrict or comment possible fixes. Some approaches improve performance by incremental constraint checking.

Model Synchronization-based Approaches: Transformations between Models

The *model synchronization-based approaches* differ from each other regarding the features of the used model transformation approaches (Kahani, Bagherzadeh et al., 2019), as depicted in Figure 2.14☞68. Usually, the used model transformation approaches are out-place and exogeneous in order to relate two models conforming to different metamodels to each other. Some approaches using model transformations for consistency are presented in the following paragraphs, grouped by the main characteristics of model transformation approaches (Figure 2.14☞68).

unidirectional Model Transformations

The first group of model synchronization-based approaches uses *unidirectional model transformations*: The MDA initiative (Section 2.2☞58) established CIM, PIM, PSM and Code as groups of viewpoints for development with high need of consistency between corresponding views, but does not propose techniques for ensuring the required consistency, since mainly transformations from CIM to PIM to PSM to Code are addressed. As an example, such a workflow of model transformations is reported by Sindico, Natale and Sangiovanni-Vincentelli (2012). Accordingly, various unidirectional model transformation approaches including ATL (Jouault, Allilaire et al., 2008) or QVT-O (Object Management Group, 2015) can be used for such synchronizations. In case of bidirectional synchronizations, pairs of unidirectional model transformations can be combined, one unidirectional model transformation for each direction. The drawback of this strategy is, that manual effort is required to ensure, that the two transformations match and do not contradict each other.

bidirectional Transformations (BX)

Up to now, the presented approaches support only forward transformations from higher-level models to lower-level models, in the sense of MDA. Since both involved models for such a transformation can be changed and these changes must be synchronized to the other model, bidirectional model transformations (BX) are often used (Abou-Saleh, Cheney et al., 2018). Bidirectionality is an important concept not only for MDE and model respectively graph transformation, but also for programming languages and data bases (Czarnecki, Fos-

ter et al., 2009). The latter is investigated in general and regarding the view-update problem in detail in Section 3.6.3☞139. Focusing on model transformations here, there are several bidirectional model transformation approaches, as presented by Stevens (2008), by Hidaka, Tisi et al. (2016) and by Anjorin, Buchmann et al. (2020). Some approaches using bidirectional model transformations for consistency are depicted in the following paragraphs, including the approaches TGG and QVT-R as representatives for model synchronization with automated selection and EVL+Strace and JTL as representatives for model synchronization with interactive selection (see Figure 3.6☞106). Samimi-Dehkordi, Zamani and Kolahdouz-Rahimi (2016) also selected exactly these four bidirectional model transformation approaches for their classification. BXtend(DSL) (Bank, Buchmann and Westfechtel, 2021) is investigated as fifth and quite new approach, since it increases expressiveness of BX with automated selection.

Triple Graph Grammars (TGGs)

Triple Graph Grammars (TGGs) realize out-place bidirectional model transformations by relating the source model as graph and the target model as graph to each other by explicit links as correspondence graph, as the third graph (Schürr and Klar, 2008). TGGs use declarative patterns and can be non-deterministically executed in forward direction from source model to target model and in backward direction from target model to source model, leading to bidirectionality with one model transformation definition. Usually, one pattern in the source model is related to one pattern in the target model, but multi-amalgamated TGG rules allow to relate one pattern match in the source model to an arbitrary and dynamic number of pattern matches in the target model (Leblebici, Anjorin et al., 2015). With MoTE (Hasso-Plattner-Institute), TGG Interpreter (University of Paderborn) and eMoflon (TU Darmstadt), there are different model transformation engines to execute TGG model transformation definitions (Hildebrandt, Lambers et al., 2013). Hermann, Ehrig et al. (2011, 2015) showed, that TGGs can be used for model synchronization, since TGGs always ensure consistency between any source model and any target model, if the execution of the particular TGG is deterministic in both directions. Keeping more than two models consistent to each other is possible with TGGs, since TGGs can be extended to relate multiple models to each other (Trollmann and Albayrak, 2015, 2016), but is not supported by TGG engines in practice (Anjorin, Leblebici and Schürr, 2016).

QVT-R

QVT-R is the QVT Relations language, one of the three languages of the QVT standard of the OMG (Object Management Group, 2015). QVT-R allows to specify exogeneous bidirectional model transformations. There are several engines for QVT-R (Kurtev, 2008), including engines build with TGGs (Greenyer and Kindler, 2010), Echo build with Alloy for relational logic with SAT solving for model finding (Macedo and Cunha, 2013), build with XSLT for XML documents (Li, Li and Stolz, 2011) and build as transformations into the executable UML-RSDS (Lano and Kolahdouz-Rahimi, 2021) or into colored petri nets (Guerra and de Lara, 2014). Echo translates ECore metamodels (Cunha, Garis and Riesco, 2015) and ECore models (Macedo and Cunha, 2013) to Alloy and uses SAT solving techniques to identify (target) models which match the QVT-R model transformation definition and additional OCL constraints. Users select one of these possible models as final result of the QVT-R model transformation. Stevens (2010) and Lano and Kolahdouz-Rahimi (2021) identified some semantic issues and open questions of QVT-R, e. g. whether QVT-R supports also non-bijective transformation scenarios. Stevens (2013) proposes clarifications for the semantics of QVT-R. The outdated QVT medini implementation of "QVT-R *can support target-incrementality to provide change propagation, but it cannot preserve user updates in the target*" (Samimi-Dehkordi, Zamani and Kolahdouz-Rahimi, 2016, p. 317). There are attempts to extend QVT-R for multi-directional model transformations in order to support consistency between more than two models (Macedo, Cunha and Pacheco, 2014).

EVL+Strace uses the Epsilon Validation Language (EVL) (Kolovos, Paige and

Polack, 2009, see above) together with a domain-specific trace metamodel to realize exogenous bidirectional model transformations (Samimi-Dehkordi, Zamani and Kolahdouz-Rahimi, 2018): The EVL is used to express constraints in order to detect inconsistencies. The domain-specific trace metamodel prevents invalid trace links and enables to store current values of source and target models together with traces in order to detect changed values later on. Constraints are directed and check consistency between one model and the trace model. Each constraint can be complemented with possible fixes for the trace model and the other model in EOL. If inconsistencies are detected, corresponding fixes are applied automatically or presented to the user for selection in an interactive way.

EVL+STRACE: domain-specific Traces and directed Constraints with Repairs, interactively chosen

The JANUS TRANSFORMATION LANGUAGE (JTL) realizes bidirectional model transformations supporting non-determinism in a declarative way (Cicchetti, Di Ruscio et al., 2011): Model transformation definitions are written in QVT-R-style and are automatically translated into search problems expressed in ANSWER SET PROGRAMMING (ASP). Since also the involved models and their metamodels are automatically transformed into ASP, JTL can be seen also as proof-theory-based, too. An ASP solver identifies all possible models fulfilling the constraints which are induced by the model transformation definitions. The users choose the preferred model as result. To ease the handling with the amount of possible models, they are represented as one model explicitly encoding the alternatives using uncertainty terminology (Eramo, Pierantonio and Rosa, 2015). To enable bidirectionality, trace links between source and target elements are automatically maintained to store information about mappings and deleted elements (Eramo, Pierantonio and Tucci, 2018). Summarizing, JTL uses model finding in terms of (Lúcio, Amrani et al., 2016, p. 654) for realizing model transformations.

JTL (using Proof-Theory)

BXTEND(DSL) aims to increase expressiveness of incremental BX with automated selection, since other BX approaches restrict expressiveness for formal guarantees (Bank, Buchmann and Westfechtel, 2021): The higher-level DSL BXTENDDSL allows to declaratively describe incremental and bidirectional model transformations. In order to complete these definitions, imperative and unidirectional code written with BXTEND must be used to specify details of the desired transformations. The BXTENDDSL definitions are automatically generated into BXTEND definitions, which are complemented with the hand-written BXTEND definitions. Therefore, this approach is very pragmatic and increase expressiveness with limited formal guarantees (Bank, Buchmann and Westfechtel, 2021).

BXTEND(DSL) combines declarative bidirectional and imperative unidirectional Definitions

Summarizing bidirectional transformation approaches in Figure 3.1, there is an active ongoing research of multiple BX approaches fulfilling consistency of models in both directions (Requirement R 1 (Model Consistency) ☞ 154). In order to derive transformation for two direction from one specification, TGGs, QVT-R and BXTEND(DSL) provide completely automated transformations, while EVL+STRACE and JTL let humans select the final transformation result. Only some approaches, i. e. TGGs and QVT-R, aim to explicitly support also consistency of more than two models. Surprising is the use of proof-theory-based techniques by some BX approaches, e. g. by JTL and some QVT-R implementations, while other approaches include imperative definitions, e. g. EVL+STRACE and BXTEND(DSL). Incrementality is discussed in following paragraphs.

Summary of BX Approaches

Criterion	TGG	QVT-R	EVL+Strace	JTL	BXtend(DSL)
Selection	automated	automated	interactive	interactive	automated
Proof-Theory	–	(✓)	–	✓	–
Imperative Details	–	–	✓	–	✓
Incrementality	✓	–	✓	–	✓
Multi-Models	✓	(✓)	–	–	–

Table 3.1: Comparing BX Approaches

Inter-modeling (Guerra, de Lara et al., 2010) describes relations between two metamodels, specified by bidirectional, declarative patterns (de Lara and Guerra, 2012). From these patterns, specifications for other inter-modeling activities can be generated, for e. g. forward and backward model transformations, creating and updating traceability links and model matching. While the latter could be used to check consistency of redundant concepts, real model synchronization is marked as future work (Guerra, de Lara and Orejas, 2013, p. 172). Therefore, this approach does not fulfill Requirement R 1 (Model Consistency) ☞154.

Inter-Modeling

SyncATL (Xiong, Liu et al., 2007) tries to propagate changes in the target model back into the source model having a unidirectional model transformation, applied to ATL (Jouault, Allilaire et al., 2008), without an inverse model transformation: During the execution of the model transformation, among others, mappings between source and target elements are identified and remembered (which can be seen as some kind of links), which are used to propagate changes in the target model back into the source model. This approach has some limitations in detail, which is not surprising, since the view-update problem is not decidable in general (Dayal and Bernstein, 1982), e. g. elements which are manually created in the target model cannot be propagated back into the source model, if these elements are also created by the model transformation. Therefore, this approach does not completely fulfill Requirement R 1 (Model Consistency) ☞154. The performance of this approach is challenging, since it requires the complete execution of the model transformation (with complexity linear to model size) and difference calculations of two versions of the target model (with complexity squared to model size). The implementation of this approach is limited to ATL, since the ATL model transformation engine is adapted on byte-code level. "*Only a small portion of the* ATL *standard library is supported*" (`https://xiongyingfei.github.io/modelSynchronization.html`, 2021-07-14).

derive Changes for Back-Propagation from unidirectional Model Transformation

Since model transformations enforce exactly one solution for fixing an inconsistency (i. e. the generated target elements), but there are situations with several possible and reasonable solutions, Demuth, Lopez-Herrejon and Egyed (2015) propose not to generate a single fix in the target model, but to generate constraints for the target model instead: These constraints specify the solution space for possible fixes in the target model depending on the current source model, which enables different solutions for fixing inconsistencies, as long as the constraints are fulfilled. The selection of the desired solution is done by the user in this approach, i. e. the solution space is calculated automatically, but the solution is selected manually by the user.

constrain possible Fixes for occurred Inconsistencies

Incremental model transformations can improve the performance compared to full-batch model transformations by transforming only elements or model transformation rules which are impacted by model changes. Additionally and more important, incrementality can ensure, that manual changes and information which are not in the source model are kept in the other model by updating transformed elements which already exist and by not replacing them, which is important in particular for the symmetric case. Some approaches using incremental model transformations for consistency are presented in the following paragraphs.

incremental Model Transformations

Giese and Wagner (2009) enable incrementality for TGGs by adapting the execution algorithm for TGG model transformation definitions. The correspondence graph which was created during the first complete transformation of the TGG is reused during the incremental transformation and provides the mapped elements (one in the source model and one in the target model), the executed patterns for these mapped elements and the execution order of patterns in form of a directed acyclic graph (DAG): If a previously executed pattern does not match anymore due to changes like deleted elements, the pattern execution (including all its executed sub-patterns) is reverted by deleting the corresponding element and its related elements. If a previously executed pattern still matches, but conditions for attribute values are invalid due to changes like changed values, the current values are propagated to the other model. If there are elements which are not matched by previously executed

incremental TGGs

patterns due to changes like newly created elements, the usual TGG transformations are applied to them. This algorithm saves matches and transformations for unchanged elements, resulting in improved performance (Giese and Wagner, 2009). It is fully automatic, in contrast to an ancestor algorithm which lets users decide in case of conflicting rule applications (Becker, Herold et al., 2007). The presented algorithm is improved by Greenyer, Pook and Rieke (2011) in order to prevent unnecessary deletions of elements in the target together with their manual changes: Instead of immediately deleting elements of invalid patterns, these elements are only marked as deleted and are reused, if they are (re-)created by other TGG patterns. Lauder, Anjorin et al. (2012) present an algorithm for incremental execution of TGGs with improved performance and the formal guarantees of Hermann, Ehrig et al. (2011) (see above), mainly done by also un-transforming elements related to created (and not only deleted) elements and establishing fixed execution orders. As an alternative, Leblebici, Anjorin et al. (2017) use incremental pattern matching for incrementality. For TGG engines, there is a trade-off between performance and formal guarantees, at least regarding backtracking capabilities for incrementality (Leblebici, Anjorin et al., 2014). Blouin, Eustache and Diguet (2014) show an example, how a synthetic approach for model consistency preservation is realized with TGGs for incremental model synchronization. Abilov, Mahmoud et al. (2015) apply eMoflon for incremental bidirectional model synchronization in the domain of software development artifacts.

Also BX has to deal with the selection of one of multiple possible fixes for the back-transformation (Zan, Pacheco and Hu, 2014) in order to resolve ambiguities (Eramo, Marinelli and Pierantonio, 2014). This selection should be deterministic (Stevens, 2010), so current research investigates possible heuristics to choose fixes. Cheney, Gibbons et al. (2017) investigated *least change and least surprise* for bidirectional model transformations: While working in general, they found some issues with least change including, that metrics for least change on amount of change operations do not always fit to the expectations of users working with tool environments or DSLs, including the problem that changes can be realized also with adding and deleting but with different numbers of change operations, that least changes are often useful, but not always and that "*metric-least consistency restoration is NP-hard*" (Cheney, Gibbons et al., 2017). Least surprise, i. e. small changes in one model are reflected by small (but not minimal) changes in the other model, requires further investigations in general. Echo realizes least change for QVT-R (Macedo and Cunha, 2016). The TGG engines MoTE and TGG Interpreter apply some strategies to realize least change in practice, but cannot guarantee least change in general (Leblebici, Anjorin et al., 2014).

least Change and least Surprise

Since explanations and other heuristics prevent project-specific decisions, Zan, Pacheco and Hu (2014) allow to imperatively customize the desired back-propagation of changes for bidirectional model transformations, demonstrated in contrast to QVT-R. Since these approaches support methodologists, these approaches are discussed in Section 6.3.1 ☞ 199.

customize Back-Propagation

Lenses (Foster, Greenwald et al., 2007) are a theoretical concept[6] to formalize the semantics of pairs of model transformations and their properties for model synchronization: Since lenses describe the relation of models in total to each other and not of single consistency relations, the cases of information overlaps between these models are relevant, i. e. bijective, asymmetric and symmetric (see Figure 3.5 ☞ 104). *Asymmetric lenses* for the asymmetric case consist of two functions *get* and *put* (or *putback*) which allow to synchronize a source model $s \in S$ with a view $v \in V$ of it (Abou-Saleh, Cheney et al., 2018):

Lenses formalize Model Synchronization

asymmetric Lenses

$$get : S \longrightarrow V \quad (3.1)$$

$$put : S \times V \longrightarrow S \quad (3.2)$$

[6]Note, that the formalisms given in the cited publications for lenses are strongly summarized here with strongly simplified notations.

$get(s^0) = v^0$ derives the current view v^0 from the current source s^0, while $put(s^0, v^1) = s^1$ propagates user changes $^{\text{User}}\Delta$ in the updated v^1 back into the source s^1 to update it accordingly. S as additional input for *put* is required to prevent information loss, since v does not contain all information of s. Except for this additional parameter, these formalizations fit to the generic visualization of model synchronization-based approaches in Figure 3.4c ☞103. Based on these formalizations, some laws for lenses are proposed in order to specify the desired behavior for model synchronization:

- *Well-behaved* lenses do not change the source (respectively view), if the view (respectively source) is unchanged: $put(s, get(s)) = s$ respectively $get(put(s, v)) = v$
- *Very-well behaved* lenses (this property is also known as “history ignorance”) ignore effects of views in older states, i. e. only the newest state of the view influences the source: $put(put(s, v^0), v^1) = put(s, v^1)$

This formalization is state-based up to now, since only changed models are involved, but no information about the concrete changes. Since there are multiple possible model differences between two versions of a model, state-based approaches have more ambiguities to deal with than delta-based approaches. A typical example in practice are simple changes of attribute values in v, which could also be represented as deleting an object and recreating it with a different value. But the result of *put* is different regarding values of this object which are only in the source s. Therefore, asymmetric *delta-lenses* are introduced (Diskin, Xiong and Czarnecki, 2011, 2010):

asymmetric Delta-Lenses

$$dget : \Delta_S \longrightarrow \Delta_V \tag{3.3}$$
$$dput : S \times \Delta_V \longrightarrow \Delta_S \tag{3.4}$$

dget and *dput* are similar to *get* and *put*, but work with model differences instead of models. Concretizing the model differences helps to uniquely determine, which objects are created, changed or deleted, as (vertical) *alignment* of objects in two different model versions (Anjorin, Buchmann et al., 2020). Since the preparation of model differences is separated from the direct model synchronization, they can be controlled better by using different strategies. Usual model difference calculation can be used to calculate Δ_V between v^0 and v^1, which allows to map “simple” lenses to delta-lenses. The laws for lenses are accordingly adapted for delta-lenses (Diskin, Xiong and Czarnecki, 2011). Therefore, delta-lenses with *dget* and *dput* can be seen as the transition from model synchronization-based approaches to change translation-based approaches, investigated in following paragraphs. Delta-lenses for the asymmetric case can be complemented with explicit links (as in incremental model transformations) as additional inputs into *dget* and *dput*, which provide updated explicit links as additional outputs (Diskin, Xiong and Czarnecki, 2011) and is also used for the symmetric case now.

Symmetric lenses for the symmetric case between source model $s \in S$ and target model $t \in T$ are investigated in similar way including support for deltas by Diskin, Xiong et al. (2011) with the two operations *fPpg* (*f*orward *P*ropa*g*ation) and *bPpg* (*b*ackward *P*ropa*g*ation), which take the current user diff and the old correspondences $r \in R$ between s and t as input (see Section 2.3 ☞71) and produce the other model differences and the updated correspondences as output:

symmetric Delta-Lenses

$$fPpg : \Delta_S \times R \longrightarrow \Delta_T \times R \tag{3.5}$$
$$bPpg : \Delta_T \times R \longrightarrow \Delta_S \times R \tag{3.6}$$

Since symmetric lenses target the symmetric case between two models, these signatures are symmetric, too, and the terms forward and backward (as well as source and target) could be switched. Adapted laws for symmetric delta-lenses target only the information

overlap between s and t and therefore are weaker versions of the strong laws covering all information.

Since the lenses which are described up to now are binary lenses working with only two models, Diskin, König and Lawford (2018) investigate symmetric *multiary lenses* with deltas in order to keep $n \geq 2$ models $m_{1,\ldots,n} \in M_{1,\ldots,n}$ consistent to each other:

symmetric multiary Delta-Lenses

$$ppg_i : \Delta_{M_i} \times R \longrightarrow \Delta_{M_1} \times \Delta_{M_2} \times \ldots \times \Delta_{M_n} \times R \quad (\forall i \in 1, 2, \ldots, n) \tag{3.7}$$

As for binary symmetric lenses, the *p*ro*pag*ation operators $ppg_{1,\ldots,n}$ use and update correspondences $r \in R$ for horizontal alignment between n models. Again, adapted laws for symmetric multiary delta-lenses target only the information overlap between the models $m_{1,\ldots,n}$. Note, that if Δ_{M_i} in the output is not empty, it amends the input Δ_{M_i} and is concatenated with it in order to specify the model differences from M_i^0 to M_i^1.

Summary of Lenses

Summarizing lenses, they are only a formalization of BX. Lenses are distinguished regarding the number of involved models (binary vs multiary), the information overlap of involved models (asymmetric vs symmetric) and the encoding of model changes (updated model vs model deltas). Since lenses provide *formal hints guiding implementations*, they lack a direct, concrete implementation: Lenses can be implemented with TGGs for the binary symmetric case with deltas (Hermann, Ehrig et al., 2015) and can be realized as composable combinators respectively operators forming a DSL for BX (Diskin, König and Lawford, 2018; Foster, Greenwald et al., 2007). Some more relations to other approaches are summarized in Section 3.7 ☞146. An alternative formalization for BX is relational (Abou-Saleh, Cheney et al., 2018, p. 6f) basing on the formalisms in Section 2.3 ☞71.

Round-trip Engineering

Round-trip engineering in the context of models is defined by Hettel, Lawley and Raymond (2008) as a property of model transformations for two directions and is required, since model "*transformations in general are neither total nor injective*" (Hettel, Lawley and Raymond, 2008) in the symmetric case: As visualized in Figure 3.7, after executing a

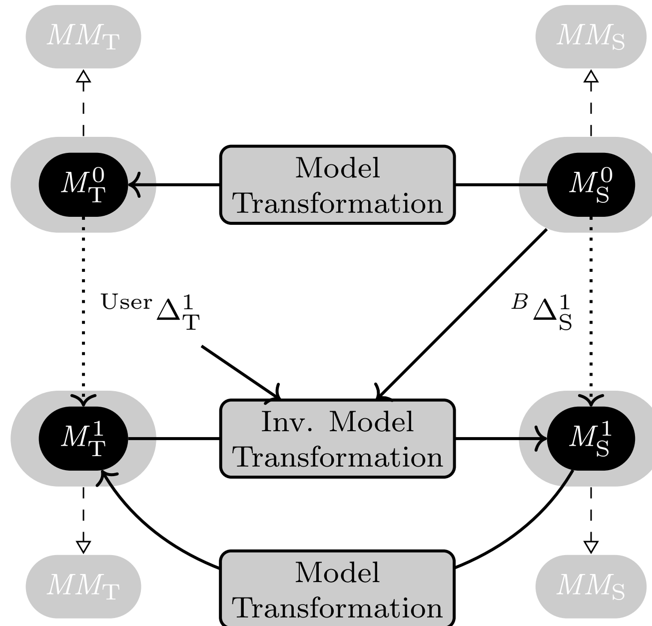


Figure 3.7: Round-Trip Engineering of Model Transformations

forward model transformation ("Model Transformation"), an inverse model transformation ("Inv. Model Transformation") executed on the updated M_T^1 fulfills round-trip properties, if the forward model transformation provides the same M_T^1 from the output M_S^1 of the inverse model transformation. In other words, forward and inverse model transformations

do not contradict each other regarding the information targeted by them. With this design, properties for round-trip engineering are very similar to symmetric lenses with their laws. "*In contrast to reverse engineering, round-trip engineering does not aim at recovering lost or otherwise unavailable source models, but is rather concerned with propagating changes from target to the source model*" (Hettel, Lawley and Raymond, 2008). Approaches fulfilling round-trip engineering often use incremental bidirectional model transformations (see above), but there is also another proof-theory-based approach using "*abductive reasoning, the inference to the best explanation*" (Hettel, Lawley and Raymond, 2009, p. 113).

Round-trip with Source Code

Having models and code, Stahl, Völter et al. (2005, p. 74) define round-trip engineering as having the transition from model to code (forward engineering) and the transition from code to model (reverse engineering). In that sense, model-driven software development is only forward engineering without reverse engineering and therefore without round-trip engineering. Angyal, Lengyel and Charaf (2008) realize model-code-round-trip by model difference calculation and merging on the AST representing the code.

Summary of Model Synchronization-based Approaches

Summarizing model synchronization-based approaches, they use model transformations to ensure that parts of two models fit together in terms of consistency (Requirement R 1 (Model Consistency) ☞ 154). BX is well-suited for this task, since one specification allows to execute model transformations in both directions. Incremental model transformations improve performance and ensure, that changes are propagated from one model into the other, while other, manually edited parts of the models remain unchanged, which is important in symmetric settings occurring in synthetic approaches. The presented approaches strongly vary in the strategies, how to determine the final results of model transformations (in particular for the inverse direction), including non-deterministic automated selection, manual selections by users, heuristics like least change, additional imperative specifications and only defining the possible solution space. Since model transformations for both directions must ensure consistency for both involved models, formalizations in form of lenses and round-tripping specify the desired behavior.

in-place Graph Transformation Rules for 1st marking and 2nd fixing Inconsistencies

While the presented approaches using model transformations work out-place and exogeneous, using model transformations also in *in-place and endogeneous* way is possible for ensuring consistency: Mens, Van Der Straeten and D'Hondt (2006) use graph transformation rules on AGGs working like pattern matching for both searching for inconsistencies and fixing them. Possible cycles and conflicts of depending rules are covered by critical pair analysis at development time. At runtime, all rules for inconsistency checking are applied and inconsistent elements are annotated with special nodes. Afterwards, these nodes are used by the rules for inconsistency fixing for matching and fixing the inconsistent pattern (Mens and Van Der Straeten, 2007). This approach requires additional support for multi-models. To improve performance, Blanc, Mougenot et al. (2009) present the idea of representing the model as sequence of model differences, specifying the rules not on (meta)models but on model differences and evaluate only relevant rules on the currently added model differences. This extension can be seen as incremental execution, very similar to change translation-based approaches.

Change Translation-based Approaches

Change translation-based approaches propagate only changes made in one view to all other related views (and do not transform whole views). The challenge is to cover the syntactic and semantic heterogeneity of views in form of different metamodels and meanings on level of model changes, since changes of the first view must be converted, i. e. translated in changes for the next view. These converted changes allow to give up incremental model transformations as used in model synchronization-based approaches. Change translation-based approaches allow to give single views more control about the changes which they communicate to other related views, e. g. Lee (2010) proposes an actor-based design to communicate changes between models.

El Hamlaoui, Bennani et al. (2019) establish explicit correspondences between the heterogeneous models as first step. In the second step, changes of elements within one model

propagate Creations and Deletions along 1-to-1 Correspondences

are propagated along their correspondences into the other models. This approach supports automated change propagation for created and deleted elements and semi-automated solutions for changed elements, which must be decided by experts. Experts might collaborate with each other in order to identify (more) correspondences, to decide consistency issues and to break cycles of the consistency management process (Bennani, El Hamlaoui et al., 2018; El Hamlaoui, Bennani et al., 2019). Up to now, this approach does not support consistency goals which establish no direct one-to-one links between the involved elements. Therefore, this approach fulfills Requirement R 1 (Model Consistency)☞154 only partially.

Establish consistency-specific explicit Links

Feldmann, Wimmer et al. (2016) establish links stored in link models between pairs of involved models. These links are typed to indicate different relationships like refinement or equivalence. Together with project-specific constraints (formulated as patterns, see de Lara and Guerra (2012) above) and consistency rules, the links are used to check consistency and to fix inconsistencies. Details of the inconsistency handling seem to be under development.

Beanbag: translate Updates through OCL-like Expressions

Xiong, Hu et al. (2009) develop a restricted OCL-like language Beanbag that combines checking and fixing inconsistencies with the same declarative expression. These expressions are evaluated on the current model in order to detect inconsistencies, like in rule-based approaches. Additionally, these expressions can be evaluated in case of changes by users with previous values and updated values resulting in further updates, which makes Beanbag a change translation-based approach. For that, all language concepts are complemented with semantics, how to deal with occurred changes. Mainly, this includes "*to propagate updates through equality constraints [(=, let)], control the propagation order by logic operators [(and, or, not)], derive structural updating through logic quantifiers [(forall, exists)], restrict fixing behavior through special constructs [(protect, test)], and introduce recursion for describing more involved fixing strategies*" (Xiong, Hu et al., 2009, p. 320). Evaluated in the context of UML, Beanbag "*can support many, though not all, useful fixing scenarios in practice*" (Xiong, Hu et al., 2009, p. 324), due to the small number of language concepts. Therefore, this approach fulfills Requirement R 1 (Model Consistency)☞154 only partially. Beanbag is applied for synchronizing views for web-development (Ruiz-González, Koch et al., 2009).

Summary of Change Translation-based Approaches

Summarizing change translation-based approaches, they directly react on occurred changes in one model and generate changes for related models to keep them consistent to the changed model. Some approaches exploit explicit links between models to directly propagate changes along them, similar to incremental model transformations also using explicit links in order to improve performance and to keep unchanged model elements. All the presented change translation-based approaches have some limitations regarding Requirement R 1 (Model Consistency)☞154. Another more generic change translation-based approach as part of Vitruvius is presented in Section 3.5.2☞126.

Synthetic Settings by explicitly linking Models with each other

Summarizing the technique of explicitly linking models, all approaches which use explicit links between pairs of models can be classified as synthetic approaches. In order to store explicit links outside of the linked models, an additional model is introduced, which makes the setting even "more synthetic". This counts for all kinds of explicit links (Figure 3.2☞100): As an example for traceability, Baumgart and Ellen (2014) use traceability links to explicitly relate models of different tools to each other in order to enable validation and verification across tool boundaries, but without fixing inconsistencies (Requirement R 1 (Model Consistency)☞154) or other findings. As an example for model weaving, Mehner, Monga and Taentzer (2009) check the consistency between use case diagrams and concretizing activity diagrams according to pre- and post-conditions by weaving them together controlled by defined pointcuts following aspect-orientation. Fixing inconsistencies is not targeted (Requirement R 1 (Model Consistency)☞154). Section 3.5.4☞131 reviews approaches using aspect-orientation to weave different models into a single model. But only linking models without ensuring consistency is not sufficient here. The other way around, linking models can be used as supporting technique for approaches ensuring inter-model consistency.

There are also approaches, which are not related to the synchronization of models, but

Demarcation

are synthetic: Zhang and Moller-Pedersen (2013) define “tool integration models” which are models used for integrating tools: These integration models define APIs, used models and more specifics of tools, which are used to orchestrate integrations of the described tools. This is done in synthetic way, but not with the focus on consistency of the involved models the tools are working with. In particular, propagation of changes in both directions is not covered (Requirement R 1 (Model Consistency)[☞ 154]).

Summary of synthetic Approaches

Summarizing synthetic approaches, they realize the consistency between different models (Requirement R 1 (Model Consistency)[☞ 154]) by directly propagating changes between related models without intermediate models. Synthetic approaches use various techniques for change propagation. Since nearly all of the presented approaches support Requirement R 1 (Model Consistency)[☞ 154], but not explicitly Requirement R 3 (Define new View(point)s)[☞ 156], the following Section 3.3.2 reviews approaches for defining new views in synthetic settings.

3.3.2 Synthetic View Definition

This section presents some selected approaches to define new view(point)s in the synthetic setting, since there are approaches which allow to define new views, that do not support consistency of existing data sources in synthetic way: Such approaches could be integrated with synthetic consistency approaches in order to define new views according to Requirement R 3 (Define new View(point)s)[☞ 156]. The easiest way is to create a new view(point) manually and treat it as existing data source using the mechanisms for consistency preservation in Section 3.3.1[☞ 108], but there are more advanced approaches specially designed to define new views. Bruneliere, Burger et al. (2019) provide a survey for such approaches.

new View(Point) on top of multiple View(Point)s

Important in the synthetic setting is, that new view(point)s are defined on top of *multiple* existing view(point)s, if the new view should contain concepts of different existing data sources, since there is no SUM in synthetic approaches. As an example, the new view in Part 6[☞ 40] of the ongoing example for the traceability between requirements and source code needs the requirements sentences from the requirements data source and the implementing methods from the Java data source. This motivates to concretize Requirement R 3 (Define new View(point)s)[☞ 156] with the following Requirement R 3.1[☞ 157]:

> **Requirement R 3.1: New Views reuse whole System Description**
>
> New views must be able to reuse all information which represent the whole system under development.

Only approaches fulfilling this requirement are presented in this section. If the new view contains only concepts of one existing view, Section 3.5.5[☞ 134] presents approaches to define new views on top of exactly one existing view. Keeping a new view consistent to the existing views is slightly easier than keeping existing views consistent to each other, since the new view contains only information which is already present in the existing views. Therefore, asymmetric approaches are sufficient in this section here, while ensuring consistency between existing views in Section 3.3.1[☞ 108] requires symmetric approaches.

asymmetric Approaches are sufficient

EMF Views are read-only

With EMF Views (formally known as VirtualEMF (Clasen, Jouault and Cabot, 2011)), there is an approach to define non-materialized view(point)s from multiple source (meta)models (Bruneliere, Perez et al., 2015). Bruneliere, de Kerchove et al. (2018) improved scalability for performance and add support for EMF models stored in different data bases and queries (Bruneliere, de Kerchove et al., 2020). Since these new views are read-only[7], EMF Views does not fulfill Requirement R 3 (Define new View(point)s)[☞ 156] com-

[7] In Bruneliere, Burger et al. (2019) with overlapping authors, there is a hint for some limited support for back-propagation. Following Burger and Schneider (2016), only changes for primitive attributes can be propagated. Bruneliere, de Kerchove et al. (2020) mark editability as future work.

pletely and it again motivates to concretize Requirement R 3 (Define new View(point)s) ☞156 with the following Requirement R 3.3 ☞157:

> **Requirement R 3.3: Editable new Views**
>
> New views must be editable by users.

Since EMF Views does not target the consistency between existing data sources, but only to derive new view(point)s from multiple data sources, it is helpful in synthetic approaches for realizing (read-only) views. In general for read-only views, any approaches for model analyses including model querying can be used, including incremental model queries (Hinkel, Heinrich and Reussner, 2019).

EMF-IncQuery: incremental Back-Propagation with Proof-Theory

Debreceni, Horváth et al. (2014) realize new views by incremental queries on multiple (meta)models using EMF-IncQuery as a unidirectional model transformation as part of the VIATRA framework (Varró, Bergmann et al., 2016). Correspondences between the view and the source models are used to incrementally back-propagate changes in the view by using the SAT solver of Alloy (Semeráth, Debreceni et al., 2016), which generates possible corresponding source models for selection by users, representing a proof-theory-based technique. This approach fulfills Requirement R 3 (Define new View(point)s) ☞156 with limited automation of the back-propagation of changes.

Summary of synthetic View Definition

Summarizing approaches for synthetic view definition, they are required to present information spread over multiple models in a different way within one new view. The other challenge is to propagate changes in the new view back into the source models, which occur also for BX (see Section 3.3.1 ☞108) and for the view-update problem (see Section 3.6.3 ☞139). Both challenges occur for the definition of new views in synthetic settings and strongly reduce the number of usable approaches. Another approach to define editable viewpoints and views on multiple base (meta)models is ModelJoin in the context of the Vitruvius approach, presented in Section 3.5.2 ☞126.

3.4 Single Underlying Model (SUM)

SUM as intermediate Structure for projectional Approaches

After analyzing synthetic approaches in Section 3.3 ☞108, projectional approaches are analyzed in Section 3.5 ☞121. In contrast to synthetic approaches, projectional approaches use an intermediate structure to store at least overlaps of views in order to synchronize them indirectly via the intermediate structure. Therefore, this section introduces the concept of Single Underlying Models (SUMs) as intermediate structure as preparation for the analyses of projectional approaches in Section 3.5 ☞121.

SUM conforms to SUMM

SUM describes System completely

To realize the intermediate structure required for projectional approaches, Atkinson, Stoll and Bostan (2009) present the idea of a Single Underlying Model (SUM), which conforms to a corresponding Single Underlying MetaModel (SUMM). The SUM completely describes the system under development of the particular project in total. In particular, it contains the information of all view, as well as additional information like explicit links between non-overlapping information from different views. Compared with Figure 3.2 ☞100, the SUM stores all “Non-Overlaps”, “Overlaps” and “Inter-Correspondences”.

Ensure Consistency between View and SUM

Views are projected from the SUM by users on-demand. Users work only on their views and changes in the views are propagated back into the SUM. In this way, consistency is not ensured in pair-wise manner between all views directly as in synthetic approaches, but consistency is ensured between the particular view and the SUM. Since all views are projected from the SUM, e. g. by model transformations, changes in the SUM are propagated also into the other views (Requirement R 1 (Model Consistency) ☞154).

Advantages of a SUM

Since the SUM contains all information of the system in a uniform way, the SUM serves as single point-of-truth in projectional settings. Another advantage of a SUM is, that the

management of dependencies between information spread over multiple views (page 33) could be simplified in a SUM: Redundancies of views could be reduced by representing the redundant information only once in the SUM, links between views can explicitly represented in the SUM and constraints can be checked with usual techniques directly on the SUM.

The *single model principle* of Paige and Ostroff (2002) can be seen as an early ancestor of the SUM idea, since it advocates the use of multiple views which are derived from a single model to ensure their consistency. While in particular views with different levels of abstraction are considered for SUMs, the definition of the single model principle restricts its scope to software development with levels for modules and systems. The single model principle is used by (Haesen and Snoeck, 2005) for the single formal specification and different views for conceptual modeling. The *One-Thing-Approach* (Margaria and Steffen, 2009) advocates the use of a single and consistent representation for all information of a system during its whole life cycle and different views on it.

Related Work

In related work, lots of different terms are used to name structures covering important overall information, which act similar to SU(M)Ms, even if not all properties of the SUM idea are fulfilled, including "repository" (Guerra, Diaz and de Lara, 2005) according to the original term used in IEEE (2011): In the field of embedded software-intensive systems, Broy, Feilkas et al. (2010) call the SUM as "product model" covering all artifacts and their relations of the whole development. Other terms are "unique model" (de Lara, Guerra and Vangheluwe, 2006), "pivot model" (Kurtev, 2008, p. 382) and "global model" containing only explicit links in a synthetic setting (El Hamlaoui, Bennani et al., 2019). A SUMM is called "common metamodel" (Baumgart, 2010; Persson, Torngren et al., 2013), while Persson, Torngren et al. (2013) provide also the alternatives "shared metamodel" and "pivot metamodel" (Kappel, Kapsammer et al., 2006, p. 14). This thesis uses the terms SUM and SUMM in the sense of Atkinson, Stoll and Bostan (2009), since this SUM idea is designed for projectional settings and is already used by realizing approaches (Meier, Werner et al., 2020).

alternative Terminology

While the SUM idea focuses on a single, central *model*, other ideas use other infrastructures as single means for synchronization: As an example, ModelBus (Hein, Ritter and Wagner, 2009) provides a central model repository and an interaction pattern for communication for the integration of tools, but without a single model, and does not focus on consistency, but provides sharing of models and versioning support for models. In contrast, the tool support and communication infrastructure are not specified.

Demarcation: ModelBus

Since the SUM describes the current system under development in its entirety, SUM and SUMM are specific for the current system and its project for development and are not generic as UML. Approaches fulfilling this SUM idea realize the SUM either *explicitly* as OSM (Section 3.5.1 ☞124) or *implicitly* as Vitruvius (Section 3.5.2 ☞126). Persson, Torngren et al. (2013) raise the question, how complete the SUM should be, e. g. whether non-overlapping information should be stored in the SUM (Figure 3.2 ☞100). Following the SUM idea as presented above, the SUM contains *all* information about the system under development including all information of all views in this thesis. Nevertheless, this discussion is taken up in Section 13.3.3.2 ☞476.

towards realizing SUMs

3.5 Projectional Approaches

The main characteristic of projectional approaches is, that they establish direct relations between models of views and an intermediate structure, not between views directly. Following the SUM idea (Section 3.5.3 ☞129), this section focuses on related projectional approaches, which use a SU(M)M as intermediate structure. The focus of this section is on *generic* approaches to support arbitrary development projects by projectional view management. In particular, projectional approaches which represent a specific solution for a restricted

generic vs specific Approaches

project setting are neglected.

Restricted SUMM Approaches

Some examples for *restricted* approaches are sketched here to get an impression for such approaches and to see the practical relevance of projectional approaches. In particular, such restricted approaches follow the main idea of the OSM approach (even if they are not explicitly stated as SUM approaches), but have fixed viewpoints (France and Rumpe, 2007), in contrast to generic approaches, which support arbitrary viewpoints:

- Cicchetti, Ciccozzi et al. (2012) present the projectional tool environment CHESS for modeling complex industrial systems with some fixed viewpoints and hard-coded support for automated consistency management: The integrated modeling language CHESSML supports viewpoints for requirements, system design, components, deployment, analysis and instances (Debiasi, Ihirwe et al., 2021).

- Shah, Kerzhner et al. (2010) use the SysML metamodel extended by using its profiles mechanism as SUMM and derive new views with model transformations. They marked the involvement of a general approach for consistency management as open question.

- Makedonski and Grabowski (2020) improve consistency of specifications in standards by keeping a unified information model as SUM up-to-date which represents all information elements of all specifications.

- UML uses different diagram kinds which can be treated as different viewpoints of the same SUMM, e. g. by de Lara, Guerra and Vangheluwe (2006) for consistency preservation using TGGs. Section 3.6.1 [☞ 136] analyzes some existing approaches for consistency between UML diagrams.

- The Siemens views approach (Hofmeister, Nord and Soni, 2000) introduces viewpoints for conceptual for functional components, modules, the execution and code artifacts to describe software architectures. Their integration into a SUMM to describe the architecture as a whole is sketched, but not made explicit. Therefore, the integration of these extended viewpoints with MoConseMI is demonstrated in Chapter 10 [☞ 373].

- Vogel-Heuser, Fay et al. (2015, p. 65) present some approaches using predefined SU(M)Ms for system design and call the problem to be solved with these approaches, but do not answer the question, how the required SU(M)Ms can be developed in a structured way.

- Rinker, Waltersdorfer et al. (2021) follow the projectional SUM idea for continuous integration in the domain of production systems engineering. Since this approach is ongoing research, it is unclear, if this approach is generic and could be applied to other application domains. While the approach seems to be bottom-up taking existing view(point)s into account, the concrete strategy to develop the required SU(M)M is left open.

Generic Data Consistency without Tool Integration

Summarizing, these restricted approaches and tools show, that the projectional management of multiple views works in practice. Disadvantage of them is, that they support only views which are realized with the techniques specified by the tools or whose viewpoints and consistency goals are fixed in the approach. In contrast, MoConseMI aims to keep the data of multiple views consistent to each other, independently from the underlying tools: The objective is to achieve data consistency (by means of data integration) without forcing tool integration. These findings motivate to concretize Requirement R 1 (Model Consistency) [☞ 154] with the following Requirement R 1.1 [☞ 154]:

Requirement R 1.1: Generic Metamodels

The approach must support arbitrary metamodels.

Therefore, generic approaches are analyzed here, which are able to support arbitrary viewpoints and consistency goals. If the views and the SUM are available as explicit models, they can be kept consistent in projectional way by reusing synthetic techniques, as investigated in Section 3.3 ☞108: Since the SUM contains all information of all views by design, techniques for the asymmetric case are sufficient to synchronize each view with the SUM. This counts in particular for model synchronization-based approaches like BX, but also proof-theory-based, rule-based and change translation-based approaches can be used for projectional settings, since again models are kept consistent to each other, only with differences in topologies (stars vs mashed graphs) and information overlap (asymmetric vs symmetric). This includes also formalizations with (asymmetric) lenses and round-tripping. Since these techniques are already discussed, the following sections analyzes approaches which explicitly follow projectional ideas. In particular, the approaches which are explicitly designed to support projectional development following the SUM idea are discussed, according to the following papers:

Reuse synthetic Techniques for projectional Settings

Related MoConseMI Publication

Johannes Meier, Christopher Werner, Heiko Klare, Christian Tunjic, Uwe Aßmann, Colin Atkinson, Erik Burger, Ralf Reussner and Andreas Winter (2020): *Classifying Approaches for Constructing Single Underlying Models.* In: Slimane Hammoudi, Luís Pires Ferreira and Bran Selic (Eds.): Model-Driven Engineering and Software Development. MODELSWARD 2019. Communications in Computer and Information Science (CCIS), Springer, Cham, pp. 350–375.

This publication is cited as Meier, Werner et al. (2020) in this thesis.

It is an extension of this paper:

Related MoConseMI Publication

Johannes Meier, Heiko Klare, Christian Tunjic, Colin Atkinson, Erik Burger, Ralf Reussner and Andreas Winter (2019): *Single Underlying Models for Projectional, Multi-View Environments.* In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, pp. 119–130.

This publication is cited as Meier, Klare et al. (2019) in this thesis.

Additionally, these papers provide some conceptual design choices to classify projectional approaches. While these design choices target the conceptual solution space, the design choices of Figure 3.2 ☞100 focus on the technical realization of approaches after deciding their conceptual design choices. The design criteria of Meier, Klare et al. (2019) describe two conceptual design choices for projectional approaches, which are depicted in Figure 3.8 ☞124: The *construction process* (criterion C1 in Meier, Werner et al. (2020)) refers to the process of creating the SU(M)M, which can be *top-down* by designing the SU(M)M on the greenfield or *bottom-up* by combining reused (meta)models into the SU(M)M. Bork and Sinz (2013, p. 29) use the same distinction, but without explicitly using this terminology. The *resolved dependencies* (Pureness as criterion C2 in Meier, Werner et al. (2020)) refer to the dependencies like redundancies between views, which could be all unresolved in the SUM (*none*), completely resolved with a pure SUM (*all*) or resolved in some cases, but not all depending on the particular project (*adjustable*).

additional conceptual Design Choices

Since Meier, Klare et al. (2019) focus only on projectional approaches, Figure 3.8 ☞124

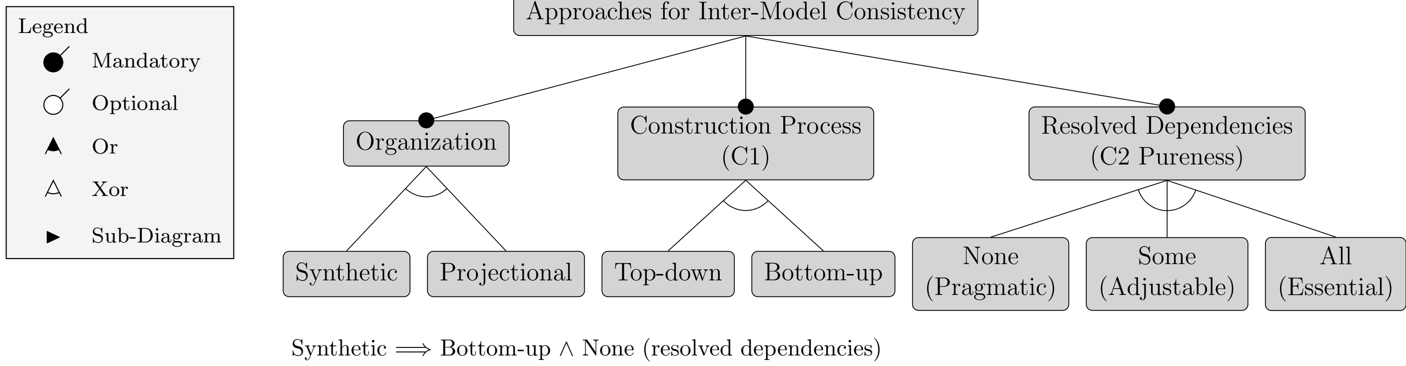


Figure 3.8: Design Choices for conceptual Realization (derived from Meier, Werner et al. (2020))

Conceptual Design Choices for synthetic Approaches

makes this explicit with the feature “organization”. In order to make this conceptual design choices generic, the feature model supports synthetic approaches, too: Since there is no SU(M)M in synthetic settings, synthetic approaches are always bottom-up. Since there is no SU(M)M in synthetic settings, the existing views remain with all their dependencies and none of them are resolved. Since Figure 3.8 supports synthetic *and* projectional approaches now, it can be used to design new approaches for ensuring inter-model consistency with synthetic vs projectional as additional design choice, as it is done in Section 5.1☞163.

Outline for related projectional Approaches

Since they state to be generic projectional SUM approaches, three approaches of Meier, Werner et al. (2020) are analyzed in more detail in the following three sections: OSM in Section 3.5.1, Vitruvius in Section 3.5.2☞126 and RSUM in Section 3.5.3☞129. The design of the fourth approach MoConseMI of this thesis is presented in Part III☞163. While these approaches target all challenges of projectional SUM approaches, other generic approaches fulfilling only some of these challenges are investigated, too: Section 3.5.4☞131 investigates, how a SU(M)M can be created, if only its view(point)s exist. As follow-up step, Section 3.5.5☞134 reviews, how new view(point)s can be defined on top of an existing SU(M)M.

3.5.1 OSM

The “Orthographic Software Modeling” (OSM) approach uses *modeling* techniques to support *software* development (initially), but also other domains like enterprise architecture modeling and service modeling (Atkinson, Stoll and Tunjic, 2011), by multiple viewpoints, which are arranged in *orthogonal* way. Two main concepts of OSM are the generation of views on-demand to manage consistency and the alignment of viewpoints along orthogonal dimensions Atkinson and Stoll (2008b), investigated in the following paragraphs.

explicit SU(M)M

Ensuring Consistency in OSM

OSM uses one explicit model as SUM, which conforms to one explicit metamodel as SUMM. The SUM contains all information about the system under development in uniform way without any dependencies (pure in terms of Meier, Werner et al. (2020)). Since the SUM contains no dependencies, no inconsistencies can occur and no additional mechanisms to ensure consistency inside the SUM are required. This corresponds to Thomas and Nejmeh (1992), who have “nonredundancy” as important property: “*Redundant information [...] is undesirable because it is difficult to maintain consistency*” (Thomas and Nejmeh, 1992, p. 32). Instead, consistency must be ensured between the SUM and the views: Views are generated from the SUM on request be users and are initially consistent to the SUM therefore. After the user changed the view, the changes are propagated back into the SUM to keep it consistent. Other views are projected from the (updated) SUM again and are directly consistent and so on, as sketched in Figure 3.9☞125. Users are not allowed to change the SUM directly.

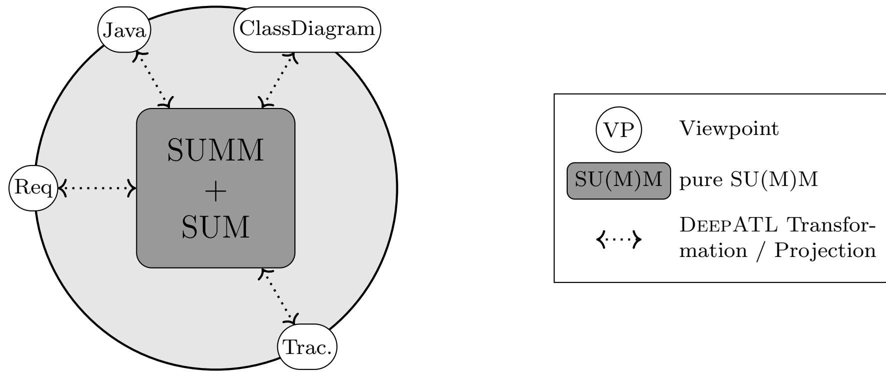


Figure 3.9: SUM approach OSM (taken and slightly adapted from Meier, Werner et al. (2020))

The change propagation between SUM and views is model synchronization-based: A view is created on-demand from the SUM by executing a unidirectional full-batch model transformation. To propagate changes in the view back into the SUM, an inverse unidirectional model transformation was used initially (Atkinson, Gerbig and Tunjic, 2013b). As improvement, Tunjic and Atkinson (2015) require only one unidirectional model transformation from the SUM to the view (*get* in terms of asymmetric lenses (Diskin, Xiong and Czarnecki, 2011), see above in Section 3.3.1 ☞ 108) which is restricted to map each element in the SUM to zero or one elements in the view and which automatically creates trace links between transformed elements. In that cases, changes for created and deleted elements can be propagated along the trace links to realize *dget* and *dput* functionalities including updated trace links (asymmetric delta-lenses with explicit links). The update facilities for attribute values are not discussed. Atkinson and Tunjic (2017) allow projections of one or more elements in the SUM to one or more elements in the view, probably leading to hyper-links, but do not discuss the view-update problem for these cases. The same counts for aggregating information from the SUM, which are probably read-only, meaning, that their changes in the view are not propagated to the SUM and are lost. If the new view is only an exact subset of the elements in the SUM, such a filtering for this special case can be specified by (deep) OCL expressions instead of complex model transformations (Lange, Atkinson and Tunjic, 2020). Summarizing, OSM fulfills Requirement R 1 (Model Consistency) ☞ 154 and Requirement R 3 (Define new View(point)s) ☞ 156 with the same mechanisms in general, but with some lack in detail.

Restricted unidirectional Model Transformation from SUM to View with Traces

Propagate Changes along Traces back into the SUM

OSM organizes viewpoints along orthogonal dimensions (Atkinson and Tunjic, 2014a), which helps users to choose the best view for their current needs and methodologists to keep an overview of already developed and still missing viewpoints. Main idea is, that n orthogonal dimensions form a n-dimensional hypercube, while each dimension has some values as possible choices for this dimension. Each cell in the hypercube is defined by one value for each of the n dimensions and can be realized by one viewpoint. Examples for such dimensions (with possible values in round brackets) include high-level components (dynamically depending on the particular system), level of abstraction (services, classes, code), development stage (specification, implementation, validation), variant (selected features in a product line) and version (released versions).

Orthogonal Dimensions for Viewpoints

Clarification: Orthogonal dimensions vs. Consistency

Note, that the hypercube does *not* represent all information of the system, but organizes all ways how stakeholders can look on this information. In particular, orthogonal dimensions target the alignment of viewpoints and do *not* separate the information into independent parts, e. g. the same source code can be visualized by

Consistency is *not* ensured by orthogonal Dimensions

the different values of the dimension level of abstraction (see above). Therefore, there is still the need for keeping information in different views consistent. *Consistency is not ensured by orthogonal dimensions*, nevertheless, the overlap of views in terms of shared information should be minimal (Atkinson and Tunjic, 2014a, p. 49).

Applications of OSM

Application examples with orthogonally aligned viewpoints include among others modeling of component-based systems according to the KobrA method (Atkinson, Bostan et al., 2008) and of components and their orchestration for workflow engines (Atkinson and Stoll, 2008a). Additionally, orthogonal dimensions and viewpoints are defined for the Archimate approach (Atkinson and Tunjic, 2014b). The activities for modeling architectures for enterprise applications are embedded into the general vision of multi-level modeling with orthographic viewpoints (Tunjic, Atkinson and Draheim, 2018) and can be seen as one important application of OSM.

Methodologists manually create the SU(M)M when applying OSM

Methodologists using the OSM approach create the SUMM (and the SUM if necessary) by hand from scratch, i. e. OSM is a top-down approach starting with the SUMM and viewpoints are defined afterwards on top of the SUMM. Therefore, methodologists can not directly reuse existing metamodels and models, i. e. Requirement R 2 (Reuse existing Artifacts)☞155 is not fulfilled. As benefit, the desired SUMM can be created completely free in an optimized way, without being restricted by existing metamodels to reuse. By this top-down design, an optimized SUMM as in Figure 3.10 for the ongoing example can be realized. Beyond that, OSM provides no explicit method to cope with the challenge to create the optimized SUMM, but requires such methods (Atkinson, Stoll et al., 2013). As look-ahead, the MoConseMI approach of this thesis provides such a method.

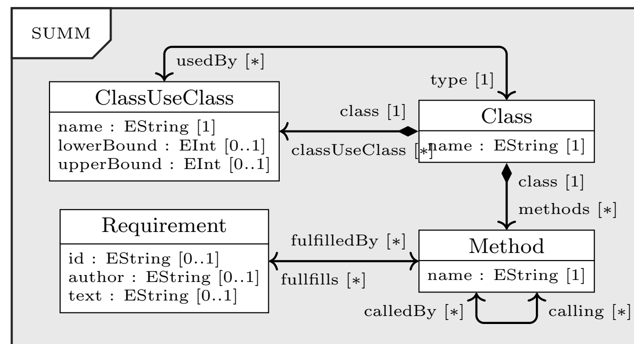


Figure 3.10: Exemplary Metamodel for SUMM in OSM (taken from Meier, Werner et al. (2020))

OSM Implementation

The implementation of OSM is done in nAOMi (Atkinson, Stoll et al., 2013) and use MelanEE (Atkinson and Gerbig, 2016) as technical space which supports multi-level modeling (MLM) (Atkinson, Gerbig and Tunjic, 2013a): Some basics for the understanding of metamodels in MLM are already given in Section 2.2.2☞60. Therefore, transformations between SUM and views require model transformation approaches which support multiple levels like DeepATL (Atkinson, Gerbig and Tunjic, 2013b) as extension of ATL (Jouault, Allilaire et al., 2008). A generic visual language to visualize multi-level models in UML-like style (Atkinson, Kennel and Goß, 2011) is complemented with facilities to develop domain-specific textual and graphical languages for multi-level models (Atkinson and Gerbig, 2013).

3.5.2 Vitruvius

The "VIew-cenTRic engineering Using a VIrtual Underlying Single model" (Vitruvius) approach (Kramer, Burger and Langhammer, 2013; Klare, Kramer et al., 2021) aims to

fulfill the projectional SUM idea in pragmatic and scalable way. In contrast to the explicit SU(M)M in OSM, Vitruvius uses a virtual SU(M)M which reuses the existing (meta)models as parts of it, as depicted in Figure 3.11. This modular SU(M)M consists of the (meta)models (Java, Req and ClassDiagram with package icons in Figure 3.11) which are reused in unchanged and non-invasive way (VP_{Req}, VP_{Java}, $VP_{ClassDiagram}$). Therefore, Figure 3.12 shows the SUMM for the ongoing example with Vitruvius with the three reused metamodels. Vitruvius is bottom-up and realizes an easy reuse of existing (meta)models, but resolves none of their dependencies. Therefore, an explicit mechanism for consistency preservation between the single (meta)models (CPR in Figure 3.11) is required. The reused (meta)models are provided to users as initial view(point)s. An additional approach is required to define new view(point)s. Both mechanisms are presented in the following paragraphs.

modular SU(M)M, composed of reused (Meta)Models with Correspondences

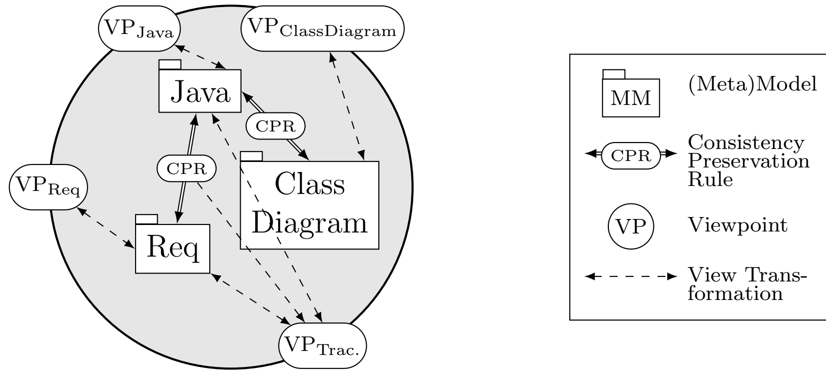


Figure 3.11: SUM approach Vitruvius (taken and slightly adapted from Meier, Werner et al. (2020))

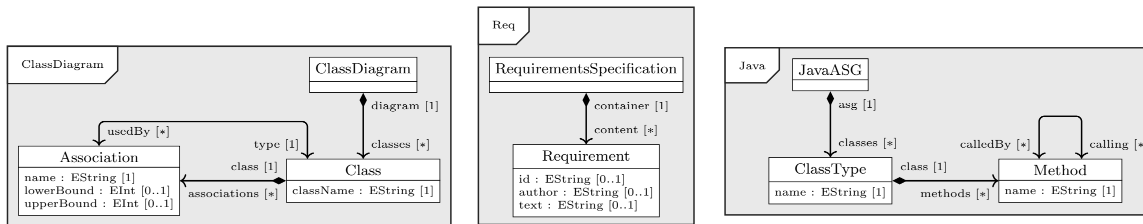


Figure 3.12: Exemplary Metamodel(s) for SUMM in Vitruvius (taken from Meier, Werner et al. (2020))

Vitruvius uses Consistency Preservation Rules (CPRs) to keep depending models in the SUM consistent to each other in pair-wise way. The CPRs use and maintain explicit links in form of correspondences which can be annotated with further metadata between related elements in two different models. CPRs are change translation-based by reacting on changes made by a user in one model and deriving corresponding changes for the other model to keep it consistent (Kramer, 2015). With this design, Vitruvius fulfills the projectional SUM idea for users having multiple views with automated consistency preservation (Requirement R 1 (Model Consistency)☞ 154) by internally using synthetic techniques.

Ensuring Consistency in Vitruvius

Since this change translation-based approach requires consistent models connected with correspondences *before* a user changes one view, the reuse of existing models in Vitruvius is challenging, when these models are not consistent to each other or not connected: Since the mechanisms for consistency preservation react only on user changes, more complex strategies for the initial import of existing models into Vitruvius are required. Leonhardt, Hettwer et al. (2015) propose two strategies to create corresponding elements in other models together with correspondences when importing models conforming to the *same* meta-

Reuse existing Models

model, i. e. first by inserting the current elements as active delta into the change-driven consistency preservation mechanism and second by explicitly creating linking other models using forward or reverse engineering techniques. Importing multiple models conforming to *different* metamodels and importing models which are not consistent to each other is targeted by Mazkatli, Burger et al. (2018): Corresponding elements in different models are searched by using the defined mappings (see below) and additionally defined identifiers. Missing corresponding elements are created and conflicting values of found matches are edited, whose resulting changes are synchronized in the usual way of VITRUVIUS. Therefore, VITRUVIUS supports Requirement R 2 (Reuse existing Artifacts) ☞ 155, but with some restrictions. These findings motivate to concretize Requirement R 2 (Reuse existing Artifacts) ☞ 155 with the following Requirement R 2.3 ☞ 156:

Requirement R 2.3: Fix existing Models

The approach must allow to fix inconsistencies within reused models.

Multi-Model Consistency

Another challenge is to realize consistency between multiple models (Klare, 2018): Since consistency is ensured in pair-wise way, redundant information in three or more models require a dense or even complete graph of consistency specifications between them, in order to fulfill modularity, i. e. the possibility to use arbitrary subsets of these models. Since changes can be propagated transitively, there are multiple execution paths, which could be contradicting and leading to termination problems (Klare, Syma et al., 2019). As solution, such redundant concepts should be made explicit with concept metamodels, which form trees together with the concrete metamodels as leaves: This design aims to prevent contradicting consistency specifications by design and to keep modularity (Klare and Gleitze, 2019).

Define new View(point)s with MODELJOIN

In order to provide new view(point)s which are derived from the concepts and information of the modular SU(M)M (e. g. $VP_{Trac.}$ in Figure 3.11 ☞ 127), MODELJOIN is design as textual DSL (Burger, 2013b, 2014): Using this DSL, the methodologist can select and combine concepts from multiple metamodels of the SUMM, which lead to a new viewpoint and its conforming new view, which can be used by users. Therefore, methodologists can rapidly define *flexible views* and their viewpoints (Burger, 2013a). Similar to SQL, MODELJOIN defines declarative queries on different metamodels and provides concepts for joining, projection, selection and aggregation of model elements. With MODELJOIN it is not possible to define any viewpoint, but only elements already existing in the input metamodels and models can be explicitly selected to be part of the new viewpoint and view (keywords for the textual syntax in round brackets) as defined in Burger, Henss et al. (2014):

- Classes can be reused for the new viewpoint in same form, renamed or joined with other elements (`natural join`, `outer join`, `theta join` with any conditions as OCL expressions). Super-classes and sub-classes of such classes must be explicitly specified to be in the viewpoint (`keep supertype`, `keep subtype`).
- Attributes for reused classes can be explicitly selected to be reused (`keep attributes`) or calculated with five predefined arithmetic functions (`keep aggregate`) or arbitrary OCL expressions (`keep calculated attribute`) over any information in the source models.
- References for reused classes can be explicitly selected to be reused (`keep outgoing`, `keep incoming`).
- All elements in the new viewpoint can be renamed (`as [type]`).
- The amount of objects conforming to a class which is part of the new viewpoint can be restricted with any OCL expressions (`selection`).

These restrictions and the creation of internal traces between reused objects during execution allow to enable editability for ModelJoin views (Burger and Schneider, 2016): For each concept of ModelJoin, a fixed strategy for change translation i. e. to propagate updates in the view back into the source models is chosen. Additionally, OCL constraints for each chosen change translation strategy are formulated which define the possible changes in the new view by users, after which the resulting updates in view are still translatable into the source models. These OCL constraints allow to decide, if an updated view is translatable back into the source models, and could be used to inform users during their work on the view about the translatability of their changes without doing the translation. Details of the chosen translation strategies can be found in Schneider (2015). Some changes are not translated, e. g. values of aggregated and calculated attributes remain read-only. Summarizing, Vitruvius in form of ModelJoin support Requirement R 3 (Define new View(point)s) ☞156, but with some restrictions in detail.

Applications of Vitruvius

Completed and ongoing application examples of Vitruvius include among others component-based software systems supported by consistency preservation of architectural models in the Palladio Component Model language (Becker, Koziolek and Reussner, 2007), contracts in the Java Modeling Language and Java source code (Kramer, Langhammer et al., 2015). The last application could be extended with UML component diagrams and configuration files for Eclipse plugins (Kramer and Langhammer, 2014). New viewpoints include Java source code annotated with related components and component diagrams extended with realizing Java classes and interfaces (Kramer and Langhammer, 2014). Other applications include modeling of hardware-system-systems like automotive systems with SysML, AMALTHEA and ASCET (Mazkatli, Burger et al., 2017), electrical engineering with printed circuit boards and electronic circuit simulations (Zimmermann and Reussner, 2018), smart grids for the energy domain (Burger, Mittelbach and Koziolek, 2016) and automated production systems with AutomationML (Ananieva, Burger and Stier, 2018). Another application is to keep architectural models up-to-date (Monschein, Mazkatli et al., 2021), e. g. to keep performance models in software architectures up-to-date in incremental way (Mazkatli, Monschein et al., 2020). An ongoing application is to manage variants and versions of models in a consistent way (Ananieva, Klare et al., 2018).

Vitruvius Implementation

The implementation of Vitruvius uses EMF as technical space. The textual DSL ModelJoin is realized with Xtext for IDE-like tool support. For ensuring consistency within the SUM, there are two languages (Kramer, 2017; Klare, Kramer et al., 2021): Expressions in the unidirectional *reactions language* are triggered by user changes and specify imperative actions as reactions to these changes in order to fix occurred inconsistencies. Expressions in the *mappings language* define consistency goals between related objects of different models in declarative and bidirectional way (Kramer and Rakhman, 2016) and are transformed into expressions of the reactions language for executions and for both directions. While the reactions language is Turing-complete, the mappings language has reduced expressiveness in order to automatically derive unidirectional reactions for both directions.

3.5.3 RSUM

The "Role-oriented Single Underlying Model" (RSUM) approach (Werner and Aßmann, 2018) realizes the projectional SUM idea with a pragmatic SU(M)M composed of loosely-coupled existing (meta)models (Figure 3.13 ☞130). Similar to Vitruvius, by reusing existing (meta)models RSUM is bottom-up and does not resolve any dependencies. Central element in RSUM is the *role* concept of role-based modeling which is used to couple the (meta)models into the SU(M)M and to ensure consistency within the SUM. In order to provide new views with information derived from the SUM, the syntax of ModelJoin of the Vitruvius approach is reused.

The role concepts (Kühn, Leuthäuser et al., 2014) extend object-orientated model-

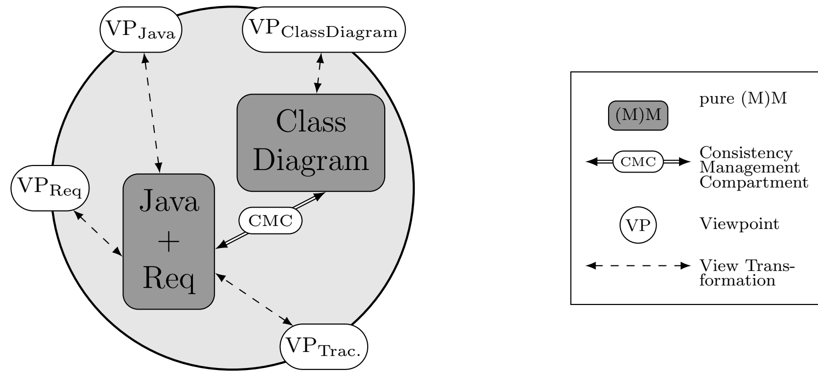


Figure 3.13: Exemplary Metamodel for SUMM in RSUM (taken and slightly adapted from Meier, Werner et al. (2020))

Role-based Modeling

ing, since usual associations between classes are replaced by compartment types and links conforming to associations are replaced by compartments conforming to the compartment type which replaced the association. Within compartments, objects play roles conforming to the compartment types. With this design, compartment types realize usual associations (relational compartments), but in a more flexible way, since compartment types and compartments can be added and removed depending on the current context. Additionally, roles adapt the behavior of objects and interact with other roles, which allows to realize consistency preservation between the adapted objects at runtime (consistency management compartments, CMC). Figure 3.14 uses the concrete syntax for role-oriented modeling of Kühn, Böhme et al. (2015). The role-based technical space makes RSUM suitable in particular for models-at-runtime applications.

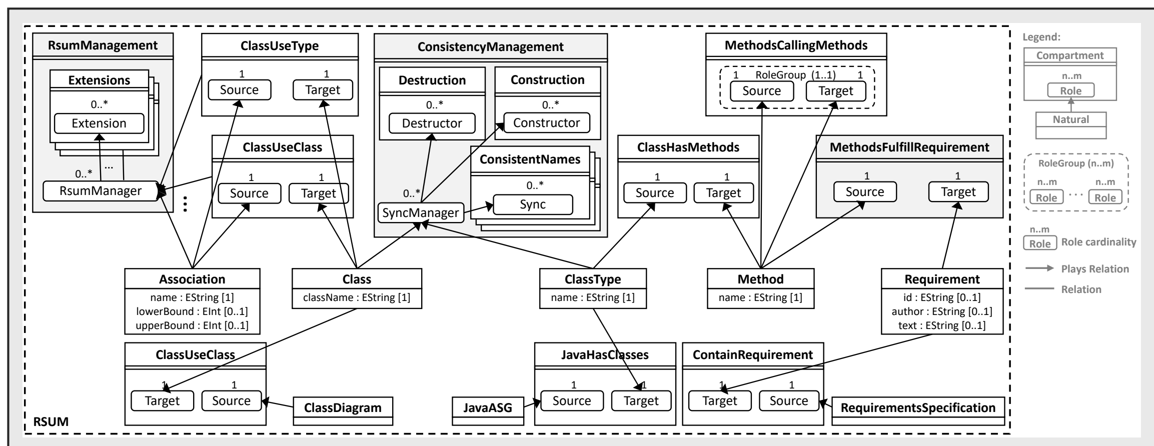


Figure 3.14: SUM approach RSUM (taken from Meier, Werner et al. (2020))

Reuse (Meta)Models

Metamodels to be reused can be transformed into the role-based paradigm by replacing associations by relational compartment types (white compartments in Figure 3.14). Links within models can be extended accordingly to support roles at runtime. Therefore, RSUM supports Requirement R 2 (Reuse existing Artifacts)☞ 155 in general.

Compartments ensure Consistency

The methodologist ensures consistency challenges by defining further compartment types depending on their kind: Explicit links are realized by relational compartment types (gray `MethodsFulfillRequirement` in Figure 3.14), while redundancies and constraints are realized by consistency management compartments (e. g. gray `ConsistencyManagement` for redundant classes in Figure 3.14). Therefore, the metamodels of Java and requirements are combined with a relational compartment type, as depicted in Figure 3.13, similar to an association which would integrate the two metamodels with an association. Redun-

dant classes in ClassDiagram and Java are kept consistent by a consistency management compartment in contrast. One overall `RsumManagement` compartment manages the whole consistency management, in particular, recording of changes by users. Since the CMCs react on these recorded user changes and provide additional changes to fix occurred inconsistencies, RSUM with RSYNC for change propagation (Werner, Schön et al., 2018) falls into the category of change translation-based approaches: Consistency management compartments use different contexts and roles (`Destructor`, `Constructor`, `Sync`) to react on changes for deletion, creation and changes of elements. Bindings of these roles to affected elements work as correspondences and support the direct change propagation. Therefore, RSUM fulfills Requirement R 1 (Model Consistency)☞154. The base RSYNC is extended to PARALLEL RSYNC in order to support concurrent editing of multiple users with conflict handling (Ebert, Kluge and Götz, 2021).

New View(Point)s

Additional new view(point)s (which are not depicted in Figure 3.14☞130) can be defined with the syntax of MODELJOIN (see Section 3.5.2☞126), from which relational compartment(type)s for new view(point)s are generated (Meier, Werner et al., 2020). These relational compartments provide the objects in the SUM as projections to users, whose changes are done directly on the SUM, since the views are virtual and not materialized. Therefore, RSUM inherits the restrictions of MODELJOIN and supports Requirement R 3 (Define new View(point)s)☞156, but with some restrictions in detail.

RSUM Implementation

The implementation of RSUM uses SCROLL (Leuthäuser and Aßmann, 2015), a library for SCALA, to realize the role-based modeling. The current research prototype requires writing compartments, views and consistency management with this library in SCALA by hand (Werner, Bergmann et al., 2019), but DSLs for easier definitions and code generation are desired for future work, in particular, for the definition of views (Werner and Aßmann, 2018) and for synchronization (Werner, Schön et al., 2018).

3.5.4 Combining Views into a SUM

As motivated in Section 1.2.2☞36, often views and their viewpoints already exist and must be reused. For projectional approaches for ensuring consistency of such views, a SU(M)M is required as intermediate structure. In bottom-up projectional approaches, such SU(M)M must be created from the existing view(point)s as starting point, while top-down projectional approaches start with designing the SU(M)M from scratch. Creating an appropriate SUMM is an open question, indicated by OSM (see Section 3.5.1☞124). Another challenge is, that the views to reuse conform to different viewpoints and different metamodels must be reflected in the SUMM. Therefore, this section investigates approaches to combine multiple different view(point)s into a SU(M)M.

Walter and Ebert (2009) present an example for a manual integration of a particular DSL, feature models and ontologies on the metamodel level only.

Already Darke and Shanks (1996) propose the *integration of viewpoints* to be a central activity for managing viewpoints and sketches some approaches, including to merge existing viewpoints into each other to form an integrated metamodel, which could serve as SUMM, and to translate the different viewpoints into the same formal description as in proof-theory-based approaches for consistency (Section 3.2☞99).

Model Merging

Chechik, Nejati and Sabetzadeh (2012) discuss the integration of models using automated model merging operators, which depend on links between related elements which must be provided by users. The presented operators unidirectionally merge only models into each other which conform to the same metamodel.

merge (Meta)Models using Correspondences indicating Commonalities

Stünkel, König et al. (2018) present a very similar approach, but define commonalities i. e. the direct overlaps of elements between metamodels with correspondences represented as an additional commonalities metamodel first. As second step, this commonalities metamodel is used to compose a comprehensive metamodel like a SUMM by merging the initial

metamodels. After defining correspondences between the models by hand or by special identification rules, the models can be virtually merged in similar way. If the result of this merge is made explicit, a SUM is available. Since only direct overlaps of elements in metamodels and models can be unified, the SU(M)M can still have dependencies. As future work, Stünkel, König et al. (2018) plan extensions for ensuring consistency on these integrated (meta)models by formulating inter-model constraints on the comprehensive metamodel and checking them on the virtually merged model. Fixes for inconsistencies are derived from completion rules which are added to the constraints.

SU(M)M as explicitly linked View(Point)s

Another general idea for combining views into a SUM is to keep the views as they are, but link them explicitly with each other (see the overall techniques for linking in Section 3.2[☞ 99]). This idea is realized in the RSUM approach (Section 3.5.3[☞ 129]) as an example. Note, that the resulting SUM contains all information of all views, but with redundancies. A corresponding SUMM can be created with the same idea.

aspect-oriented Modeling and Weaving

Redundancies between multiple views can be treated as crosscutting concerns, which motivate the use of *aspect-oriented modeling* (Wimmer, Schauerhuber et al., 2011): Kienzle, Al Abed and Jacques (2009) provide an approach for multi-view modeling supported by aspect-oriented modeling in the domain of UML diagrams (classes, states, sequences): It allows to define directed acyclic graphs (DAGs) of dependencies between aspects of multiple models. These dependencies are used to weave the different models into one model representing the whole system. The used weaver checks also semantic consistency between the UML models, but does not fix inconsistencies. However, it is unclear, how easy the approach can be generalized to other domains outside of UML.

Model Weaving is project-specific and unidirectional

Jézéquel (2008) even defines, that "*modeling is the activity of separating concerns*" (Jézéquel, 2008, p. 210) as aspects, while "*the design process can be characterized as a weaving of these aspects into a detailed design model*" (Jézéquel, 2008, p. 210). Analyzing and discussing examples with sequence diagrams, Jézéquel (2008) summarizes, that there is no generic model weaving approach, since defining patterns (called pointcuts) and composing the aspects into the matched patterns (called join points) both must take the semantics of the models into account and are project-specific therefore. While aspect weaving enables to generate the design model whenever the base models or aspect models evolved, it is not possible to propagate changes in the design model back into into its source models, since weaving is a unidirectional model transformation. Therefore, model weaving can be used to create a SUM from single views, sometimes with their consistency is precondition, but is not usable to enforce consistency (Requirement R 1 (Model Consistency)[☞ 154]).

extend Base Metamodels with Aspect Metamodels for Metamodel Evolution

Jung, Heinrich et al. (2014) propose to extend base metamodels with aspect metamodels in a non-intrusive way, i. e. without explicitly weaving them. Main objective seems not to ensure consistency between base and aspect models, but to keep mainly existing editors for DSLs, model transformations and simulations working on the unchanged base metamodel and conforming models, while they are extended at the same time in non-intrusive way to overcome the challenge of evolving metamodels.

UML Profiles

UML Profiles are not usable as technical space (see Section 2.5.1[☞ 84]), but could be used to combine multiple viewpoints into one metamodel. Since UML profiles are defined by the UML super structure, this idea works only for viewpoints whose metamodels are part of the UML super structure or are realized as UML profiles. Egyed, Zeman et al. (2018) sketch a corresponding example combining UML models and CAD components. To overcome this limitation in general, the border between UML profiles and other technical spaces like EMF must be bridged, e. g. by transforming UML models to EMF models and vice versa (Maro, Steghöfer et al., 2015) or by explicitly linking the UML profile with the EMF metamodel using model weaving in order to generate transformations on model level (Abouzahra, Bézivin et al., 2005). Maro, Steghöfer et al. (2015) discuss some more approaches for bridging UML profiles with EMF. Related studies show, that the use of UML profiles seems to be decreasing in general, probably due to the increasing quality of

language workbenches for DSLs (Pardillo, 2010, p. 416), which is supported by evaluations in detail like of Bernardino, Rodrigues and Zorzo (2016), who report on faster modeling with a DSL compared to a UML profile.

EMF Profiles

Since EMF does not provide a profile mechanism, Langer, Wieland et al. (2011) adapt the concept of profiles as introduced by UML for EMF: While UML profiles are a specific concept for UML and not generic for MOF, EMF profiles allow to define extensions with stereotypes and tagged values for arbitrary existing metamodels modeled with ECore. Such extensions result in EMF models representing the defined profiles and conforming to the EMF metamodel presented by Langer, Wieland et al. (2011) representing the concepts for profiles in EMF. These EMF models are automatically transformed into EMF metamodels (model-to-metamodel-transformation), whose instantiations allow to apply the profile for regular EMF models in a non-invasive way. Instead of using this EMF profiles mechanism, the generated EMF metamodels could be created directly by hand, which would allow to introduce also non-stereotype `EClass`es.

Composition of DSLs

The composition of multiple, overlapping languages, i. e. DSLs or Profiles, called the *language extension problem* (Leduc, Degueule et al., 2020), requires the composition of multiple abstract syntaxes and the composition of multiple concrete syntaxes (Noyrit, Gérard et al., 2010): The composition of abstract syntax is related to the integration of viewpoints, as discussed above. But the additionally required composition of concrete syntaxes is not required here when keeping models consistent to each other. Therefore, related work for composing concrete syntaxes is out of scope.

Composition for integrated Applications

Composition of Software Building Blocks with AOP according to composed i. e. linked Domain Models

Other approaches aim to compose executable software applications from multiple parts: Estublier, Ionita et al. (2009) and Estublier, Vega and Ionita (2005) aim to compose applications for different domains by targeting two dimensions: In the first dimension (called horizontal), domains are described with DSLs and the developed (meta)models are composed by annotated links between them. In the second dimension (called vertical), software building blocks like tools, services, libraries, legacy applications and components are developed, configured and executed according to their domain models. These software building blocks are composed by aspect-oriented programming to enable communication between them. This approach is synthetic by explicitly linking corresponding elements (for the domain models and for the software building blocks), but focuses mainly on the combination of bigger software building blocks at runtime, less on the consistency preservation of the involved domain models (Requirement R 1 (Model Consistency) ☞ 154) at development time. Therefore, this approach focus more on tool integration with the help of composing the corresponding domain models. Additionally, the redundancy within the composed domain model can not be reduced.

Composition of low-level Models by Links

Yie, Casallas et al. (2010, 2009a) try to avoid a strong composition of high-level models conforming to different metamodels by transforming them into low-level models conforming to the same metamodel, e. g. into models for general-purpose programming languages. These low-level models are combined automatically by transforming links between elements of the high-level models into links between corresponding elements of the low-level models. This approach convinces not completely, since the number of possible links between models increase with growing number of models (Yie, Casallas et al., 2010, p. 239) and the approach requires completely automated transformations from high-level models to low-level models, which is not always possible.

Summary

Summarizing approaches for combining views into a SUM, they usually use manual techniques to define corresponding elements in different models as first step. This step is manual, since correspondences dependent on the semantics of the models, which is project-specific. If these correspondences are materialized as explicit links between (meta)models, a SU(M)M with all remaining dependencies results. In an optional second step, the specified correspondences are automatically exploited to merge corresponding elements of different models in order to improve the SU(M)M by reducing redundancies. But these transfor-

mations for merging are unidirectional and do not allow to update the source models after their merge anymore, therefore hurting Requirement R 2 (Reuse existing Artifacts) ☞ 155. Since profiles add additional information (or restrict existing concepts), but are not able to restructure or to remove already existing concepts, it is not possible to reduce redundant information in the SU(M)M. Composing DSLs or complete software applications often depend on composed models. Often the combination of views into a SUM requires the views to be consistent to each other: Therefore, executing combination approaches can be used as checking for consistency, but the combination approaches usually do not support fixing inconsistencies.

3.5.5 Projectional View Definition

This section presents some selected approaches to define new view(point)s in the projectional setting, since there are approaches which allow to define new views, that do not support consistency of existing data sources in projectional way: Such approaches could be integrated with projectional consistency approaches in order to define new views according to Requirement R 3 (Define new View(point)s) ☞ 156. The easiest way is to create a new view(point) manually and treat it as existing data source using the mechanisms for consistency preservation, but there are more advanced approaches specially designed to define new views. Bruneliere, Burger et al. (2019) provide a survey for such approaches.

new View(Point) on top of one View(Point) i. e. the SU(M)M

Important in the projectional setting is, that new view(point)s are defined on top of *one* existing view(point) i. e. the SU(M)M, even if the new view should contain concepts of different existing data sources, since the SUM contains all their information. Only such approaches are presented in this section. Approaches which define new views on top of multiple existing views could be used here, too, but are presented in Section 3.3.2 ☞ 119. Keeping a new view consistent to the existing views is comparable with keeping existing views consistent to each other, since all views (the new view and the existing views) are synchronized directly with the SUM and contain only information which is already present in the SUM. Therefore, for both kinds of views, asymmetric approaches are sufficient to keep the views consistent with the SUM.

asymmetric Approaches are sufficient

editable new View(Point)s as Sub Set of the Elements of the SU(M)M

Cicchetti, Ciccozzi and Leveque (2012) derive new viewpoints from an existing SUMM by selecting a subset of the elements available in the SUMM: To decide, if an element of the SUM (mainly classes, attributes, associations) should appear in the new viewpoint or not, eases the definition of new viewpoints by the methodologist, e. g. by providing graphical dialogs (Cicchetti, Ciccozzi and Leveque, 2011). Additionally, this approach eases the backpropagation of changes in the new view to the SUM and to the other views, since changes in the view are directly applicable to the SUM, since the new view(point) is a direct subset of the SU(M)M. The drawback of this approach is, that restructurings of the concepts in the SUMM and additionally computed concepts are not possible. Instead, the quality of the SUMM is inherited to the new viewpoints, which cannot be tailored to all needs of the users.

similar Approaches

The same strategy is used and shortly described by de Lara, Guerra and Vangheluwe (2006) in order to keep multiple UML diagrams consistent to each other. In similar way, Guerra, Diaz and de Lara (2005) support multiple view(point)s for one meta(model) for the development of DSLs, including existing data sources called system views and new views called derived views (Guerra and de Lara, 2006).

Formalizations with injective Graph Morphisms

According to the presented practical approaches, Ehrig, Ehrig et al. (2008) use typed attribute graphs to formalize viewpoints as injective graph morphisms $vp : \mathit{Viewpoint} \rightarrow \mathit{SUMM}$, meaning that the viewpoint contains only, but not necessarily all elements of the SUMM. Additionally, the selected elements might be renamed in the viewpoint. Ehrig, Ehrig et al. (2008) provide similar formalizations for views as injective graph morphisms $v : \mathit{View} \rightarrow \mathit{SUM}$. These findings motivate the Requirement R 3.2 ☞ 157 in order to concretize

Requirement R 3 (Define new View(point)s) ☞ 156:

Requirement R 3.2: New Viewpoints with arbitrary Metamodels

New viewpoints must be able to use arbitrary metamodels.

Jakob, Königs and Schürr (2006) present an approach to define non-materialized views on top of a single source model like the SUM. They adapt TGGs in order to keep one side (the view) and the correspondences virtual and to have only the other side (the SUM) explicitly materialized. Since such new views are editable, this approach is helpful in projectional approaches for realizing new views on top of the SUM.

editable virtual new Views using adapted TGGs

Anjorin, Rose et al. (2014) restrict TGGs to View Triple Graph Grammars (VTGGs) for the asymmetric case in order to increase the performance of transformations for views on a SUM.

restricted TGGs for editable Views

Wang, Gibbons and Wu (2011) formally define incremental updates, which can be used to keep new views consistent to a SUM, but are restricted to tree-like data structures.

Formalizing incremental Updates for Views (on Trees)

Summarizing approaches for defining new view(point)s in projectional settings, simple approaches exactly design for this setting provide new view(point)s as subsets of the elements provided by the SU(M)M, easing their specifications and fulfillment of the view-update problem. Additionally, generic approaches for model synchronization including bidirectional model transformations (Section 3.3.1 ☞ 108) can be used with the SUM as source model and the new view as target model. Using these generic approaches for this purpose is simplified on the one hand, since the setting here is asymmetric and not symmetric as in synthetic settings. Therefore, sometimes these generic approaches are restricted to improve the support for this setting, e. g. TGGs. On the other hand, the metamodels for new viewpoints must be created before by hand.

Summary

3.6 Further Research Areas

After identifying several generic approaches for ensuring consistency, this section analyses some exemplary approaches, which are designed for single application domains or other research areas. In particular, approaches for consistency of spread information outside the modeling domain are analyzed, which can show much research for ensuring consistency or combining spread information. This list of further application domains is not complete and presents only some prominent examples. Of course, there are more research areas facing consistency challenges like the following ones:

further Research Areas with Consistency Challenges

- Franzago, Ruscio et al. (2018) discovered several projectional and synthetic approaches for *collaborative modeling*.
- Feldmann, Herzig et al. (2015a) reviewed consistency management approaches and discussed them regarding their applicability for *mechatronic manufacturing system design*.
- When supporting MDSD with *software product line engineering*, Corrêa (2011) identified consistency challenges between the involved artifacts, in particular, between the features and the software product under development, but did not provide solutions. The survey of Pol'la, Buccella and Cechich (2021) supports the importance of consistency for variability modeling. Buchmann and Westfechtel (2014) ensure consistency between a feature model and domain models: After relating an element in domain models for a feature, all dependent elements are related to the same feature, which mainly targets conformance of domain models to their domain metamodels and intra-model consistency in a synthetic setting.

Outline

Some of these and of the following approaches are already investigated above, if they contain generic ideas which are usable for different application domains. UML is analyzed in Section 3.6.1 as an example for a language with multiple overlapping diagrams, which is widely used during the development of software systems. Language workbenches allow to develop such languages with overlappings representations, whose build-in mechanisms for ensuring consistency are analyzed in Section 3.6.2 ☞ 137. Since they are mainly designed and used for storing and representing information, data bases (Section 3.6.3 ☞ 139) and ontologies (Section 3.6.4 ☞ 142) are analyzed. As an example for an application domain with the strong need to manage a huge amount of distributed information, enterprises are selected and analyzed in Section 3.6.5 ☞ 144. Additionally, the existence of several related approaches for discussing consistency in the following research areas shows, that it makes sense to select them for discussion.

3.6.1 UML

UML with a single Metamodel and multiple Viewpoints …

The Unified Modeling Language (UML) is a language to model software-based systems introduced by the OMG and specified by the UML super structure (Object Management Group, 2017). UML provides several diagram kinds to model the system under development regarding static aspects by e. g. class diagrams and object diagrams and dynamic aspects by e. g. sequence diagrams, activity diagrams, use case diagrams and state machines. These diagram kinds can be treated as viewpoints. The particularity of consistency challenges with UML is, that UML provides a single metamodel, which can be split into multiple viewpoints (diagram kinds). Therefore, UML can be seen as projectional. Accordingly, Paige, Brooke and Ostroff (2007) assume a single metamodel involved in managing consistency.

… has Consistency Challenges

Aziz Ahmad and Nadeem (2010) identify groups of consistency challenges for class diagrams, sequence diagrams and state machines, classified regarding among others structure vs behavior and type vs instance level.

Surveys

There are several surveys investigating approaches targeting consistency in UML: In a recent survey, (Knapp and Mossakowski, 2018) identify 57 approaches for consistency in UML, classified regarding their support for UML1 or UML2. They report, that existing approaches support only some diagram kinds, in particular, not more than six diagram kinds. Lucas, Molina and Toval (2009) identify 32 approaches for consistency for UML diagrams and found limited extendability and limited support for consistency of models at different levels of abstraction as recurring properties. Additionally, they sketch another approach for consistency management in UML using MAUDE as formal specification for a SUMM with model transformations written in QVT-R. Usman, Nadeem et al. (2008) survey approaches for only checking consistency with a classification depending on the used intermediate representations for UML (formal, extended UML, none). Aziz Ahmad and Nadeem (2010) analyze some of these approaches which use description logic in more detail. Gabmeyer, Kaufmann et al. (2019) present model checking approaches for UML diagrams.

some exemplary Approaches

Egyed, Letier and Finkelstein (2008) identify possible fixes for inconsistencies in UML models automatically and represent a rule-based approach. This approach is extended later for generic models, as presented in Section 3.3.1 ☞ 108. Paige, Brooke and Ostroff (2007) realize consistency for the BON modeling language, having similar diagram kinds as UML, i. e. class diagrams, collaboration diagrams and contracts expressible as state machines in UML.

Summary

Summarizing approaches for keeping different UML diagrams consistent to each other, the surveys found lots of approaches, mainly in the first decade of the twenty-first century and supporting only a small subset of UML diagrams. Therefore, theses approaches can be seen as preceding research for the more general approaches ensuring consistency (including projectional and BX approaches) in terms of history. The sketched approaches specific for UML already use the different techniques for change propagation (Section 3.2 ☞ 99).

3.6.2 Domain-Specific Languages (DSLs)

Domain-Specific Languages (DSLs)

While UML (Section 3.6.1 ☞136) can be seen as general-purpose modeling language, domain-specific languages (DSLs) (Kosar, Bohra and Mernik, 2016) are languages tailored to selected concerns of particular stakeholders. DSLs realize concrete syntaxes for users to edit models conforming to one metamodel. Language workbenches are frameworks to technically realize editors for such DSLs. Usually language workbenches allow to define more than one DSL on top of the same metamodel. Therefore, they follow projectional SUM ideas in the sense of Section 3.4 ☞120.

Additionally, editors for DSLs are distinguished into parser-based editors and projectional editors depending on the kind how editors update the abstract syntax tree (AST) as underlying model (Voelter, Siegmund et al., 2014):

parser-based vs projectional DSL Editors

- In *parser-based editors*, parsing is used to analyze the concrete representation edited by the user and to extract the (updated) AST.
- In *projectional editors*, users change the AST directly by their actions, while the (updated) AST is rendered to visualize it for the users.

This distinction does not target synthetic vs projectional viewpoints, but how the concrete syntax is technically realized for one viewpoint and how the user interacts with the DSL editor (Erdweg, Storm et al., 2013). Therefore, projectional editors realize projectional viewpoints, but the concrete syntax of projectional viewpoints is not always realized by projectional editors. Examples for widely used language workbenches which use parsing are Sirius (Viyovic, Maksimovic and Perisic, 2014) for graphical DSLs and Xtext (Bettini, 2013) for textual DSLs. Spoofax is another example for a language workbench producing parser-based editors (Wachsmuth, Konat and Visser, 2014).

projectional DSL Editor $\Longrightarrow$ projectional Viewpoint

JetBrains MPS

An example for a language workbench whose editors use projections is JetBrains MPS[8] (Voelter, Warmer and Kolb, 2015): Users use no free-form editors, but the DSL provides an editor in form of a "template" which is pre-filled with the information of the current AST. This information can be edited by users, while the template ensures, that changed information is directly updated at the corresponding position in the AST. With this design, no inconsistencies arise, since all changes in views are directly reflected into the underlying AST which plays the role of a SUM. This design follows the idea of OSM how to avoid inconsistencies. JetBrains MPS supports different notations including textual, symbolic, tabular and graphical (by plugins) notations and supports their composition and extension, since projectional editors enable multiple and diverse notations (Voelter, Warmer and Kolb, 2015). The most important application of JetBrains MPS is mbeddr, an integrated set of 81 DSLs with IDE support for developing embedded software (Voelter, Kolb et al., 2019) unifying modeling and programming (Voelter, 2010). As in general for projectional editors, usability is also for JetBrains MPS a challenge (Voelter, Siegmund et al., 2014). To enable direct projections between the concrete syntax and the underlying model, JetBrains MPS requires to have all projectional editors integrated within on tool, which hinders to add new viewpoints only on data level without the dimension of tool integration. This motivates the following requirement to avoid approaches which require to use a single tool as precondition for data consistency:

Requirement R 5: Reusable Library

The approach must be realized in form of a reusable library.

[8] More publications related to JetBrains MPS can be found here: `https://confluence.jetbrains.com/display/MPS/MPS+publications+page`

projectional Editors for EMF

Schröpfer, Buchmann and Westfechtel (2020) started a comparable approach in the technical space EMF to define projectional textual syntaxes for EMF models. This approach is extended for modeling multi-variant models for software product line engineering (Schröpfer, Buchmann and Westfechtel, 2021).

The development of DSLs and language workbenches touches also other directions of related work:

DSLs with additional Constraints

- While the discussion of parser-based vs projectional editors focuses on the question, how to model models which conform to the metamodel of the DSL, *additional constraints* like OCL constraints defined for the metamodel must be ensured, too: Neubauer, Bill et al. (2017) present an approach to generate editors which ensure such additional constraints using rule-based approaches for ensuring consistency, as discussed in Section 3.3.1☞[108], while they are applied here for *intra-model consistency* of a single DSL.

Integration of DSLs

- The *integration of multiple DSLs* covers not only the metamodel level, which is discussed in Section 3.5.4☞[131], but also the concrete syntax with additional challenges regarding composing grammars (Kats, Visser and Wachsmuth, 2010) for parser-based editors, while projectional editors can be composed easily, since they do not require sometimes ambiguous grammar composition (Voelter, 2010).

Developing DSLs with multiple Viewpoints

- Additionally, the development of DSLs itself often *involves multiple viewpoints*, usually at least one viewpoint to define the abstract syntax and one viewpoint to the define the concrete syntax of the DSL under development: While these viewpoints again rise consistency and combination challenges, usually the models of views of the current DSL are used for code generation for the resulting editor for the DSL without round-trip facilities. Examples for language workbenches with multiple viewpoints to develop DSLs include GMF, Sirius (Viyovic, Maksimovic and Perisic, 2014) and Kermeta (Jézéquel, Combemale et al., 2015). Finally, these approach have to deal with the challenge of evolving abstract syntaxes like GMF-co-evolution (Di Ruscio, Lämmel and Pierantonio, 2011, p.152f), which is discussed in Section 6.2.1☞[193] as impact of metamodel evolution.

Blended Modeling

Blended modeling (Ciccozzi, Tichy et al., 2019) generalizes the idea of having DSLs with concrete syntaxes tailored to users concerns by researching, how to enable multiple concrete syntaxes respectively notations for the same metamodel: Therefore, blended modeling can be seen as projectional from an conceptual point of view, but is orthogonal to multi-perspective modeling, which concentrates on view(point)s with their (meta)models (View-Model-relationship in Figure 2.22☞[90]). Consistency in blended modeling targets to propagate changes between the view (respectively its model) and all its concrete renderings (Model-ConcreteRendering-relationship in Figure 2.22☞[90]) according to the concrete syntaxes coming with the viewpoint of the view.

Marama Language Workbench with Subset Views and Reactions on Changes

Grundy, Hosking et al. (2013) present the language workbench Marama for visual DSLs, each realized by one SU(M)M including constraints and multiple view(point)s with graphical representations. SUM and views are kept consistent, since changes to the view are directly applicable for the SUM, since views present only a subset of the elements of the SUM. Additionally, changes in views can be used as events to trigger further actions, which can be used for follow-up changes (Grundy, Mugridgett and Hosking, 1998), e. g. to fix inconsistencies.

Summary

Summarizing consistency in DSL research, language workbenches for DSLs usually follow projectional *concepts*, but *technically* realize editors either parser-based or with projectional editing. Therefore, they lead to a stronger binding of viewpoints to concrete tooling. Research regarding consistency of multiple DSLs focuses on how to keep multiple concrete

syntaxes consistent to the underlying model. Depending on parser-based or projectional editing, language workbenches provide different concepts for DSL engineers with build-in strategies to ensure consistency.

3.6.3 Data Bases

Scope

Since data bases are designed to store and persistent data in software applications, this section investigates research approaches in the area of data bases for keeping redundant data consistent, integrating data and deriving new views. This section restricts itself to relational data bases ignoring, among others, key-value-based and document-based data bases (Lu and Holubová, 2019), since relational data bases have a clear schema, which does not always (explicitly) exist, e. g. for NoSQL approaches (Roy-Hubara and Sturm, 2020). Such schemata are similar to metamodels in the modeling domain. The evolution of data base schemata is discussed in Section 6.2.1 ☞193.

Combination of Data Bases

An important challenge in data base research is, how related or redundant data located in different data bases can be combined. For this, the following approaches are shortly described:

- integrate data bases into one data base
- ETL and data ware houses
- federated data bases

mediated Schema ≈ SUMM

A central role in these approaches plays the *mediated schema* (Doan and Halevy, 2005), which provides the concepts of all original schemata, comparable with the SUMM in modeling. Distributed data bases are not investigated here, since they have a single schema and the conforming data are spread over multiple data bases, which lead to challenges of data replication (Section 1.3.2 ☞43), but not of consistency between heterogeneous data.

Schema Matching

Schema Integration

Schema Mapping

The combination of data stemming from multiple data bases can be made explicit by introducing a new data base containing all concepts and all data of all original data bases as data sources (Helms, 2020). Usually, this is realized with the three steps[9] schema matching, schema integration and schema mapping (Özsu and Valduriez, 2020): *Schema matching* identifies semantic relationships called mappings between two independent schemata, surveyed by Bernstein, Madhavan and Rahm (2011) and Rahm and Bernstein (2001), who found, that there are lots of different techniques to find matching elements, but still require humans for final validation, in particular, since deciding the information capacity equivalence (Miller, Ioannidis and Ramakrishnan, 1993) is undecidable (Miller, Ioannidis and Ramakrishnan, 1994). This counts even for approaches which focus on semantics in terms of semantic relationships and between concepts and their meanings in the schema (Giunchiglia, Shvaiko and Yatskevich, 2005). *Schema integration* takes the mappings found by schema matching and uses them to create the mediated schema. *Schema mapping* transforms the data from the original data bases into the new data base according to the mappings and conforming to the mediated schema. Batini, Lenzerini and Navathe (1986) analyze different strategies for schema matching and schema integration, while Batini and Lenzerini (1984) present an example for such an approach. The tool support for these three steps is poor (Bernstein, Madhavan and Rahm, 2011; Skok, 2020). In the data base research, the original data sources are ignored after the integration and are not kept up-to-date. Therefore, this approach can be seen as data migration, while the general integration idea is projectional. This approach relates to the question in terms of modeling, how a SU(M)M can be created from multiple independent (meta)models, as discussed in Section 3.5.4 ☞131.

[9]Other authors like Batini and Lenzerini (1984) comprise schema matching and schema integration as schema integration only.

Combination by Integration vs Combination by Synchronization

The previous paragraph discussed, how a mediated schema can be provided by explicitly integrating the data ("combination by integration" (Helms, 2020)). Instead of this explicit data integration, the data could remain within the original data bases and be accessed via the mediated schema with the global-as-view approach ("combination by synchronization" (Skok, 2020)): While this approach allows reading the data of all source data bases, but without reducing duplicates, writing changes is usually impossible. As an example, the SQL operator `UNION` used to return elements from two data bases leaves the question open, in which of the two data bases a new element should be added. In general, to propagate changes back, the view-update problem must be solved, as discussed in following paragraphs.

ETL

Other approaches follow the ETL principle "extract-transform-load" (Leser and Naumann, 2007): The extraction step extracts the desired data from the underlying data sources. The transform step transforms the extracted data into the format used within the data ware house. The load step executes the extraction and transformation steps depending on required time intervals and performance issues. A prominent example for ETL approaches are *data ware houses*, which collect and store various data in order to present them for analyses and decision-making (Chandra and Gupta, 2018). Other frameworks for integration data following the ETL principle are shortly surveyed by Sharma, Tripathi and Srivastava (2021) including GOBBLIN (Qiao, Li et al., 2015) as an example. Even when treating the extracted and collected data as SUM (with pure quality and possible redundancies) with derived new views, the main problem remains unsolved, that all these data are provided read-only, i. e. no propagation of changes back is possible by design of ETL. The underlying data sources and their use remain as they are, independently from data ware houses.

Data Ware Houses

federated Data Bases

Federated data bases provide a mediated schema in order to provide a unique access to multiple underlying data sources (Leser and Naumann, 2007): This mediated schema contains only the concepts of the underlying data sources which are explicitly exported by them for this purpose (Leser and Naumann, 2007). Therefore, the mediated schema is not always complete in contrast to a SUMM. The underlying data sources remain active and can be still used independently from the mediated schema.

Model Management Operators

In all analyzed cases, the mediated schema can be realized using techniques for schema matching and schema integration, sometimes combined as schema merging. These generic techniques are operationalized as *model management operators* (Bernstein and Melnik, 2007): These operators realize transformations with schemata and mappings between schemata as inputs and outputs. These operators are generic for the data base research area and therefore are not able to realize project-specific situations like project-specific consistency goals. Therefore, they are not able to fulfill Requirement R 1 (Model Consistency)☞154. Some examples for such model management operators include schema matching, mapping composition, mapping inversion, schema difference calculation and schema merging (Bernstein and Melnik, 2007; Melnik and Bernstein, 2004). Bernstein (2003) shows model management operators and their application to schema integration, schema evolution and round-trip engineering, but on conceptual level.

Views of Data Bases

Data base *views* present information which is derived from the materialized relations of the data base, often defined by SQL queries. In particular for materialized views, as used in data ware houses, it is important to keep them up-to-date in efficient way, which can be realized with incrementality (Gupta, Mumick and Subrahmanian, 1993; Mohania, Konomi and Kambayashi, 1997). This involves complex algorithms for incrementality for different kinds of SQL expressions like aggregate and outerjoin (Gupta and Mumick, 2006). Varde and Rundensteiner (2002) present an approach to keep views provided by data ware houses consistent to changes in multiple underlying data sources in an incremental way.

incremental Views

View-update Problem

More interesting is the *view-update* problem arising from the question, whether and how to propagate changes in the view back into the underlying data base: To propagate

updates of relational views into the underlying data base relations, among others, Bancilhon and Spyratos (1981) and Dayal and Bernstein (1982) investigate view-update strategies and conditions. In general, a view-update operation to update the underlying relations does not always exist or is not unique (Dayal and Bernstein, 1982) or there are infinitely many repair actions (Dam, Egyed et al., 2016, p. 138). Therefore Tran, Kato and Hu (2020) develop an approach to explicitly write code to realize the view-update problem. While the view-update problem stems from the data bases area, it is also applied to tree structures (Foster, Greenwald et al., 2007). Comparing with the modeling domain, the view-update problem is a historical precedent of BX (Abou-Saleh, Cheney et al., 2018).

Data Provenance

The *provenance* of information investigates "*the origin, context, or history of data*" (Cheney, Chong et al., 2009) and is a research area cross-cutting domains like modeling, data bases or ontologies, since they all handle with information. Data provenance is discussed here with the focus on data bases, since most research for provenance is focusing on data managed with data bases. When combining multiple data bases as discussed above, knowing the initial source of information after the combination is necessary for updating the initial data sources in terms of Requirement R 1 (Model Consistency) ☞ 154 and Requirement R 2.2 (Reuse existing Models) ☞ 156, but not sufficient. But provenance is not only important when combining multiple data bases, but also when querying information and representing them as views (Rani, Goyal and Gadia, 2015). In both cases, provenance can be seen as links back into the original sources of the current information, while these links can be realized by annotations or by explicit links (Doan, Halevy and Ives, 2012). In any case, provenance requires additional meta-data, whose amount can be challenging, e. g. in big data settings (Wang, Crawl et al., 2015), while "*it is impossible in practice to record all relevant provenance information*" (Buneman and Tan, 2019, p. 5), which shows the need to define context-specific provenance scenarios (Buneman and Tan, 2019). In the area of modeling, explicit links between models, e. g. between source models and target models of model transformations, could be used for provenance, as discussed by Anjorin and Cheney (2019). Since BX is a model synchronization-based approach ensuring consistency between models, they propose, that "*provenance will play an important role in explaining consistency management operations*" (Anjorin and Cheney, 2019).

Summary

Summarizing consistency challenges in data base research, there are lots of approaches for combining, selecting and representing information from multiple sources, but propagating changes back is rarely supported. This counts in particular for combination approaches, ETL approaches including data ware houses and data base views. Therefore, Requirement R 1 (Model Consistency) ☞ 154 and Requirement R 3 (Define new View(point)s) ☞ 156 are usually not fulfilled, since only one transformation direction is supported. Even approaches for data provenance provide only meta-data about the origin of information, but no realization techniques for back propagation up to now. The investigated approaches for composing multiple data base into a single data base like a SU(M)M are either read-only (see above) or do not support the source data bases after integration anymore, leading to restricted support for Requirement R 2 (Reuse existing Artifacts) ☞ 155. Techniques for data base integration require manual effort as for schema matching to create a SUMM and rarely supported by tools. There are two more cross references between research areas within data base research (Doan, Halevy and Ives, 2012): Uncertainty modeling can be used also for the metamodel level, e. g. to enrich automatically found mappings during schema matching with probability information. Another cross reference in data base research is, that data provenance information can help to concretize uncertainty information. Both cross references are not deepened here, since this thesis expects methodologists to explicitly decide on the mappings between metamodels basing on domain knowledge about the data sources.

3.6.4 Ontologies

As an alternative to models, ontologies can be used to describe information. Therefore, ontologies with related or interfering information must be mediated or combined. This section discusses, how some research areas of ontologies are related to the consistency of models.

Ontologies describe Domain Knowledge

Ontologies are a means to describe knowledge in a certain domain (Hesse, 2002) as graph. While being models, ontologies describe domains mainly in analyses phases, in contrast to models used in software engineering, which describe systems mainly in design and implementation phases (Aßmann, Zschaler and Wagner, 2006). Therefore, ontologies can be used to reuse domain knowledge in multiple software engineering projects within the same domain (Horst, Bachmann and Hesse, 2012).

Ontologies allow multi-level Modeling

Ontologies support multiple levels of instance-of relationships, i. e. ontologies support multi-level modeling in general, as sketched in Section 2.2.2 ☞60, but this distinction often blurs (Leser and Naumann, 2007, p. 274f), in particular, using the term ontology does not make clear, if the ontology schema or the ontology instance is discussed. Since this thesis is restricted to two meta-levels, i. e. metamodels and models, ontologies are discussed in this section with two meta-levels called schema and instance, too. The evolution of ontologies is discussed in Section 6.2.1 ☞193 with the focus on the distinction between schema and instance.

open vs closed World Assumption

Note, that ontologies in knowledge representation follow the *open world assumption*, meaning, that elements which are not modeled might exist, so far, it is only unknown, if they exist or not. In contrast, modeling in software engineering follows the *closed world assumption*, meaning, that elements which are not modeled do not exist. This is reflected by the Definition 2 ☞32, which states, that all views together describe the system in its entirety, i. e. there is no more information required for the system outside of the views. In particular, this thesis focuses on ensuring consistency of information which is explicitly known and available.

Technical Space

Ontologies are realized in technical spaces (Section 2.5 ☞84) which are different to technical spaces for modeling, in particular, for EMF. To bridge technical spaces for ontologies and modeling, Rahmani, Oberle and Dahms (2010) present a transformation between OWL for ontologies and ECore with OCL for modeling. An alternative formal approach for unifying ontologies and models is presented by Mossakowski, Codescu et al. (2015). In general, such bridges can be defined on the meta-metamodel level, i. e. the modeling space of models and ontologies are mapped, or on the metamodel level, i. e. the classes in the metamodel are mapped to classes in the ontology schema (Staab, Walter et al., 2010).

Intra-Model Consistency

In order to realize model checking, models can be translated into ontologies, whose reasoning techniques are reused for this purpose (Parreiras, Staab and Winter, 2007). When translating only single (meta)models, only *intra-model consistency*, conformance of models to their metamodel and conflicting constraints can be checked using techniques to check the internal consistency of ontologies like Baclawski, Kokar et al. (2002). The approach of Haase and Stojanovic (2005) is change translation-based and supports arbitrary consistency goals with explicit repair strategies to fix occurring inconsistencies.

Inter-Model Consistency

Discussing relations of ontologies to *inter-model consistency* requires to distinguish two cases, whether the different ontologies target same or different domains:

- If the two ontologies aim to describe the *same domain*, there are many redundancies between them, but usually they do not lead to inconsistencies in the understanding of this thesis: If a particular ontology is accepted for a particular domain, then there are no misunderstandings and no inconsistencies. Otherwise, an alternative domain is developed, probably, with a slightly shifted focus. There is no request to fix the differences between these two versions, since the differences are not seen as inconsistencies, but as alternatives.

- If the ontologies describe *different domains*, redundancies between different ontologies are small, since only concepts which are part of both domains are redundant. Such ontologies can be combined into one joint ontology (see below), but usually do not lead to inconsistencies, since they complement each other.

The need for fixing probably occurring inconsistencies between ontologies is decreased in both cases, since ontologies usually describe domains with a generalized claim instead of systems, compared to models describing the same, concrete system under development, whose inconsistencies lead to failing development projects.

Ontology Matching

Additionally, in both cases, same concepts described in different ontologies can be identified by ontology matching: *Ontology matching "finds correspondences between semantically related entities of ontologies"* (Shvaiko and Euzenat, 2013, p. 158). Additionally, Shvaiko and Euzenat (2013) describe the state of art of ontology matching including references to additional surveys. Since fully-automatic mapping of ontologies seems not be possible in general, i. e. their results require human reviews, heuristics and machine learning algorithms are used (Noy, 2004). After identifying the overlaps of ontologies on schema level, these overlaps must be made explicit, e. g. in form of explicit links or translation rules, in order to use them to translate conforming instance ontologies into each other (Parreiras, Staab et al., 2008). Another use case is to merge ontologies for different domains according to the found correspondences into an integrated ontology in order to describe the domains in integrated way.

Top-down vs Bottom-up

While some ontologies are designed as abstract foundation and are reused and concretized for more specific ontologies (Noy, 2004), which can be seen as top-down procedure, combining ontologies as described above is usually bottom-up (Choi, Song and Han, 2006). While combining ontologies is a means for their reuse in the ontology research (Choi, Song and Han, 2006), combination is a means for ensuring consistency in this thesis.

Ontologies as supporting Technique

Ontologies can be used as supporting technique for challenges in software engineering, with some some concrete examples for ontologies used for software engineering collected by Bernstein (2011). More interesting are applications of ontologies for modeling:

- France and Rumpe (2007) propose the use of ontologies for metamodel integration.
- Walter, Parreiras and Staab (2014) integrate the modeling spaces of ontologies and ECore in order to support the development of and reasoning on DSLs.
- Jin, Cordy and Dean (2003) propose to use ontologies for realizing adapters.
- Feldmann, Herzig et al. (2015b) propose to use knowledge representation formalisms like RDF for describing commonalities of different views in production system development projects.

Ontologies for Data Interoperability

Finally, ontologies can be used to deal with the heterogeneity of data, as shown by this example: In order to enable tool integration regarding data interoperability, Kramler, Kappel et al. (2006) propose to use ontologies for a semi-automatic approach: The concepts of metamodels describing the data of single tools are lifted into ontologies. These ontologies are mapped to a predefined ontology for tool integration and commonalities between them are identified based on these mappings. These commonalities are used to derive bridges implemented in QVT which realize data interoperability between the tools (Kappel, Kapsammer et al., 2006). This approach uses a projectional and fixed SUMM, but without a SUM. As another approach, Hakimpour and Geppert (2001) integrate ontologies in order to integrate equivalent schemata of data bases (Section 3.6.3☞139). El Hamlaoui, Trojahn et al. (2014) transform ECore (meta)models into ontologies and use ontology matching to identify correspondences between different models for consistency purpose. On the other hand, knowledge encoded as ontology can be used as vocabularies or thesaurus for other (schema) matching approaches (Leser and Naumann, 2007, p. 280f).

multiple Views on the same Ontology

Since knowledge representations can become huge, Bork, Buchmann and Karagiannis (2015) propose to use multiple views to represent only parts of the whole amount of knowledge. They distinguish approaches for realizing views into *views-by-generation* conforming to model transformation-based approaches here *and views-by-design* conforming to change translation-based approaches here. Since these views are read-only, consistency is ensured by re-generating the views after changes in the whole knowledge representation.

Summary

Summarizing the sketched research approaches in the ontologies area, consistency plays a more supporting role at various points, but is not the main purpose, in contrast to e. g. BX research: Since particular ontologies as representations of knowledge live by agreements of their particular communities, there is no need for *automated* approaches changing ontologies, even not for ensuring consistency, since changes need the understanding and agreement of the involved communities, making updates of ontologies a manual process. Nevertheless, there are approaches to ensure intra-ontology consistency, which could be applied to multiple ontologies by applying the ideas of multi-models to ontologies. Similar to schema matching in data base research, ontology matching can be used to determine overlapping elements of ontologies as first step for merging ontologies into each other. Ontologies are also used as supporting technique in software engineering including for data interoperability.

3.6.5 Enterprise Applications

Enterprises at Runtime vs at Development Time

In the context of enterprise applications, data occur with two different purposes (Ehrig, Ermel et al., 2015a, p. 328): Data are used by enterprise applications to realize their business goals *at runtime* or data are used by stakeholders of enterprise applications to design and realize them *at development time*. If two or more sources with such data are used due to distributed organizations or acquisitions, there is the need for *enterprise integration* in order to synchronize the different units of enterprises. Giachetti (2004) distinguish four levels of enterprise integration focusing on the runtime dimension:

Levels of Enterprise Integration (at Runtime)

Network targets the physical connectivity of hardware and platforms and is not relevant here.

Data overcomes the separation of heterogeneous data: "*The integration goal is data sharing where two or more subsystems or organizational units exchange data with each other*" (Giachetti, 2004, p. 1151). This aspect is highly related here and is discussed in the following paragraphs.

Application aims at interoperability as "*the ability of one software application to access/use data generated by another software system*" (Giachetti, 2004, p. 1151). This aspect is clear distinguished by Halevy, Ashish et al. (2005) as "enterprise application integration" from "enterprise information integration", their name for the data aspect before. The application level focuses on interoperable interfaces of software applications and components and is related to tool integration (Section 1.3.2 ☞43) and therefore is not related here.

Process targets the coordination of dependencies between processes and of resources required by different processes at runtime and is therefore not related here.

vertical and horizontal Integration

These four levels are called *vertical* integration by Kühn, Bayer and Karagiannis (2003), while *horizontal* integration covers different models within the same vertical level of different business units or partners of a supply chain. Mixing vertical and horizontal integration leads to *hybrid* integration. At development time, the static dependencies between these four vertical levels as well as the horizontal dimension must be taken into account: Business process models (located on the process level) must be realized with underlying IT

Levels at Development Time

applications (application level) which use heterogeneous and depending data (data level), shared via hardware (network level). Therefore some information on other levels than the data level are shortly sketched in this section, too, like business process models at the end of this section. In all cases, heterogeneous and depending data must be consistent: Van Belle (2003) identified consistency as quality criterion for enterprise models, but focuses more on intra-model consistency, while this section focuses on inter-model consistency in the enterprise domain, not explicitly distinguishing horizontal and vertical dimensions.

Since companies *use* multiple data bases for managing their data (Section 3.6.3 ☞139), they have the usual needs to integrate or keep their underlying enterprise data consistent, in order to combine different enterprise applications fulfilling the different business use cases. Means for this are SU(M)M-like approaches, federated data bases and data ware houses (Giachetti, 2004). The term enterprise information integration is clarified and demarcated from data ware houses by Halevy, Ashish et al. (2005), focusing on virtually integrating distributed and heterogeneous data by federated queries on a mediated schema, in contrast to data ware houses, which duplicate and integrate data outside of their sources. Since enterprise information integration (respectively interoperability) usually is read-only, updates of data via defined business process require enterprise application integration (Halevy, Ashish et al., 2005, p. 779, 782). Additionally, they can use ontologies (Section 3.6.4 ☞142) and their integrations to manage understandings of different domains and knowledge required for their business use cases (Giachetti, 2004).

Data Integration at Runtime

The *development* of enterprise applications requires additional concerns and therefore viewpoints. In particular, different viewpoints are used also in enterprise architectures with the example of the ARCHIMATE approach (Atkinson and Tunjic, 2014b). Therefore, the consistency of different views must be ensured, which is aimed by some frameworks in the enterprise domain:

Multiple Viewpoints for developing Enterprise Applications

- As an example, the MEMO approach (Frank, 2014) integrates viewpoints for, among others, object-oriented information modeling, organizations including structures and processes, business strategies and the IT infrastructure into one integrated metamodel. Therefore, MEMO can be realized also as an instantiation of the OSM approach (Tunjic, Atkinson and Draheim, 2018) due to its top-down procedure and the explicit SUMM, which shows the feasibility of OSM and of projectional SUMM approaches for enterprise modeling with multiple viewpoints in general.

- Another example approach for realizing multiple viewpoints is MODELMOSAIC (Delen, Dalal and Benjamin, 2005), whose general design is projectional, but with a SUM as collection of multiple models and change propagation, similar to VITRUVIUS. It supports collaborative development, simulation and code generation, while the different viewpoints and consistency goals seem to be fixed (Fernandes, Li et al., 2009). This approach supports also the mapping of ontologies (Section 3.6.4 ☞142) representing schemata of different business units in order to exchange data between them (Fernandes, Li et al., 2010) at runtime for business application integration (see above).

- An even more generic approach is Dijkman, Quartel and van Sinderen (2008), having viewpoints for behavior, structure and information, establishing explicit links on metamodel and model level and checking OCL constraints for consistency, but without fixing found inconsistencies, while the approach is not restricted to enterprise in general.

- Bork and Sinz (2013) propose a high-level conceptual approach for realizing multi-view modeling and apply it to Semantic Object Model (SOM) business process modeling (see below). Bork and Karagiannis (2014) sketch the MUVIEMOT tool which

should work as workbench for multi-view modeling with some support for consistency between views, but the details are not shown.

Business Process Models

An important means to design enterprise applications are *business process models* to model use cases in the enterprise domain: For business process modeling, consistency is an important challenge, e. g. between business process models and supporting IT models (Branco, Xiong et al., 2014) or between business models and enterprise architectures (Iacob, Meertens et al., 2012), and there are synthetic and projectional approaches to ensure consistency (Awadid and Nurcan, 2019). As an example, Küster, Völzer et al. (2016) support different viewpoints for business analysts and IT experts on the same business process using a SUM called “Shared Process Model” here with a fixed set of viewpoints and fixed consistency goals.

Technical Space

In order to realize consistency also for models from the enterprise domain, technical spaces used in enterprise modeling must be bridged to technical spaces used in modeling for software engineering. As an example, Kern and Kühne (2007) present a bridge between *ARIS* for enterprise architectures and EMF. Generalizing this idea, all information used by enterprise applications and all information about the design of enterprise applications can be treated as views conforming to viewpoints.

Reuse Inter-Model Consistency Approaches for Enterprises

After treating enterprise information as models, all presented approaches for ensuring inter-model consistency can be applied, in particular, synthetic approaches (Section 3.3[☞108]) and projectional approaches (Section 3.5[☞121]). This explicitly counts also for more formal approaches like TGGs (Ehrig, Ermel et al., 2015a).

Summary of Enterprise Consistency

Summarizing consistency within enterprises, lots of approaches for models, data bases and ontologies can be reused for enterprises as an application domain, mainly at their runtime. More specific approaches are available for the development of enterprise applications with specific enterprise views including some support for their consistency. In particular, the consistency between business process models and realizing IT applications is investigated. Also in the domain of enterprises, there is the discussion between explicitly storing the integrated information e. g. within a data ware house and keeping the data of interest within their sources together with a virtual integration e. g. with enterprise information integration, which is similar to the discussion between projectional and synthetic approaches in the modeling domain.

3.7 Summary: Lessons Learned

Objectives of this Section

Objective of this section is to summarize the lessons learned when investigating related approaches for ensuring consistency. Additionally, groups of related approaches are compared with the requirements and the criteria of Figure 3.1[☞94] as summary. Design decisions based on these comparisons are done in the design of the new approach of this thesis in Chapter 5[☞163]. The requirements identified during this Chapter 3[☞93] are explicitly established in the following Chapter 4[☞153].

Initially, some criteria (depicted in Figure 3.1[☞94]) for the functional objectives of approaches for ensuring model consistency (Section 3.1[☞94]) successfully guide the selection of related approaches to investigate and help to roughly evaluate the investigated related approaches. In particular, the levels of heterogeneity help to focus on approaches targeting *semantic* heterogeneity. However, the clear distinction between semantic and structural heterogeneity is difficult or not always possible, since they often depend on each other. Therefore, also structural heterogeneity must be supported, when overcoming semantic heterogeneity as main objective. Important is the finding, that supporting *multi-directionality* of change propagation is an important challenge, which is not decidable in general e. g. for the view-update problem and is not covered by most approaches in data base (Section 3.6.3[☞139]), ontology (Section 3.6.4[☞142]) and enterprise (Section 3.6.5[☞144]) research.

Multi-Directionality of ensuring Consistency

In that research areas, integrated or aggregated data and new views are usually read-only and their changes can not be propagated back into the sources. In particular, data base research and ontology research seem to focus on composing or aggregating data and not on ensuring their consistency (Requirement R 1 (Model Consistency) ☞ 154). For the data base research, data provenance can be seen as first but not sufficient step to update initial data sources after their combination.

Even bidirectionality, the simplest form of multidirectionality, introduces the challenge of having *multiple possible results*, when transforming information in two directions: If a transformation transforms information as bijection between two models, the result is unique. Otherwise, the relationship is non-bijective and there are multiple possible solutions in general. Therefore, someone has to decide about the final solution (or the solution is done in non-deterministic way). This problem can be faced and found from different perspectives:

Non-bijective Bidirectionality leads to multiple possible Fixes for Inconsistencies

- Strategies for fixing inconsistencies in particular projects depend on the current project (Section 1.2.1 ☞ 31).
- BX need to derive two unidirectional transformations from the same specification: As an example, if one direction is explicitly specified, can the inverse direction be automatically executed with a predictable result? If the result is the expected one, then the BX satisfies the principle of least surprise. Again, the surprise is project-specific.
- Does a function have an inverse function *and* is the particular result of the function sufficient for the inverse function to produce the original input of the function? If additionally the original input is still known, this setting refers to lenses.
- The view-update problem is not decidable in general.

Therefore, different possible strategies to decide the final fix for inconsistencies are depicted in Figure 3.6 ☞ 106, which can be used by different stakeholders (Section 2.4 ☞ 79).

The solution space for deciding the final fix for inconsistencies involves the degree of automation and the involved stakeholders: Users of BX expect automated selections to be deterministic (Stevens, 2010). Transferring this result to model consistency, *users* want fully-automated and predictable fixes for inconsistencies. Fixed heuristics hard-coded within approaches are often not usable in practice, e. g. least change provides not always the results desired by users (beside technical issues) and least surprise requires more theoretical investigations, while the surprise depends on the particular project requiring domain-specific information (Cheney, Gibbons et al., 2017) and therefore cannot be solved by *platform specialists*. In the investigated approaches, *methodologists* are rarely supported, but in particular by the three projectional SUM approaches. The clear distinction between users and methodologists is usually not done or not exploited, e. g. persons using BX approaches to specify a concrete BX (methodologists) have different skills and needs than persons using the concrete BX by starting the engine of the BX approach for automated execution (users). In particular, model synchronization-based approaches try to solve the selection challenge on level of platform specialists with heuristics like least change and least surprise. In order to enable project-specific selections (according to project-specific consistency goals), platform specialists should provide means to enable methodologists to specify the selections which are desired by users.

Selecting Fixes: deterministic, Degree of Automation, involved Stakeholders

Another important finding in the BX research during extensions of bidirectional to multidirectional model transformations is, that there are consistency goals targeting three or more models which cannot directly be split into multiple pairs of binary consistency relations between two models (Stevens, 2017; Macedo, Cunha and Pacheco, 2014), with a simple example in Stünkel, König et al. (2021). In the data base domain, it is challenging to realize even cardinality constraints for ternary relationships (Cuadra, Martínez et al.,

binary vs n-ary Consistency Goals

2013). Therefore, realizing n-ary consistency goals is another challenge when fulfilling Requirement R 1 (Model Consistency)☞154.

Fix initial Inconsistencies

Another challenge is to *fix initial inconsistencies* which occurred before applying an approach for ensuring consistency: This challenge is rarely discussed in related approaches, but relevant for all approaches expecting consistency as precondition at some point in time "before their application", in particular for change translation-based approaches. In order to make this challenge explicit, Requirement R 2.3 (Fix existing Models)☞156 is established later.

Representing Changes

Techniques for change propagation require the model changes, usually done by users ($^{\text{User}}\Delta$), to propagate them to related models. Changes in the model are encoded in three different ways by the investigated approaches:

- Changes in the model can be explicitly encoded by using model deltas, as in delta-lenses.
- Changes in the model can be implicitly encoded by the current (and updated) version of the model with two different options to derive the actual model deltas:
  - The current version of the models is compared with its previous version (state-based), if the current model version is given as an additional model ("out-place"). This option introduces accidental i. e. unnecessary ambiguity, since the changes as result of model difference calculation between two model versions are not unique.
  - If the model is updated in-place and explicit links or other meta-data for the previous state of the model are available, they can be exploited to identify created and deleted elements, leading to explicit model deltas. An example are the correspondences in TGGs, which indicate previously matched patterns and are used for realizing incremental TGGs (Section 3.3.1☞108).

Explicit Model Deltas are more expressive than new Model Versions

The model changes are directly required for change translation-based approaches or for incrementality of the other kinds of approaches for change propagation: It is important to use the changes like the $^{\text{User}}\Delta$ directly and not only the updated model, as seen for (delta) lenses, since deletions and (re-)creations tend to loose information, which can be prevented or restored by model differences. Additionally, correspondences as representatives for consistency between source model and target model are reused and updated, too, i. e. the model synchronization does not provide "any new model" which is consistent to the other model, but provides an updated model according to the previous consistency correspondences and to the $^{\text{User}}\Delta$ (Abou-Saleh, Cheney et al., 2018, p. 10). Therefore, model synchronization-based approaches taking the $^{\text{User}}\Delta$ into account tend to behave as change translation-based approaches.

$^{\text{User}}\Delta$ can be amended during Change Propagation

Another generic finding from symmetric multiary delta-lenses is, that the $^{\text{User}}\Delta$ can be amended during the change propagation. This fits to findings in Section 2.3☞71 when formalizing consistency as relation, that there are cases, where the currently changed (and inconsistent) model must be fixed in order to fulfill consistency. Therefore, amending the $^{\text{User}}\Delta$ is allowed in consistency rules.

Cross-cutting Techniques

Section 3.2☞99 identifies lots of *cross-cutting techniques* and strategies to realize approaches for ensuring consistency including the distinction between projectional and synthetic settings, explicit links and four different techniques for change propagation. This section uses these techniques for the outline here in order to summarize main characteristics of these techniques in the following paragraphs.

Proof-Theory-based Approaches

Summarizing proof-theory-based approaches, they need bridges between technical spaces for modeling and technical spaces for formal specifications, but can explicitly span the solution space. Proof-theory-based approaches often have a performance in NP, due to

the used solvers. The investigated proof-theory-based approaches usually do not support incrementality, the exception of Almeida da Silva, Mougenot et al. (2010) directly represents model differences instead of model elements in Prolog. This fits to the observation for BX approaches, that only JTL using proof-theory techniques does not support incrementality, while the other three BX approaches support incrementality (Samimi-Dehkordi, Zamani and Kolahdouz-Rahimi, 2016).

Rule-based Approaches

Summarizing rule-based approaches, they are very flexible in adding and removing rules for consistency goals, depending on the current needs of the current project. There are several rule engines which support the incremental evaluation of rules in order to improve performance. Checking consistency by evaluating rules is very similar in the investigated approaches, but often require external support for multi-models. In contrast, the strategies for providing fixes for found inconsistencies vary.

Model Synchronization-based Approaches

Summarizing model synchronization-based approaches, they use model transformations and therefore require similar granularity levels for the information in the involved models. Compared with the ongoing example, representations for classes can be transformed between Java and class diagrams, but transformations between requirements and Java do not work. Even in the case, that some information can be transformed between models, usually there is more information which cannot be transformed. In particular in symmetric cases (Figure 3.5 [☞ 104]), often some information is missing in some of the models. This requires *incremental model transformations* to keep such information unchanged and to keep transformable information consistent. The surveys of Kusel, Etzlstorfer et al. (2013) and Kahani, Bagherzadeh et al. (2019) report, that all investigated incremental model transformation approaches require correspondences between the transformed model elements[10]. The case study of Buchmann and Westfechtel (2016, p. 179f) for incremental consistency between class diagrams and Java source code with TGGs shows, that rules in TGGs have limited flexibility due to fixed elements and changes in the patterns, which results in the combinatorial explosion of rules to transform associations for example. BX approaches have to deal with the trade-off between formal guarantees and expressiveness, leading to more practical approaches with imperative parts (Bank, Buchmann and Westfechtel, 2021). The model synchronization-based approaches which support the symmetric case are required for synthetic settings, for projectional settings, the asymmetric case is sufficient, while the bijective case rarely occurs in practice. Composability of model transformations is an important design goal, in particular for BX (Stevens, 2010) and for lenses (Diskin, König and Lawford, 2018), which is deepened in Section 6.4.1 [☞ 203].

symmetric for synthetic, asymmetric for projectional, bijective rare

Change Translation-based Approaches

Summarizing change translation-based approaches, they are incremental by design, since they directly react on occurring changes. With this design, they immediately translate changes and can not control the point in time of change propagation (as the other three kinds of approaches can do), otherwise, generated follow-up changes and stored changes must be merged and might be in conflict. Another limitation is, that all models must initially be consistent to each other, before changes occur and can be translated, leading to Requirement R 2.3 (Fix existing Models) [☞ 156].

comparing Change Propagation Techniques

Comparing the four groups of approaches for change propagation with each other is done along Table 3.2 [☞ 150]: Interesting is the comparison regarding checking consistency (first row) and fixing inconsistencies (second row): These two steps are explicitly separated by rule-based approaches, while model synchronization-based and change translation-based approaches focus on fixing inconsistencies without explicitly searching for inconsistencies before. The classification regarding additional technical spaces (third row), incrementality with a performance depending on the amount of model changes (fourth row) and the possibility of delayed inconsistency fixing (fifth row) are already discussed during the sum-

[10] The only exception is Echo (Macedo and Cunha, 2013), since it realizes QVT-R model transformations not directly, but with a proof-theory-based technique (details above).

Criterion	Proof-Theory	Rules	Model Synchro.	Change Translation
Checking	✓	✓	(implicitly)	–
Fixing	✓	✓	✓	✓
Within Space	–	✓	✓	✓
Incrementality	–	✓	✓	✓
Delayed Fixes	✓	✓	✓	–

Table 3.2: Comparing Techniques for Change Propagation

maries of the respective outlier approaches. Beyond this comparison, proof-theory-based approaches seem to be used more for behavioral aspects and the other approaches for static aspects, e. g. Van Der Straeten, Mens et al. (2003) use description logics for dynamic aspects of UML and Mens and Van Der Straeten (2007) as rule-based approach for static aspects of UML, as mentioned by the research group of Tom Mens (Mens, Van Der Straeten and D'Hondt, 2006, p. 212).

clear Separation of Techniques is blurring

Looking at concrete approaches within the four groups, the clear separation of the four techniques (proof-theory, rules, model synchronization, change translation) is blurring: In-place model transformations for checking consistency are similar to rule-based approaches evaluating constraints. Bidirectional and incremental model transformations behave similarly like change translation-based approaches with similar performance depending on the amount of model changes. This results in approaches for change-driven model transformations with model changes as first-class citizens (Bergmann, Ráth et al., 2012). Accordingly, the focus of BX research is moving from model transformations to maintain consistency (Stevens, 2018), e. g. with principles of least change and least surprise. Proof-theory is used by other approaches like model transformations for BX.

Synthetic Approaches

Summarizing synthetic approaches, they require lots of direct relations between pairs of models, in form of rules or model transformations, sometimes with additional explicit links between them. This graph is not always completely mashed, but in the magnitude of $O(n^2)$ in general with n as the number of models to keep consistent to each other. A challenge are n-ary consistency relations due to their missing "binarization" in general: Some model transformation approaches allow to target multiple models at the same time (including TGGs). As an alternative, multiple pairs of model transformations can be used, if the corresponding n-ary consistency relation can be split into binary consistency relations. Another challenge is the *execution order* in networks of BX, since it is not obvious and must be explicitly specified in general (Stevens, 2017). But also in rule-based approaches, the order of applying rules is important, since fixes by the first rule can influence possible fixes of the following rules or introduce new inconsistencies requiring additional rules for their fixes. Interestingly, approaches for synthetic settings support either Requirement R 1 (Model Consistency) ☞154 or Requirement R 3 (Define new View(point)s) ☞156, i. e. the according Section 3.3.1 ☞108 and Section 3.3.2 ☞119 have no overlap of approaches[11]. Therefore, one approach for keeping source models consistent should be combined with one approach for deriving new view(point)s.

Projectional Approaches

SU(M)M idea

Summarizing projectional approaches, they depend on the SUM idea (Section 3.4 ☞120), having one SUM conforming to one SUMM containing all information respectively concepts about the system under development. All views are projections from the SUM. Therefore, for model synchronization-based approaches, it is sufficient to support the asymmetric case, while the symmetric case is required for synthetic settings. There are specific approaches with fixed viewpoints and consistency goals and there are generic approaches enabling arbi-

[11]This counts for specifically designed approaches for synthetic settings, generic model transformation approaches can be used for both use cases.

trary viewpoints and consistency goals. The specific approaches show, that the projectional idea is working in practice in different application domains.

generic projectional Approaches

There are three generic approaches (Section 3.5☞121) which are explicitly designed to realize the projectional SUM idea: Vitruvius and RSUM are very similar, since they both are bottom-up, use ModelJoin for deriving new view(point)s and use change translation-based synthetic techniques internally for keeping the models consistent to each other, without an explicit and redundancy-free SUM. Since these two approaches are change translation-based, their reuse of initially inconsistent models is restricted (Requirement R 2 (Reuse existing Artifacts)☞155). The CPRs of Vitruvius enforce only binary consistency goals, not n-ary ones, with ongoing research to overcome this challenge (Klare, 2018). The top-down approach OSM in contrast is missing a strategy to develop a clean and redundancy-free SU(M)M from existing source (meta)models (Requirement R 2 (Reuse existing Artifacts)☞155). Section 3.5.4☞131 investigates approaches for this purpose with the result, that transformations are required to eliminate redundancies, depending on the project-specific semantics of the source models. But these transformation approaches are unidirectional and do not update the source models after their combination into the SUM anymore, therefore hurting Requirement R 2 (Reuse existing Artifacts)☞155. The concepts of all three SUM approaches for deriving new view(point)s are restricted in order to enable editability by change propagation, e. g. ModelJoin used by Vitruvius and RSUM does not support new classes in new viewpoints (Requirement R 3 (Define new View(point)s)☞156). Therefore, all three SUM approaches do not completely fulfill all requirements.

more Projections

Language workbenches supporting the development of multiple representations for the same model follow also the projectional SUM idea, independently, if parser-based editing or projectional editing is used for realizing concrete syntaxes. Since they are top-down, language workbenches do not support the reuse of existing models conforming to arbitrary metamodels (Requirement R 2 (Reuse existing Artifacts)☞155). The organization of viewpoints along orthogonal dimensions by OSM is independent i. e. orthogonal to the mechanisms for ensuring consistency, therefore, this approach for viewpoint organization can be reused by other approaches ensuring consistency between different views.

Formalizations: Lenses, Round-trip

Summarizing formalizations with lenses and round-trip engineering, they formalize conditions for approaches ensuring consistency, in particular for model synchronization-based approaches. They do not propose concrete implementations, but point to conceptual challenges and motivate some ideas for implementation, including . . .

- the explicit specification and use of explicit links for horizontal alignment between different models and of model differences for vertical alignment between different versions of the same model,
- the use of model deltas instead of updated models to reduce ambiguities and
- the possibility for amending the ${}^{User}\Delta$ during the change propagation.

Consistency in different Research and Application Areas

Summarizing different research and application areas, in the end, ensuring consistency as the main challenge is the same in different research and application areas like modeling (Section 3.3☞108, Section 3.4☞120, Section 3.5☞121), modeling languages (Section 3.6.1☞136, Section 3.6.2☞137), data bases (Section 3.6.3☞139), ontologies (Section 3.6.4☞142) and enterprises (Section 3.6.5☞144). This is discussed along two aspects: In the *first* aspect, similar matching techniques can be applied in principle both for schemata of data bases and for schemata of ontologies, as proposed by the classification and survey of Shvaiko (2005).

Matching Techniques for Schemata of Data Bases, Ontologies and Models

Since these schemata describe the concepts and structures of data, which counts also for metamodels, these matching techniques could be transferred to metamodels, too. The other way around, techniques for (meta)model matching (Somogyi and Asztalos, 2020) could be

transferred to schemata of data bases and ontologies. In the *second* aspect, fixing inconsistencies in the modeling domain and the view-update problem in data base research look quite differently first, but can be seen as two sides of the same problem: Both mainly target the model (or instance) level, not the metamodel (or schema) level, but are solved on the metamodel level. Fixing inconsistencies between interrelated models (usually used as views) sounds synthetic, but can be solved also in projectional way, while the view-update problem arises in projectional settings between views and their SUM. Since data base views can be described by "*in general unidirectional, partial and non-injective functions; very similar to model transformations*" (Hettel, Lawley and Raymond, 2008), they are similar to model synchronization-based approaches using model transformations to ensure consistency. Therefore, the main challenge of fixing inconsistencies in modeling and of the view-update problem in data base research is the same.

Fix Inconsistencies vs View-update Problem

Summary

Summarizing this summary, there are lots of approaches for ensuring inter-model consistency in various research areas with various realization techniques and strategies. During their exemplary investigation, the challenges for ensuring consistency are concretized and extended, which results in concretized requirements, which are finally motivated and collected in the following Chapter 4☞153. Additionally, promising realization techniques are detected, which could be reused for new approaches. As main classification, approaches for ensuring inter-model consistency can be distinguished into synthetic and projectional approaches: *Synthetic approaches* fulfill the main requirements in general, even with some restrictions in details like n-ary consistency goals in model synchronization-based approaches and execution orders, but require a square amount of explicit links, rules or model transformations between the involved models. Generic *projectional approaches* can be distinguished into bottom-up and top-down approaches: The investigated bottom-up approaches reuse existing (meta)models, but have restrictions with initially inconsistent models and use synthetic techniques internally. The investigated top-down approaches lack strategies to reuse existing models and their metamodels. Summarizing, there is no approach which fulfills all requirements for ensuring inter-model consistency with satisfactory degree. Therefore, a new approach for ensuring inter-model consistency is required and designed in Part III☞163 of this thesis. Based on the findings and comparisons of this Chapter 3☞93, design decisions for that approach are made in Chapter 5☞163.

Chapter 4

Requirements

Objective of this section is to collect and document the requirements for the design of MoConseMI, the new approach which is proposed in this thesis. The first high-level requirements are directly derived from the objectives of this thesis, described in Section 1.3.3[☞46]:

Objective: collect Requirements

High-level Requirements

- **R 1** Changes in one model have to be propagated into all related models. (**Model Consistency**)
- **R 2** The approach must allow to reuse existing artifacts. (**Reuse existing Artifacts**)
- **R 3** The approach must allow to define new view(point)s. (**Define new View(point)s**)

During the analyses of foundations for inter-model consistency (Chapter 2[☞51]) and of related approaches (Chapter 3[☞93]), these requirements were concretized by refining sub-requirements and detecting additional requirements. These new requirements were already shortly mentioned at their first occurrence, but officially introduced in this Chapter 4 with more explanations and with their origins. Therefore, this section can be seen as a summary of Part I[☞25] and Part II[☞51].

The identified requirements for MoConseMI are grouped into functional requirements (Section 4.1[☞154]), which focus on the conceptual design of the approach, and into technical requirements (Section 4.2[☞157]), which focus on the technical implementation of the approach. The quality of the desired approach is not described by requirements, but is evaluated and discussed later in the evaluation in Part V[☞467]. Each requirement is presented with an description and its origin. Possible origins of requirements are motivation and challenges of ensuring consistency, the objective of this thesis, foundations or related approaches including their implementations.

Grouping and Content of Requirements

The first three high-level requirements (see above) are motivated by the challenges of ensuring inter-model consistency (Section 1.2[☞31]). They reflect all objectives of this thesis (Section 1.3.1[☞42]) and thus form the complete set of functional requirements, which only needs refinement by sub-requirements. Since these requirements focus on objectives and not on realization strategies, they are valid also for other approaches and help to evaluate existing approaches for ensuring inter-model consistency.

Validity

4.1 Functional Requirements

This section collects functional requirements, which must be fulfilled by MoConseMI for ensuring inter-model consistency. The three high-level requirements are directly derived from the objectives of this thesis (Section 1.3.1☞42) and are already depicted in Section 1.3.3☞46. As summary of the motivated challenges of ensuring inter-model consistency (Section 1.2☞31), they are complete, but are concretized according to the findings in Chapter 2☞51 and Chapter 3☞93.

Ensure Inter-Model Consistency as main requirement

Since the use of multiple views can lead to inconsistencies between their respective models (Section 1.1☞26), the main challenge is to ensure consistency between different models by changing related models according to changes within one model (Section 1.2.1☞31). To overcome this challenge is the main motivation for this thesis and leads to the following Requirement R 1:

Requirement R 1: Model Consistency

Changes in one model have to be propagated into all related models.

Automation to relieve Users

Since manual consistency preservation is error-prone, requires high effort and is an repetitive task, change propagation requires support for *automation*. Such an automation reduces the knowledge and care of users using views, since they can concentrate on their particular tasks within their views, but impacts caused by their changes can be automatically propagated into all related views. Ensuring consistency between different models is aimed, independently from the models' roles. In particular, this includes models of existing data sources (Requirement R 2☞155) as well as models of newly derived views (Requirement R 3☞156). Change propagation is the behavior desired by users of models, as deepened in Section 2.3☞71. But change propagation is no concrete realization strategy for ensuring consistency, since Section 3.2☞99 identifies several classes of concrete realization strategies for change propagation, including model synchronization and change translation.

Models conforming to arbitrary Metamodels

Since the models to be kept consistent not only have different roles but also represent different information from various domains including different views tailored to particular concerns of different stakeholders, models contain different kinds of information and are structured differently. Nevertheless, consistency between such heterogeneous models must be ensured. Since metamodels are the means to determine the general concepts of models (Section 2.2.2☞60), arbitrary metamodels must be supported. The amount of projectional approaches supporting the consistency between models conforming to *fixed* metamodels (Section 3.5☞121) shows, that there is the need for generic approaches supporting models conforming to arbitrary metamodels. This leads to Requirement R 1.1:

Requirement R 1.1: Generic Metamodels

The approach must support arbitrary metamodels.

Support any Models, independent from their Metamodels and concrete Renderings

Since metamodels guide the construction of models and determine valid models, the support for arbitrary metamodels enables the support for arbitrary conforming models. The information encoded by these models might have different concrete renderings including different formats, DSLs and so on, as the adapters for different formats demonstrate in Section 8.4☞271. Again, the approach should be independent from concrete renderings of models.

Consistency Goals depend on project-specific Semantics

Section 1.2.1☞31 analyzes, that the desired consistency depends on the particular development project, since consistency depends on the particular semantics of the involved models, as understood by the particular stakeholders. Part 4☞35 of the ongoing example shows different possible consistency goals for the same (meta)models, which both might

be used in different projects. Therefore, approaches with fixed consistency goals are not sufficient, leading to Requirement R 1.2:

Requirement R 1.2: Generic Consistency Goals

The approach must support arbitrary consistency goals concretized by consistency rules.

In particular, this requirement also includes the support of n-ary consistency relations (Section 3.7[☞146]) between $n > 2$ models as consistency goals. This is in line with Stünkel, König et al. (2021), who establish n-ary consistency goals as an important requirement for inter-model consistency. Since consistency goals should be specifiable in a generic way, even their consistency rules must not be fixed. Requirement R 1.2 depends on Requirement R 1.1 (Generic Metamodels)[☞154], since the consistency goals are formulated for elements of metamodels (Section 2.3[☞71]): Related approaches supporting only a fixed set of metamodels, usually support also only fixed consistency goals (Section 3.5[☞121]). Requirement R 1.2 ensures, that even for the same set of metamodels, different consistency goals can be specified in different projects. Section 14.2.2[☞489] discusses some preconditions for consistency goals and their consistency rules, which must hold for them in order to be realizable, i. e. "arbitrary" does not mean "any" consistency goals, but "arbitrary" within those preconditions. Aim of this requirement is to ensure, that the consistency goals are not pre-defined, but highly configurable by methodologists within those preconditions.

Flexible Consistency Goals and Consistency Rules

The second high-level requirement is already motivated by Section 1.2.2[☞36]: Since there exist lots of standards, tools and environments in already running development projects with lots of already developed artifacts, they must be reused, when introducing an approach for supporting consistency. This leads to Requirement R 2:

Reuse existing environments including developed artifacts

Requirement R 2: Reuse existing Artifacts

The approach must allow to reuse existing artifacts.

Part 5[☞37] of the ongoing example shows some examples for artifacts to reuse. While the focus is to reuse existing artifacts and to keep their models consistent to each other, it is also possible to start without any reused artifacts, as discussed in Section 13.3.2[☞474]. The following sub-requirements concretize the artifacts to reuse, by proposing what to reuse and in which way.

Since artifacts to reuse (according to Requirement R 2) come with metamodels (according to Definition 3[☞36]), they must be reused, too. As already motivated in Section 1.2.2[☞36], the metamodels are given by existing standards, tools, environments and DSLs. This leads to Requirement R 2.1:

Support existing Tools

Requirement R 2.1: Reuse existing Metamodels

The approach must allow to keep existing metamodels as initial viewpoints.

To make the approach interoperable with existing tools, the metamodels given by the tools have to be supported as initial viewpoints. In particular, the concepts of the metamodels of the tools must be addressable by consistency goals for formulating the desired consistency. This is the precondition for reusing data developed with these tools, as described in the following paragraphs.

Enable Consistency Goals for Concepts of interoperable Tools

Since artifacts to reuse (according to Requirement R 2) come not only with metamodels but also with models (according to Definition 3[☞36]), they must be reused, too. As already motivated in Section 1.2.2[☞36], models developed in projects without consistency

Reuse existing Data

management often stem from (legacy) results of previous projects or are reused libraries. This leads to Requirement R 2.2:

Requirement R 2.2: Reuse existing Models

The approach must allow to reuse existing models as initial views.

Import existing Models and keep them consistent

To support existing data, existing data are treated as initial models to reuse. Reusing such models means, that existing models are treated as existing views in form of a data source and are imported by approaches. After their initial reuse, the views must be kept consistent to all other views, according to Requirement R 1 (Model Consistency) ☞154. Reusing models depends on the reuse of their metamodels, therefore Requirement R 2.2 depends on Requirement R 2.1 (Reuse existing Metamodels) ☞155.

Fix initial Inconsistencies in reused Models

Since models to reuse (according to Requirement R 2.2 (Reuse existing Models)) are handled manually or in an unstructured way in projects so far, there is a high probability for inconsistencies in such approaches. When reusing existing models and subsequently ensuring their consistency, it must be ensured, that the models are also initially consistent with other reused models. This leads to Requirement R 2.3:

Requirement R 2.3: Fix existing Models

The approach must allow to fix inconsistencies within reused models.

Challenge depends on particular Approaches, e. g. Change Translation-based

This requirement is important for approaches which assume consistency before users apply manual changes which are automatically complemented with fixes. Since Requirement R 1 (Model Consistency) ☞154 asks (only) for propagating changes, it must be ensured, that the initial models are consistent to each other before. In particular, this counts for change translation-based approaches (Section 3.2 ☞99) like VITRUVIUS (Section 3.5.2 ☞126). Fixing initial models is only necessary when models are reused, therefore Requirement R 2.3 depends on Requirement R 2.2 (Reuse existing Models).

Define new derived Viewpoints

The third high-level requirement is already motivated in Section 1.2.3 ☞39: In order to support additional stakeholders with tailored views and to realize interoperability with additional tools over time, the information of the other views must be combined, selected and provided as new, derived views according to the stakeholders concerns. Therefore, new viewpoints must be specified, which enable the construction of such derived new views. This leads to Requirement R 3:

Requirement R 3: Define new View(point)s

The approach must allow to define new view(point)s.

Newly derived views do not come with an initial model to reuse as data sources in Requirement R 2.2 (Reuse existing Models), but all information for the new view is derived from already existing, reused views. Therefore, this Requirement R 3 and Requirement R 2 (Reuse existing Artifacts) ☞155 complement each other. Part 6 ☞40 of the ongoing example shows an example for a new, derived view(point). The following sub-requirements concretize the derived view(point)s and are partially proposed also by Jakob, Königs and Schürr (2006, p. 322).

Provide Information which is spread over *multiple* Views

As explicitly motivated in Section 3.3.2 ☞119, new views should contain not only (some selected) information of already existing views, but information spread over *multiple* views. This counts in particular for information which is located in different reused views. This leads to Requirement R 3.1 ☞157:

Requirement R 3.1: New Views reuse whole System Description

New views must be able to reuse all information which represent the whole system under development.

Reusing all information of the description of the particular system under development (complete view, according to Figure 2.1[☞ 52]), involves not only the information of the multiple reused (partial) views, but also dependencies like explicit links between them. This requirement is a challenge in particular for synthetic settings (Section 3.3.2[☞ 119]), since there is no single model containing all information as in projectional settings.

All Information including Inter-View Links

Already shortly motivated in Section 1.2.3[☞ 39], new viewpoints have to reflect additional concerns of additional stakeholders or must fit to a metamodel given by an additional tool. Therefore, arbitrary metamodels for new viewpoints must be definable. Generally, it is not sufficient to provide only a direct subset of the elements which are already existing (Section 3.5.5[☞ 134]). Instead, existing elements could be restructured, including renamings, and additional elements could be added. This leads to Requirement R 3.2:

Restructuring existing Concepts

Requirement R 3.2: New Viewpoints with arbitrary Metamodels

New viewpoints must be able to use arbitrary metamodels.

While this requirement is formulated for metamodels, conforming models for new views have to be derived and restructured from the existing information in a similar way.

Already explicitly motivated in Section 1.2.3[☞ 39], additional stakeholders getting new views might want to influence the development of the current system and therefore need to change their views. These changes have to be propagated into all related existing views resulting in *editable* views (Section 3.3.2[☞ 119]) according to Requirement R 1 (Model Consistency)[☞ 154]. Additionally, Goldschmidt, Becker and Burger (2012) identify editability as a feature of views. This leads to Requirement R 3.3:

Users change derived Views

Requirement R 3.3: Editable new Views

New views must be editable by users.

This requirement directly refers to the *view-update problem*, which is not solvable in general (Section 3.6.3[☞ 139]) and approaches need to provide appropriate strategies to deal with it or to make clear, when information in new views is editable and when information in new views is read-only.

4.2 Technical Requirements

This section collects technical requirements, which must be fulfilled by the approach for ensuring inter-model consistency. These requirements mainly target the *implementation* of the approach, but also the *design* of the approach, since the approach must enable the implementation of the desired requirements. These requirements are derived from findings during the investigation of related approaches in Chapter 3[☞ 93] and their supporting tooling.

Since different approaches for model consistency expect the model to be realized according to different techniques, e. g. the three projectional SUM approaches presented in Section 3.5[☞ 121] support three different technical spaces, the structural heterogeneity of models (Section 3.1[☞ 94]) must be covered. Additionally, there are various technical spaces, as sketched in Section 2.5.1[☞ 84]. In order to support and reuse arbitrary models (Requirement R 2 (Reuse existing Artifacts)[☞ 155]), their technical spaces must be supported, too. This leads to Requirement R 4[☞ 158]:

Different Approaches support different Technical Spaces

Requirement R 4: Technical Spaces

The approach must support views realized in different technical spaces.

Bridges for EMF

Accordingly, the new approach also needs a technical space to technically realize the managed models. As motivated in Section 2.5.2☞86, EMF is chosen as technical space of the new approach. Therefore, bridges between other technical spaces and EMF are required, which are realized as adapters (Section 6.6.5☞226).

Ensuring Consistency independent of Tools

Since this thesis aims to ensure consistency between models, but should not depend on tool integration as precondition (Section 1.3.2☞43), the approach should be realized in a stand-alone and reusable way. In particular, the implementation should not depend on a particular tool or environment, as it is often the case for language workbenches that realize DSLs (Section 3.6.2☞137). This leads to Requirement R 5:

Requirement R 5: Reusable Library

The approach must be realized in form of a reusable library.

Reusable Library for different Applications

The implementation of the approach as reusable library allows to easily apply the approach to different application domains. Since ensuring consistency should be automated according to Requirement R 1 (Model Consistency)☞154, there is no need to force particular GUIs to use the approach.

4.3 Summary

In order to summarize the results from the motivating challenges, the objectives of this thesis, the foundations and investigated related approaches, this sections collects and explicitly documents functional (Section 4.1☞154) and technical (Section 4.2☞157) requirements. These requirements are summarized in the following box:

Collected functional Requirements

- **R 1** Changes in one model have to be propagated into all related models. (**Model Consistency**)
  - **R 1.1** The approach must support arbitrary metamodels. (**Generic Metamodels**)
  - **R 1.2** The approach must support arbitrary consistency goals concretized by consistency rules. (**Generic Consistency Goals**)
- **R 2** The approach must allow to reuse existing artifacts. (**Reuse existing Artifacts**)
  - **R 2.1** The approach must allow to keep existing metamodels as initial viewpoints. (**Reuse existing Metamodels**)
  - **R 2.2** The approach must allow to reuse existing models as initial views. (**Reuse existing Models**)
  - **R 2.3** The approach must allow to fix inconsistencies within reused models. (**Fix existing Models**)
- **R 3** The approach must allow to define new view(point)s. (**Define new View(point)s**)

R 3.1 New views must be able to reuse all information which represent the whole system under development. (**New Views reuse whole System Description**)

R 3.2 New viewpoints must be able to use arbitrary metamodels. (**New Viewpoints with arbitrary Metamodels**)

R 3.3 New views must be editable by users. (**Editable new Views**)

R 4 The approach must support views realized in different technical spaces. (**Technical Spaces**)

R 5 The approach must be realized in form of a reusable library. (**Reusable Library**)

The fulfillment of these requirements is explicitly discussed in Section 13.1 ☞ 467. These requirements form the starting point to design and implement a solution in Part III ☞ 163.

Part III

Approach

This part designs and implements MoConseMI (MOdel CONSistency Ensured by Metamodel Integration) as a new approach for ensuring inter-model consistency. The main design decisions for MoConseMI in order to fulfill the requirements are discussed, before this design is detailed with bidirectional operators which realize small transformations in models and their metamodels as main concept. Concrete operators are developed in form of a collection of reusable operators. In addition, the designed MoConseMI approach is implemented as MoConseMI framework. This framework enables the application and evaluation of MoConseMI in practice in the next Part IV ☞ 283.

Chapter 5

MoConseMI at a glance

Chapter 3 ☞93 investigated related approaches for ensuring inter-model consistency regarding the requirements, and identified restrictions for all investigated approaches. Therefore, Part III develops MoConseMI (MOdel CONSistency Ensured by Metamodel Integration) as a new approach for ensuring inter-model consistency. In particular, the OSM approach raises the challenge to create an optimized SUMM and requires according methods (Atkinson, Stoll et al., 2013). MoConseMI provides such a method.

MoConseMI as new Approach for ensuring Inter-Model Consistency

Objective of this section is to give a rough *overview of the design* of the new approach MoConseMI and its main *design decisions*. Based on the findings from related approaches (Chapter 3 ☞93), Section 5.1 discusses and decides the main design choices in order to fulfill the requirements, identified in Chapter 4 ☞153. Afterwards, an overview of MoConseMI with its main concepts is given in Section 5.2 ☞171, together with use cases and the ongoing example. Section 5.3 ☞179 summarizes the results. This section serves as overview of the general design, before it is discussed in detail in Chapter 6 ☞185. This Chapter 5 is inspired by a corresponding section in Jelschen (2024).

Objectives: Overview + Design Decisions

The main design ideas of MoConseMI are published in this publication as well:

Related MoConseMI Publication

Johannes Meier and Andreas Winter (2018a): *Model Consistency ensured by Metamodel Integration.* In: 6th International Workshop on The Globalization of Modeling Languages, co-located with MODELS 2018.

This publication is cited as Meier and Winter (2018a) in this thesis.

5.1 Design Decisions

This section discusses and decides design choices for the conceptual realization of MoConseMI. The first three design choices of this section are already depicted in Figure 3.8 ☞124 and are used in Chapter 3 ☞93 to classify related approaches: The first design choice in Section 5.1.1 ☞164 concerns the *starting point* of the construction process for ensuring inter-model consistency. The next two design choices discuss the outcomes of the construction process for establishing inter-model consistency as *end point*, i. e. they determine the usage of a SUM (Section 3.4 ☞120): Section 5.1.2 ☞165 decides, whether the approach uses a projectional setting with an explicit SUM or a synthetic setting without any SUM. Section 5.1.3 ☞167 bases on the decision of the previous design choice for using an explicit SUM and determines its quality. The last three design choices discuss the *way* from the starting point to the end point: Section 5.1.4 ☞168 decides to use model synchronization techniques for change propagation. Section 5.1.5 ☞169 determines the interplay of stakeholders for

Outline and Motivation of Design Decisions

ensuring consistency with the focus on methodologists. Section 5.1.6[☞170] establishes the reuse of parts of model transformations as means to ease the work of methodologists when they realize consistency by model synchronization. These design decisions are visualized in Figure 5.4[☞180] as summary in Section 5.3[☞179].

With starting point, end point and the way between them, the whole construction process is completely covered by design choices. The first and the third design choices are introduced by Meier, Klare et al. (2019) as design criteria to span the complete solution space for projectional approaches, and are extended by the second design choice to cover synthetic approaches as well. Some design choices depend on previous design decisions, which is deepened in the sections of respective design choices. These design choices in Section 5.1[☞163] are not complete, but intended to guide the main design of MoConseMI. Additional, more fine-grained design choices are discussed in Chapter 6[☞185].

Pattern for Design Decisions

Each design decision is discussed within its own section, starting with a short explanation of the design choice and its motivation and origin. The possible concrete choices are listed afterwards. Then the choice is named and motivated, that is selected as decision by MoConseMI for the respective design decision. Finally, possible impacts of this design decision are listed, which emphasize some details and technical challenges to be overcome in the detailed design in Chapter 6[☞185].

Early Design Decisions

There are also some specifications before these design decisions, which are already discussed in the sections before:

- The objectives and demarcations of Chapter 1[☞25] including its focus on inter-model consistency with the reuse of existing (meta)models and the aimed levels of heterogeneity do not restrict the solution space (and are no design decisions therefore), but determine the (functional) requirements for the desired solution.

Consistency Goals and Consistency Rules

- The formulation of consistency with consistency goals and consistency rules (Figure 2.17[☞73]) is early design, but the formulations are independent from concrete approaches. Therefore, they can be seen as a kind of requirements specification for consistency.

different Stakeholders

- The proposed use cases (Figure 2.20[☞79]) and their stakeholder groups (Section 2.4[☞79]) are early design decisions, but are already introduced in order to compare them with related approaches in Chapter 3[☞93]. Since their degree of involvement into managing consistency is a design choice, this design choice is discussed in Section 5.1.5[☞169].

EMF as Technical Space

- Using EMF as technical space (Section 2.5.2[☞86]) might be seen as early technical design decision. But the supported concepts of ECore (Figure 2.21[☞88]) are only mentioned as look-ahead, while their selection is finally decided in Section 6.6.2[☞222]. Since technical spaces realize models technically, the conceptual design is not influenced by this early design decision for EMF as technical space used within MoConseMI.

Summarizing, the only relevant conceptual design decision, which is already decided, is the design of stakeholders and is deepened in Section 5.1.5[☞169]. The following sections discuss new design choices and motive their design decisions.

5.1.1 Bottom-Up: Existing Artifacts as Starting Point

Design Choice: Top-down vs Bottom-up

This design choice targets the question, what the *starting point* of the construction process for establishing inter-model consistency is. This design choice fits to the design criterion C1 in Meier, Klare et al. (2019) for projectional approaches, which is generalized here to cover both synthetic and projectional settings, and is already established as conceptual design choice in Figure 3.8[☞124]. Possible choices are the following ones:

Bottom-up approaches use the existing artifacts as starting point and build the synchronization on top of them.

Top-down approaches establish the means for synchronization first in an ideal way and tries to make existing artifacts interoperable afterwards.

MoConseMI decides to be a *bottom-up* approach, since the reuse and fix of existing models and their metamodels is easier in bottom-up approaches, since the (meta)models to reuse form the starting point. With this design decision, Requirement R 2 (Reuse existing Artifacts) ☞155 can be fulfilled easier. In particular, OSM as a top-down approach does not come with build-in strategies to reuse existing (meta)models (Section 3.5.1 ☞124). This example fits to observations of Moreno and Vallecillo (2004), who propose bottom-up approaches for reusing existing models. Kurpjuweit and Winter (2007) also propose a bottom-up procedure, which identifies viewpoints first and integrates them into a SUMM afterwards, but leaves the concrete integration strategy open.

Design Decision: Bottom-up for reusing (Meta)Models

5.1.2 Projectional with an explicit SUM as End Point

This design choice targets the question, whether the approach is projectional using an explicit SUM or synthetic without any SUM. This design choice is already established as conceptual design choice in Figure 3.8 ☞124. Possible choices are the following ones:

Design Choice: Synthetic without SUM vs Projectional with explicit SUM

Synthetic without SUM Synthetic approaches use only the existing models and their metamodels and propagate changes directly between pairs of these models, without any SUM.

Projectional with explicit SUM Projectional approaches establish an explicit SUM according to Section 3.4 ☞120, which is used to propagate changes between the existing models and the SUM in both directions, but not between existing models directly. The quality of such an explicit SUM is covered by the next design choice.

MoConseMI decides to be a *projectional approach with an explicit SUM*, mainly since the necessary number of relationships between n models is in the order of $O(n)$ for projectional approaches, while it is in the order of $O(n^2)$ for synthetic approaches: In particular, when adding a new model, projectional approaches require only one additional relationship (which can be seen as constant effort in terms of complexity), while synthetic approaches require n additional relationships in the worst case (Kurtev, 2008, p. 382). Even in the domain of synthetic approaches, some authors are aware of this problem: Feldmann, Wimmer et al. (2016) report on the square number of relationships between models. Therefore, Broy, Feilkas et al. (2010) mention missing scalability of synthetic approaches, while Jin, Cordy and Dean (2003) call the synthetic integration an "utopia" for creating direct converters between reverse engineering tools. If the relationships are realized with model transformations, there is a high effort for transformations between large sets of metamodels (Baumgart, 2010). Therefore, "*a strategy to reduce this number is essential for a model-synchronization-based inconsistency management approach to become feasible within the context of manufacturing systems design*" (Feldmann, Herzig et al., 2015a, p. 164). Such a strategy is provided by projectional approaches. In practice, the models are usually not completely meshed in synthetic settings, e. g. in the ongoing example in Figure 2.19 ☞78 or for the application in Figure 10.7 ☞379. This is the reason, why also some approaches with a projectional user experience internally use synthetic techniques, like Vitruvius (Section 3.5.2 ☞126). Another strategy to deal with the problem is to use models only in a strict (transformation) order (Shinkawa, 2006). But in general, the number of relationships still remains in the order of $O(n^2)$. Besides the scalability, there are other aspects that

Scalability of Inter-Model Relationships

distinguish synthetic and projectional approaches and they are discussed in the following paragraphs.

unclear Execution Orders in synthetic Approaches

Since the relationships in synthetic settings are usually organized as a non-directed graph, synthetic approaches face problems with an unclear or complicated execution order when maintaining and exploiting these relationships. This concerns in particular networks of BX, that employ dense graphs. Since transformations are transitive in this graph type, in BX networks there are multiple possible transformation ways to update the models. Therefore, the execution paths usually must be explicitly controlled or specified (Stevens, 2017). In contrast, an explicit SUM is the root of a tree with the views as leafs and therefore avoids this problem. This problem occurs also in rule-based approaches with an example of Mens, Van Der Straeten and D'Hondt (2006), where the order of checking rules and resolution rules is unclear and therefore can lead to different results depending on the chosen execution order of rules.

Modularity Trade-offs

An advantage of synthetic approaches is their higher modularity, i. e. it is easier to add another model or to remove an already combined model (Yie, Casallas et al., 2009b). Therefore, Knapp and Mossakowski (2018, p. 48) propose a synthetic approach for UML consistency, since a global realization might be hard for behavioral concepts. A high modularity is a main design goal of Vitruvius in order to increase the reusability of defined relationships between models in other projects (Klare, Kramer et al., 2021). On the other hand, there are n-ary consistency goals which cannot be split into pairs of binary consistency relations, which reduce modularity and require additional concepts in synthetic settings to deal with them. Together with unclear execution orders (see above) and consistency of pairwise relationships, summarized as compatibility, this design choice is a trade-off between modularity and compatibility (Klare, 2018).

single Point-of-Truth

The explicit SUM in projectional approaches can be used as single point-of-truth: Since each view reads the data from the SUM and writes changes back into the SUM, the SUM is a natural mediator for possible conflicts, since views do not directly interact with each other as in synthetic approaches.

discard and recreate Views

Having the SUM with all information about all views allows to discard the views, since they can be recreated as projections from the SUM afterwards. This fits to the SUM vision, where views are projected on-demand from the SUM (Section 3.4☞120).

store additional Information in the SUM

Having an explicit SUM, it can be used to store additional information which is not part of any existing model. Exploiting the SUMM can be used to structure these additional information properly. A typical example for such additional information are traceability links between elements located within different models. This discussion is continued in Section 14.1.2.1☞486. Synthetic approaches need to introduce additional models which store inter-model links, if the initial models should remain unchanged.

Design Decision: Projectional and explicit SUM

Finally, this thesis values the lower complexity of the number of explicit relationships between models and the better support for n-ary consistency goals of projectional approaches as more important than the higher modularity of synthetic approaches. Additionally, projectional approaches come with natural solutions for the single point-of-truth and for storing additional information. Therefore, MoConseMI follows the projectional SUM idea. Already France and Rumpe (2007) argue, that a comprehensive metamodel (here: the SUMM) supports the consistency of interrelated views.

Impact

This design decision has the following *impacts*:

- The explicit SUM contains the whole information of all views, not only depending information of different views, according to the SUM idea as presented in Section 3.4☞120. Together with the following design decision in Section 5.1.3☞167 for a pure explicit SUM, this design provides some additional advantages, as discussed below. Nevertheless, this discussion is taken up in Section 13.3.3.2☞476.
- Since the SUM contains all information of the project as single point-of-truth, the

views can be discarded and recreated from the SUM afterwards, according to the SUM vision of on-demand projecting views from the SUM (Section 3.4[☞120]).

- The SUM is project-specific, since the SUM contains all information of all views within the current project and these views are project-specific, including data sources as “input” (Section 1.2.2[☞36]) and new views as “output” (Section 1.2.3[☞39]). These project-specific SUMs are in contrast to e. g. Pfeiffer and Wasowski (2012) with a generic SUM for all kinds of textual languages (not only for modeling languages).

5.1.3 Adjustable Approach towards an essential SUM

Design Choice: Pragmatic vs Essential vs Adjustable SUM

This design choice depends on the design decision in Section 5.1.2[☞165] for an explicit SUM and targets the question, *how many dependencies* exist between elements within the SUM. This design choice fits to the design criterion C2 “pureness” in Meier, Klare et al. (2019) for projectional SUM approaches and is already established as conceptual design choice in Figure 3.8[☞124]. Possible choices are the following ones:

Pragmatic approaches keep all initial dependencies and resolve none of them, like e. g. the modular SUMs of VITRUVIUS and of RSUM.

Essential (or pure) approaches have no dependencies within the SUM (any more) like OSM.

Adjustable approaches (still) have some internal dependencies, e. g. since they resolved some initial dependencies in order to move from pragmatic approaches towards essential approaches. Therefore, this choice is floating inbetween the two extreme choices “pragmatic” and “essential”.

If an approach is not essential, i. e. it has some internal dependencies, it requires means to manage the dependencies and keep depending information consistent to each other.

Design Decision: Adjustable

MoConseMI decides to be *adjustable*, since it starts according to Section 5.1.1[☞164] with existing models and inherits all their initial dependencies, leading to a pragmatic SUM by default. In order to keep the dependencies consistent, the SUM should contain depending information like redundant elements only once, which allows to propagate changes for depending elements from one model to according elements in the SUM and from them (as single point-of-truth) to the elements in other models. This design follows the ideas for change propagation of the OSM approach with an essential SUM. In contrast to the top-down OSM, MoConseMI is bottom-up and therefore usually does not reach the essential quality for the SUM as in OSM, but an essential SUM is reachable in the long-term. Additionally, methodologists might explicitly decide to keep some dependencies, which are synchronized in a different way or should not be automatically synchronized at all. This discussion is deepened in Section 12.2.3[☞460]. Summarizing, MoConseMI is adjustable by moving from pragmatic to essential by removing dependencies within the SUM in order to keep them consistent in the existing models.

Reuse for new Views

An essential SUM provides advantages when using it as starting point for defining newly derived views: All information about the current project can be reused directly in high quality, in particular without redundancies or other dependencies. Effort spent for transitioning a pragmatic SUM into an essential SUM is easily reused, when deriving new views from the nearly essential SUM. In particular, only one model and not multiple models must be queried without any redundancies, easily fulfilling Requirement R 3.1 (New Views reuse whole System Description)[☞157], compared with MODELJOIN used by VITRUVIUS (Section 3.5.2[☞126]) and RSUM (Section 3.5.3[☞129]). Some more characteristics of the SUM are discussed in Section 13.3.3[☞475].

Impact

This design decision has the following *impacts*:

- Since the SUM usually does not exist at the beginning in bottom-up approaches (Section 5.1.1☞164), MoConseMI must create one explicit SU(M)M at the beginning. This creation must be taken into account by the next design decision in Section 5.1.4 for the way between starting point and end point. Additionally, the initial creation of the first SU(M)M represents an additional use case which is executed once without changes and triggers of users, and is designed in detail in Section 6.5.4☞219.

5.1.4 Model Synchronization for Change Propagation

Design Choice: Proof-Theory vs Rules vs Model Synchronization vs Change Translation

This design choice targets the question, which technique should be used to realize the *change propagation* between the reused views (Section 5.1.1☞164) and the explicit SUM (Section 5.1.2☞165). This design choice is already identified as design choice for the technical realization with the feature "Change Propagation" in Figure 3.2☞100, whose sub-features are taken as possible choices here. Possible choices are the following ones, according to their descriptions in Section 3.2☞99:

Proof-Theory is used to check consistency and finds fixes for inconsistencies on formal descriptions instead of on models.

Rules in form of constraints are executed on models in order to find inconsistencies and fix them with additional strategies.

Model Synchronization is realized with model transformations, which transform parts of source models into parts of target models, leading to consistency between them after transformation.

Change Translation is used to directly transform (i. e. translate) changes within one model into corresponding changes for depending models.

Design Decision: Model Synchronization

MoConseMI decides to be a *model synchronization-based approach* using model transformations between views and the SUM, mainly since the views exist, but not the SUM (Section 5.1.1☞164). Therefore, the SUM must be created first before using it, which can be done easiest with model transformations, leading to a model synchronization-based approach. The same counts for newly derived views, since they must be created before they can be kept consistent (Requirement R 3 (Define new View(point)s)☞156).

Review Change Propagation Techniques

But there are some additional reasons, as investigated and summarized in Table 3.2☞150 in Section 3.7☞146: *Change translation-based approaches* provide only follow-up changes, but no complete models, which are required here to create the initial SUM. Additionally, they expect the reused models to be consistent to each other initially, which is not always true (Requirement R 2.3 (Fix existing Models)☞156). *Proof-theory-based approaches* are not chosen here, since they require formal specifications which are context-specific. Therefore, methodologists have to spend additional effort for each project in order to select or create a formal specification which covers the whole SU(M)M. Additionally, specific bridges between technical spaces for modeling and technical spaces for formal specifications are required. *Rule-based approaches* are sufficiently flexible for project-specific consistency management and fulfill the requirements. But this counts also for *model synchronization-based approaches*. Since they additionally provide natural means to create the missing SUM initially, they are chosen here.

Impact

This design decision has the following *impacts*:

- Since changes within one view must be propagated first into the SUM and then into all other views, the model synchronization between views and SUM must allow model *transformations in both directions*, according to the feature "Multi-Directionality" of Figure 3.1☞94.

- Since the SUM contains all information of the system under development, the views contain only a subset of this information. Therefore, the *asymmetric case is sufficient* for model synchronization here, as in all projectional settings (Section 5.1.2 ☞ 165). Symmetric model synchronization-based approaches can be used, too, but are not necessary.
- While the model transformations of a model synchronization-based approach are able to create the initial SUM (the model), they are usually *not able to create its SUMM* (the metamodel). Since the SUMM is required to create the conforming SUM, MoConseMI must cope with this challenge.

5.1.5 Methodologists decide the final Fix

Since there are multiple possible fixes for an inconsistency (Section 2.3 ☞ 71), this design choice tackles the question, how the final fix is selected by which stakeholders, according to the "Selection" feature in Figure 3.6 ☞ 106. Before the possible choices for this design choice are presented, the stakeholders are taken up in the next paragraph as preparation.

Design Choice: Select the final Fix for an Inconsistency

The proposed use cases (Figure 2.20 ☞ 79) and the groups of stakeholders (Section 2.4 ☞ 79) are early design decisions, but are already introduced by Meier, Klare et al. (2019) in general for projectional SUM approaches. The separation of platform specialists and methodologists is particularly motivated by the finding in Chapter 3 ☞ 93, that on the one hand least surprise is aimed by BX approaches and their platform specialists, but on the other hand least surprise depends on the consistency goals of the current project, that are determined by methodologists. Additionally, these groups of stakeholders exist also for model transformations in general, even if these roles are usually not explicitly mentioned. Adapter providers provide additional adapters to support information realized with additional technical spaces, fulfilling Requirement R 4 (Technical Spaces) ☞ 158. This design of stakeholders was already decided in Section 2.4 ☞ 79, since these stakeholders are required for analyzing related work in Chapter 3 ☞ 93, leading to the feature "Stakeholders (who decide)" in Figure 3.1 ☞ 94.

Stakeholders who decide: User vs Methodologist vs Platform Specialist

Therefore, for each of these three stakeholders, their involvement into the selection of the final fix can be decided. Possible choices are the following ones, according to the feature "Selection" of Figure 3.6 ☞ 106:

Stakeholder × Selection

Interactive or manual selection is explicitly and manually done by stakeholders for each occurred inconsistency.

Deterministic and automated selection is automatically done by an algorithm without directly involved stakeholders, if the finally selected fix is predictable.

Non-deterministic and automated selection is automatically done by an algorithm without directly involved stakeholders, if the finally selected fix is not always predictable.

MoConseMI decides the following: MoConseMI aims to support *users* using views with *deterministic and automated* fixing of inconsistencies, depending on the current project. *Methodologists* apply MoConseMI for this aim and configure the desired consistency goals and consistency rules once and manually, i. e. *interactive* with means which are designed by platform specialists and provided by MoConseMI. Additionally, *platform specialists* realize MoConseMI in a way, that the configured consistency goals and consistency rules can be executed in a *deterministic and automated* way. With this design, the selection of the final fix for an inconsistency is deterministic and automated for users by automations provided by platform specialists, but interactive for methodologists, as they manually configure the desired consistency goals and consistency rules. Additionally, this design fulfills

Design Decision: Interplay of Stakeholders

the Requirement R 1.2 (Generic Consistency Goals)☞155 for project-specific consistency goals and consistency rules, since they are manually realized by methodologists. Since the final fix is selected in this way, it is not required to model further fixes and to order or filter them, which counts for all stakeholders. Consequently, the features "Uncertainty Modeling for representing Fixes" and "Order / Filtering" of Figure 3.6☞106 are not used. This design decision is in line with Kramer (2017, pp. 89–94), who proposes fully-automated repairs, while in cases for necessary decisions, not users should be asked, but decisions should be configured by methodologists before-hand.

Impact

This design decision has the following *impacts*:

- Since the users should be provided with automated fixing of inconsistencies, MoConseMI must provide means to execute the configurations of methodologists automatically.
- The automations must be *deterministic*, since determinism is expected by users (Stevens, 2010): Therefore, consistency goals and their consistency rules must specify predictable fixes for inconsistencies (also discussed in Section 14.2.2☞489), which are realized by methodologists. Platform specialists must ensure, that the means to configure consistency and their automation with MoConseMI are deterministic, too.

5.1.6 Reuse Parts of Model Transformations

Aim: Ease the Work of Methodologists

This design decision aims to *ease the work of methodologists*, since they *manually* realize the consistency goals and consistency rules *for each project* (Section 5.1.5☞169). Methodologists use means, which are developed by platform specialists manually, but only *once* for developing MoConseMI. Users of a project are supported with *automated* fixes of inconsistencies, provided by the work of methodologists. Not only the manual and recurrent manner of the methodologists' work requires support but also the mostly very complex project-specific consistency. An example for this is the finding (Section 5.1.4☞168), that techniques for change propagation must support both directions and must also provide the required metamodels for the SUMM and new viewpoints.

Reuse Parts of Model Transformations

In order to facilitate the methodologists' work, recurring work can be eased by reusing techniques to realize consistency goals and consistency rules. The applicability of reusing techniques, in turn, depends on the degree of modularization of the model transformations, since in MoConseMI change propagation is realized by model synchronization (Section 5.1.4☞168), which itself is handled by model transformations. Therefore, the main idea is the reuse of parts of model transformations. This approach is also found in related approaches for model transformations by supporting their composability like lenses and BX (Section 3.7☞146).

Design Choice: Singe Transformation vs Transformation with Parts

The structure of model transformations determines the extent of their reusability. Possible choices are the following ones:

Single Transformation Usually, model transformations are written as *single, compact definition* containing lots of meshed model transformation rules (Section 2.2.3☞67) in order to fulfill the desired transformation task.

Transformation with Parts Alternatively, model transformations can be *split into parts*. Each part fulfills a sub-task of the whole transformation task and contains a small(er) number of model transformation rules. Depending on the design of these parts, they cloud be reused for recurring (partial) tasks. Terms describing parts (most generic term) of model transformations include, e.g., operators (chosen in Section 6.1☞185) and patterns. "Patterns reuse" is also motivated by Del Fabro and Jouault (2005).

The modeling community highly requests reuse of model transformations (Bruel, Combemale et al., 2020).

Design Decision: Reuse Transformation Parts for Methodologists

MoConseMI decides to use *transformations with parts* in order to increase the reuse of such parts. Predefined parts of model transformations should be provided by MoConseMI in order to support methodologists and to ease their work, as motivated above.

Impact

This design decision has the following *impacts*:

- Realizing this design decision is challenging, since reuse and modularization are still a challenge in model transformation approaches (Götz, Tichy and Groner, 2021, p. 480f). Therefore, Section 6.4.1[☞203] discusses some related approaches for modular model transformations.
- In order to enable reuse of model transformation parts (requiring generic parts in general) for project-specific purposes (requiring specific solutions in general), the parts to reuse should provide means to configure them according to project-specific needs. This configuration fits to the impact of Section 5.1.5[☞169]. The required design for configuration is discussed in Section 6.2[☞192] and Section 6.3[☞198].

This design decision is central for the realization of change propagation between the models to be kept consistent to each other. The details of the design for parts of model transformation, which are called *operators*, are discussed in Section 6.1[☞185].

5.2 Overview of the Approach

Overview along Use Cases

Based on the design decisions in Section 5.1[☞163], this section gives an overview of the overall design of MoConseMI and demonstrates it with the help of the ongoing example. Additionally, this section motivates parts of the design, that are detailed in Chapter 6[☞185]. This section is structured according to the use cases of consistency management. To each use case depicted in Figure 2.20[☞79] a separate section is dedicated (Section 5.2.1, Section 5.2.2[☞173] and Section 5.2.4[☞178]). A fourth use case is added in Section 5.2.3[☞176] that results from to the design decision in Section 5.1.3[☞167].

5.2.1 Specify Consistency

Use Case: Methodologists configure the desired Consistency …

In this use case, the methodologist completes two tasks: First, the methodologist formulates the consistency which is desired by users with consistency goals and consistency rules. Second, the methodologist realizes these consistency goals and consistency rules with means provided by MoConseMI. The first task is an organizational task with collecting, discussing and approving the consistency desired by users. Objective of MoConseMI is to support methodologists during their second task with conceptual and technical means. In particular, methodologists have to decide, how inconsistencies are fixed (Section 5.1.5[☞169]). This work is manual and is done for each project, since the consistency is project-specific.

… with Operators

Since MoConseMI uses model synchronization techniques (Section 5.1.4[☞168]), methodologists realize consistency with model transformations: In order to ease this methodologists' work, MoConseMI provides reusable model transformation parts as *operators* (Section 5.1.6[☞170]), which are composed of chains of operators. These chains of operators connect views with the SUM. The operator chains are executed at runtime in order to propagate changes between views (use case in Section 5.2.2[☞173]) or to create the initial SUM (use case in Section 5.2.3[☞176]). This idea is applied to the ongoing example in the following way:

Ongoing Example, Part 17: **Overview of MoConseMI** ← List →

Figure 5.1 shows the main concepts of MoConseMI: The nodes on the circle are views usable by users, while the content within the circle is hidden for users and contains the internals for change propagation realized by methodologists. The white nodes represent the data sources for Req(uirements), Java and ClassDiagram as starting points, which are integrated into the SU(M)M as end point by applying operators, which are annotated along edges. Operators are also used to derive the new view for Traceability from the SUM. Since the mechanisms for model synchronization are split into parts of model transformations, the nodes *i* represent internal (meta)models as intermediate steps after applying some, but not all operators of a chain between a view and the SUM.

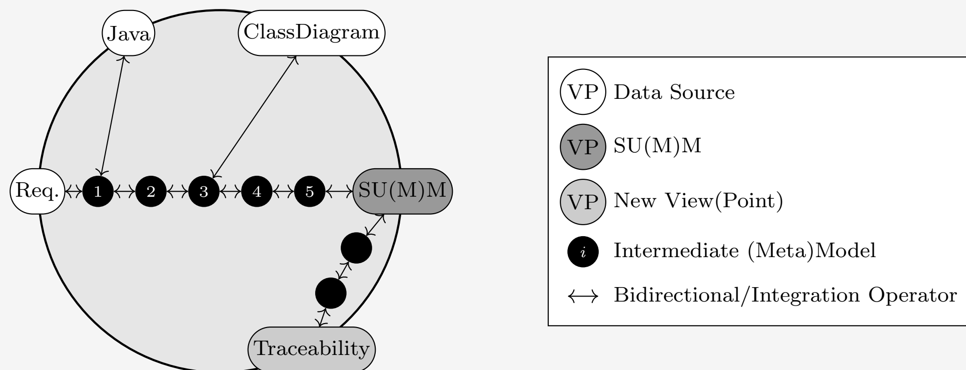


Figure 5.1: SUM approach MoConseMI (taken and slightly adapted from Meier, Werner et al. (2020))

The operators are shorter special model transformations, which are used to create the initial SUM from of the existing reused data sources once and to propagate changes between the views and the SUM afterwards on-demand. The details of the used operators are discussed in Part 21 ☞ 206 of the ongoing example for the integration of data sources into the SU(M)M (which is presented in Part 19 ☞ 176 of the ongoing example) and in Part 22 ☞ 209 of the ongoing example for the definition of a new view(point) from the SU(M)M. Here, it is sufficient to note, that all models (nodes) together with their connecting operators (edges) form a *tree*, with the SU(M)M as root and the views usable by users as leafs.

Design of Operators

With these ideas, operators realize shorter model transformations, which can be combined to chains between the SUM and views: Since the whole transformation between a view and the SUM is split into multiple operators as parts, each operator usually changes only a small portion of the current model, while most parts of the model remain unchanged. In order to improve performance, operators transform models *in-place*, since out-place transformations would copy lots of unchanged elements for each operator again and again, which is deepened in Section 6.1 ☞ 185. Since changes must be propagated from a view to the SUM and from other views via the SUM into the view, operators must allow to *transform models in both directions*, which is deepened in Section 6.1 ☞ 185. Additionally, operators transform also the *metamodels* in order to create the initial SUMM and the metamodels for newly derived views, which is deepened in Section 6.2 ☞ 192. This design of single operators is deepened in Section 6.1 ☞ 185, while their combination into chains is deepened in Section 6.4 ☞ 203.

5.2.2 Fix Inconsistencies automatically

After methodologists used MoConseMI to specify the desired consistency goals and consistency rules at development time (Section 5.1.5☞169), the use case tackled in this section exploits the configurations of methodogists in order to ensure consistency at runtime. As discussed along Figure 2.15☞71, a user takes a view and applies some changes $^{User}\Delta$ to the model of this view only. These changes could introduce inconsistencies with the SUM and other views. In order to fix these inconsistencies, the user changes must be propagated into the SUM and all other views to update them accordingly. This automated change propagation is triggered i. e. started by users after finishing their manual work with the view.

Use Case: Users trigger automated Fixing of Inconsistencies . . .

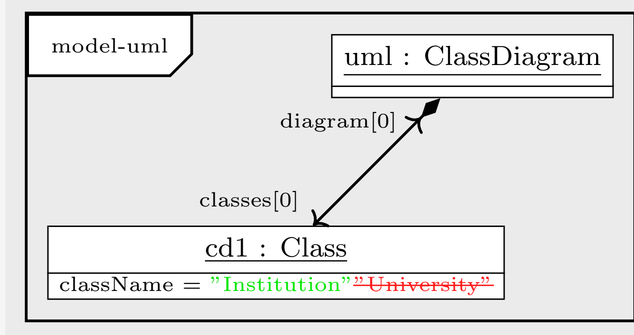
To automatically realize the triggered change propagation, MoConseMI executes the chains of operators and by this means realizes model transformations for model synchronization (Section 5.1.4☞168). Starting with the view which is changed by the user, the operators of its chain are executed one-by-one until the SUM is reached and updated. All other views are updated accordingly by executing the chains between them and the SUM. Finally, the SUM and all views which contain the changed or depending information are updated with model changes $^{E}\Delta$. Note, that the view which is changed by the user might be automatically changed by MoConseMI, too (Section 3.7☞146). Such a scenario is demonstrated for the ongoing example now:

. . . realized by Executing the Chains of Operators

Ongoing Example, Part 18: **Exemplary Inconsistency Fix** ← List →

This box demonstrates an exemplary scenario for a manual user change within one view and resulting changes in other views, which are automatically executed by MoConseMI. This scenario is already introduced in Part 2☞25 of the ongoing example with some text only: The software architect as user takes the current class diagram and renames an existing class. Afterwards, the software architect tells MoConseMI, that the manual changes are complete. Thereupon, MoConseMI automatically renames the class in the Java source code, too, as fix for the inconsistency between class diagram and source code, as specified by Consistency Rule C 2 c☞77. The details of the interactions are described below.

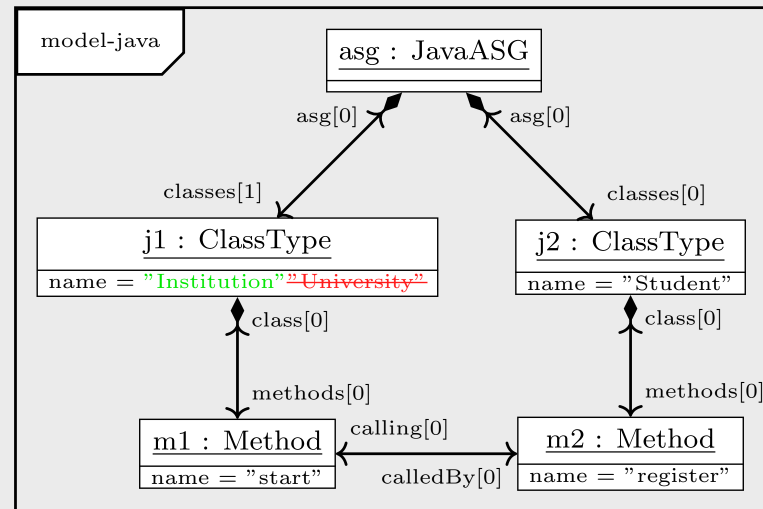
The *user* applies the desired changes to the ClassDiagram view by changing its EMF model. The model changes are represented graphically within the current model:



As result after completing the synchronization, the following changes are expected, according to Consistency Rule C 2 c☞77:

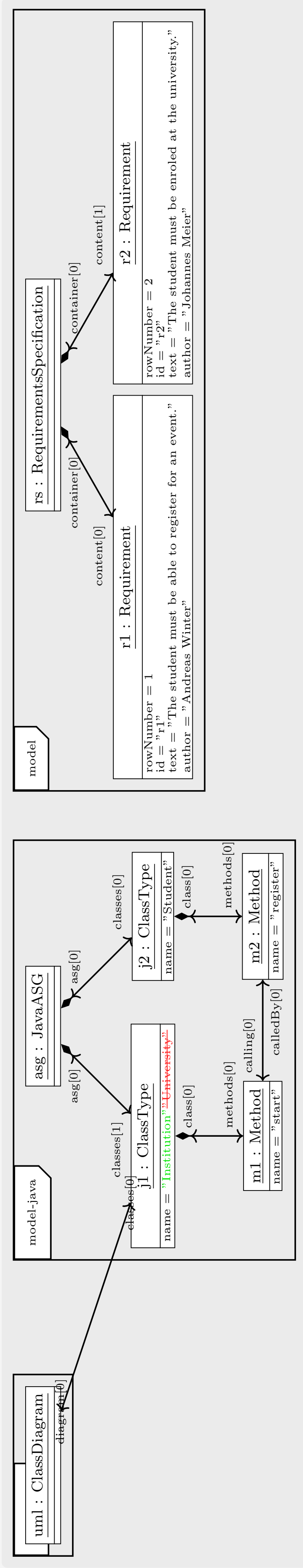
- In ClassDiagram, exactly the changes of the user are expected in the model, as visualized above, and no more changes.
- In Requirements, no changes are expected in the model.
- In Java, the following changes are expected in the model.

  The model changes are represented graphically within the current model:



- In the SUM, as shown in Part 19 ☞176 of the ongoing example, the following changes are expected in the model.

  The model changes are represented graphically within the current model:


model
rs : RequirementsSpecification
container[0]
content[0]
container[0]
content[1]
r1 : Requirement
rowNumber = 1
id = "r1"
text = "The student must be able to register for an event."
author = "Andreas Winter"
r2 : Requirement
rowNumber = 2
id = "r2"
text = "The student must be enroled at the university."
author = "Johannes Meier"
model-java
asg : JavaASG
asg[0]
classes[1]
asg[0]
classes[0]
j1 : ClassType
name = "Institution""University"
j2 : ClassType
name = "Student"
class[0]
methods[0]
class[0]
methods[0]
m1 : Method
name = "start"
m2 : Method
name = "register"
calling[0]
calledBy[0]
uml : ClassDiagram
diagram[0]
classes[0]

- In Traceability, no changes are expected in the model.

This scenario serves as test case for the consistency management in the ongoing example, i. e. as an acceptance test case from the perspective of users for an application of MoConseMI. In particular, the execution as shown in Part 23☞220 of the ongoing example must provide these changes in order to fulfill this test case.

Design of Execution

This section roughly depicted the challenge of automated inconsistency fixes. In Section 6.5☞213, the execution scheme of operators is deepened. One challenge it has to deal with is, that the operators are in-place model transformations and therefore must prevent information loss. Since consistency goals and therefore also fulfilling operators could depend on each other, operators might be executed multiple times, until a fix-point is reached, i. e. no model changes occur anymore. Therefore, model changes must be tracked and made explicit, which is deepened in Section 6.7☞227.

5.2.3 Initialize SU(M)M

Use Case: Create the initial explicit SU(M)M ...

According to Section 5.1.2☞165, consistency is not ensured directly between views, but between each view and the SUM that conforms to a SUMM. Since this requires the SU(M)M to be explicit as a single (meta)model (Section 5.1.2☞165), the SU(M)M has to be *initially created* therefore, since it does not yet exist, when the starting point are the existing data sources, as explained in Section 5.1.1☞164. This initial creation is fundamental and therefore requires a separate use case. It is not depicted in Figure 2.20☞79, since this use case is only required for projectional approaches and the design choice synthetic vs projectional was decided later in (Section 5.1.2☞165). The SUM as end point and its SUMM for the ongoing example are shown in the following box:

Ongoing Example, Part 19: **Initial SUMM and SUM** ← List →

Since MoConseMI follows the projectional idea with an explicit SUM for change propagation (Section 5.1.2☞165), this box shows the SUMM and conforming SUM as complete view for the system under development. SUM and its SUMM are both constructed from the initial models and metamodels of the data sources, as they are presented in Part 9☞64 of the ongoing example. The namespaces respectively packages are taken from the original data sources and indicate roughly the origins of the contained meta-classes.

The SUMM is a single, explicit metamodel and is visualized in Figure 5.2☞177. Compared with the metamodels of the single data sources in Part 9☞64 of the ongoing example, the SUMM contains all their concepts, but in an integrated way according to a more pure SUMM (Section 5.1.3☞167): Classes are represented only once by `ClassType`, since `Class` of ClassDiagram is unified with `ClassType` of Java in order to fulfill Consistency Goal C 2☞77. `Requirements` of Requirements are linked with their fulfilling `Methods` of Java by an association in order to enable traceability according to Consistency Goal C 1☞76.

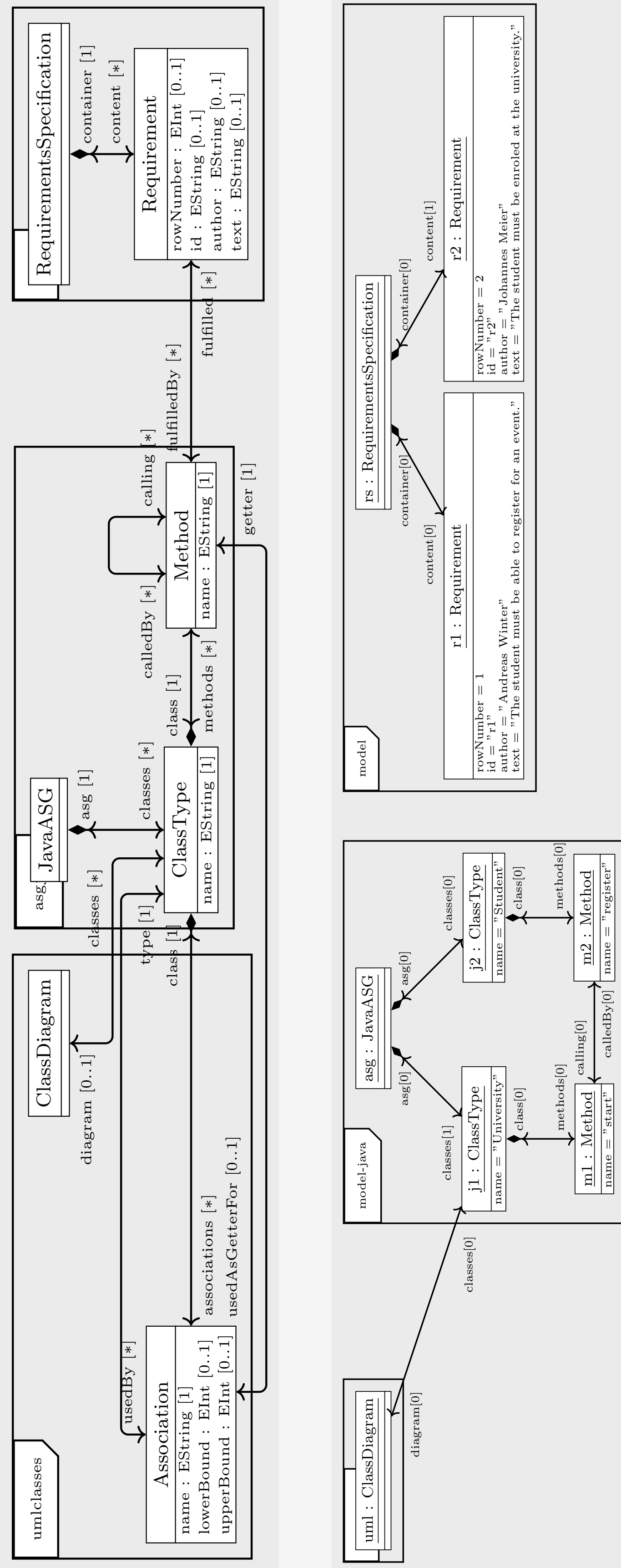


Figure 5.2: Metamodel for the SUMM (left/top) and model for the SUM (right/bottom)

The SUM is visualized in Figure 5.2☞177 as a single, explicit model conforming to the SUMM. Compared with the models of the single data sources in Part 9☞64 of the ongoing example, the SUM contains all their information, but in an integrated way according to a more pure SUMM (Section 5.1.3☞167): Each class of the system under development is represented only once as an instance of `ClassType`. The *initial* SUM does not contain any links between requirements and methods, since such traceability links are not known in the existing data sources, neither in Requirements nor in Java.

... and fix initial Inconsistencies within Data Sources

Additionally, this use case fixes possible inconsistencies within the reused data sources (Section 5.1.1☞164), fulfilling Requirement R 2.3 (Fix existing Models)☞156. Such inconsistencies might occur, since users tried to fix them manually and made some mistakes during that, before MoConseMI was introduced (Section 1.2.2☞36).

This use case is triggered by methodologists and is executed once *after* configuring the desired consistency (Section 5.2.1☞171) and *before* the first change propagation triggered by users (Section 5.2.2☞173). This use case is realized by executing the chains of operators with the existing data sources as starting point and no user changes, but similar to the use case in Section 5.2.2☞173. The details of this design are deepened in Section 6.5.4☞219.

5.2.4 Develop Adapter

In order to fulfill Requirement R 4 (Technical Spaces)☞158 and to reuse and support existing artifacts realized with different technical spaces (Section 2.5☞84), *adapters* are required to bridge technical spaces of artifacts to EMF as the technical space used by MoConseMI. While MoConseMI comes with some predefined adapters (Section 8.4☞271), in this use case, adapter providers develop additional adapters for additional technical spaces. The design of adapters is deepened in Section 6.6.5☞226. Methodologists use provided adapters and apply them in projects. The following box emphasizes the technical spaces required for the ongoing example:

Ongoing Example, Part 20: **Overview of Adapters** ← List →

MoConseMI uses EMF as technical space internally, but needs to support other technical spaces. Therefore, bridges between different technical spaces are required and realized by *adapters*. Figure 5.3 extends Figure 5.1☞172 by showing the artifacts used by users (pentagons) with their technical spaces (black labels). Adapters realize the dotted bidirectional arrows between artifacts and view(point)s.

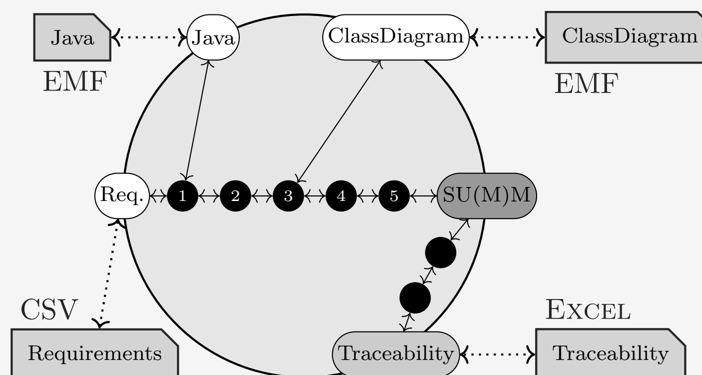


Figure 5.3: MoConseMI with Adapters

The technical spaces of the reused data sources are introduced in Part 5[☞37] of the ongoing example: Requirements are realized within a CSV file, which is deepened in Part 24[☞276] of the ongoing example. Complete Java is realized in the Java technical space, but since Java is a strongly and artificially reduced part of Java in this example, it is directly realized with EMF here. Accordingly, ClassDiagram is directly realized with EMF, since it is a strongly and artificially reduced part of complete UML. According to Part 6[☞40] of the ongoing example, the new view Traceability is realized with Excel in order to support project managers. Summarizing, the ongoing example requires adapters for EMF, CSV and Excel.

5.3 Summary: MoConseMI

This Chapter 5[☞163] provides a rough overview of MoConseMI by deciding high-level design choices and concretizing the use cases for ensuring inter-model consistency according to these design decisions.

The design decisions are depicted in Figure 5.4[☞180]: The first column contains the six design choices, while their possible choices are depicted in the second column. The third column lists the functional requirements, which are fulfilled by the decided design choices, which are marked in gray in the second column. The six design choices are grouped into those concerning either the starting point with reused artifacts or the end point with the explicit SUM with an adjustable number of internal dependencies. The remaining three design choices are related to the way between starting point and end point with model synchronization techniques, configured by methodologists and split into reusable parts.

Design Decisions fulfill Requirements

These design choices are decided in order to fulfill the functional requirements for ensuring inter-model consistency. Therefore, the fulfillment of functional requirements of Section 4.3[☞158] by the design decisions is reviewed here and summarized in the “Req.” column of Figure 5.4[☞180].

- Requirement R 1 (Model Consistency)[☞154] is fulfilled by using model synchronization techniques for change propagation (Section 5.1.4[☞168]). In particular, inconsistencies are fixed automatically and deterministically for users (Section 5.1.5[☞169]).
- Requirement R 1.1 (Generic Metamodels)[☞154] is fulfilled by methodologists, who configure the desired consistency manually and specifically for the individual metamodels of each project (Section 5.1.5[☞169]), which allows to support arbitrary metamodels.
- Requirement R 1.2 (Generic Consistency Goals)[☞155] is fulfilled by methodologists, who configure the desired consistency manually and specifically for each project (Section 5.1.5[☞169]), which allows to support arbitrary consistency goals and consistency rules.
- Requirement R 2 (Reuse existing Artifacts)[☞155] is fulfilled by MoConseMI's bottom-up design with existing artifacts as starting point (Section 5.1.1[☞164]).
- Requirement R 2.1 (Reuse existing Metamodels)[☞155] is fulfilled by MoConseMI's bottom-up design with existing metamodels of the reused artifacts as starting point (Section 5.1.1[☞164]).
- Requirement R 2.2 (Reuse existing Models)[☞156] is fulfilled by the bottom-up design of MoConseMI with existing models of the reused artifacts as starting point (Section 5.1.1[☞164]).

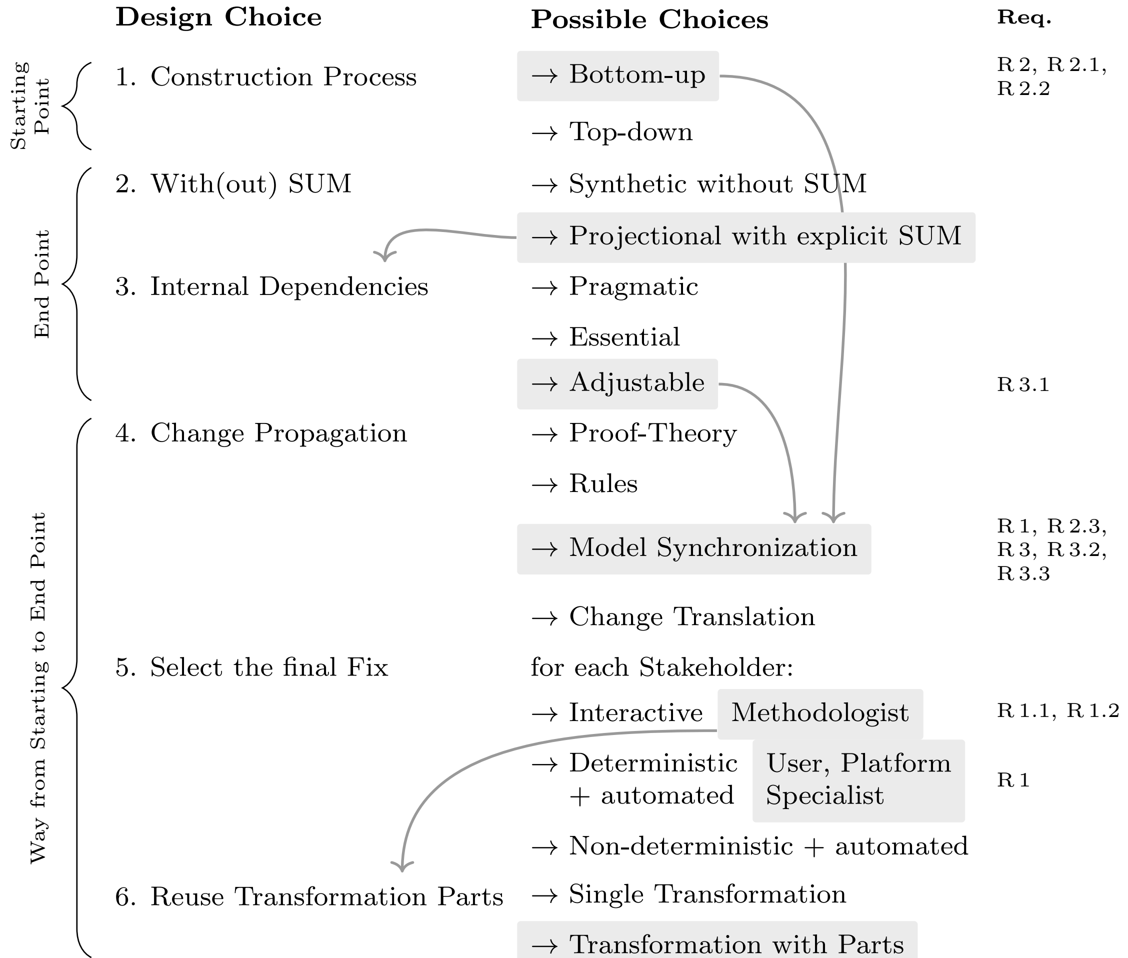


Figure 5.4: Design Choices with possible and decided (in gray) Choices fulfilling Requirements

- Requirement R 2.3 (Fix existing Models) ☞156 is fulfilled by using model transformations for model synchronization (Section 5.1.4 ☞168), which allows to transfer the current information of a model into other models, resulting in consistent models.

- Requirement R 3 (Define new View(point)s) ☞156 is fulfilled by using model transformations for model synchronization (Section 5.1.4 ☞168), which allow to create complete models for new views from the SUM.

- Requirement R 3.1 (New Views reuse whole System Description) ☞157 is fulfilled by using a SUM (Section 5.1.2 ☞165), which contains all information of all data sources and usually contains less internal dependencies (Section 5.1.3 ☞167), which eases the reuse of the information of the SUM for new views.

- Requirement R 3.2 (New Viewpoints with arbitrary Metamodels) ☞157 is fulfilled by using model transformations for model synchronization (Section 5.1.4 ☞168), since they allow to propagate changes between models conforming to different metamodels, enabling metamodels for new viewpoints which are different to the SUMM.

- Requirement R 3.3 (Editable new Views) ☞157 is fulfilled by using model synchronization techniques for change propagation (Section 5.1.4 ☞168), which are designed to support both directions: from views (including new views) to the SUM and from the SUM to views.

This list shows, that all functional requirements are fulfilled with these conceptual design decisions. The fulfillment of the technical requirements is presented in Chapter 6[☞185], while their implementation is described in Chapter 8[☞263].

The design decisions forming MoConseMI's design are not only driven by the defined requirements, but are also guided by features of related approaches (Chapter 3[☞93]). In particular, some design choices of this section are derived from these features as results of investigating related approaches. In the following, these features are compared with the design decisions for MoConseMI in order to show, that the design decisions are senseful according to the investigations of related approaches. For that, the feature models of Chapter 3[☞93] are repeated here and the selected features are marked with light gray color.

The functional objectives for consistency management approaches of Figure 3.1[☞94] are decided for MoConseMI as depicted in Figure 5.5: MoConseMI aims to ensure *inter-model consistency* with the focus to overcome *semantic* heterogeneity. *Structural* heterogeneity is supported together with supporting arbitrary metamodels, since semantic and structure heterogeneity cannot be clearly distinguished in all cases. *Technical* heterogeneity is solved by *adapters* as precondition, but is not the main objective of MoConseMI (Section 3.1[☞94]). In order to propagate changes between all semantically depending models, MoConseMI uses model synchronization techniques to fulfill *multi-directionality* (Section 5.1.4[☞168]). To solve inconsistencies, *methodologists* apply means of MoConseMI to configure the desired consistency manually. These configurations are applied automatically by MoConseMI according to the design of platform specialists in order to support users with automated and deterministic fixes for inconsistencies which are introduced by the users.

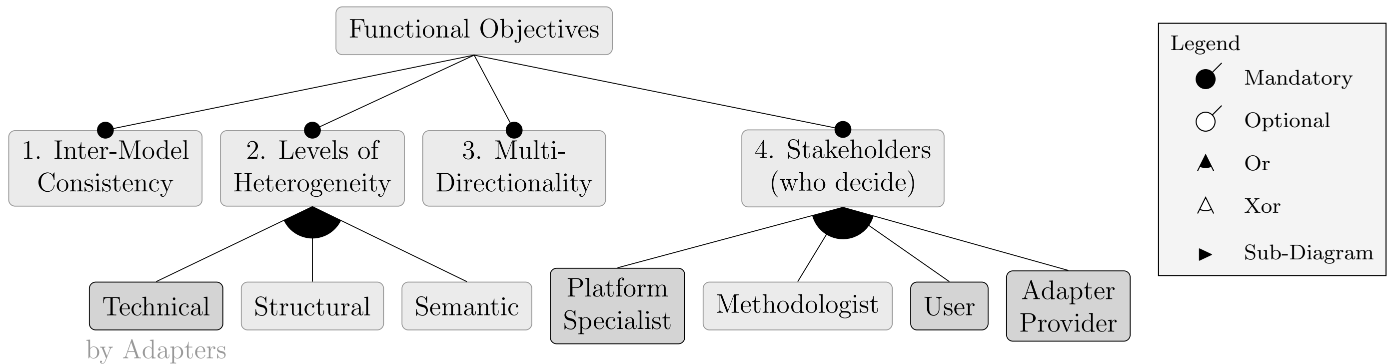


Figure 5.5: Selected Features for classifying functional Objectives of MoConseMI

The design choices of Figure 3.8[☞124] are decided for MoConseMI as depicted in Figure 5.6[☞182]. MoConseMI is *projectional* with an explicit SUM (Section 5.1.2[☞165]), but starts *bottom-up* in order to reuse existing artifacts easier (Section 5.1.1[☞164]). The initial dependencies between the reused artifacts can be resolved within the SUM (Section 5.1.3[☞167]), *adjustable* to the needs of the current project.

The design choices for technical realization of Figure 3.2[☞100] are decided for MoConseMI as depicted in Figure 5.7[☞182]: Since MoConseMI follows the projectional SUM idea (Section 5.1.2[☞165]), it uses an *intermediate model* which contains all information of all partial models, i. e. information which is contained in different models (*overlaps*), information which is contained only in one model (*non-overlaps*), and additional information "between" models like explicit links between them (*inter-correspondences*). These explicit links are on semantic level between the models of interest, while the feature "explicit links" in Figure 5.7[☞182] describes links for technical realization: Semantic and technical links can overlap and should be realized *explicitly* within the SU(M)M. For technical purposes, MoConseMI internally use UUIDs for uniquely identifying elements, which can be seen

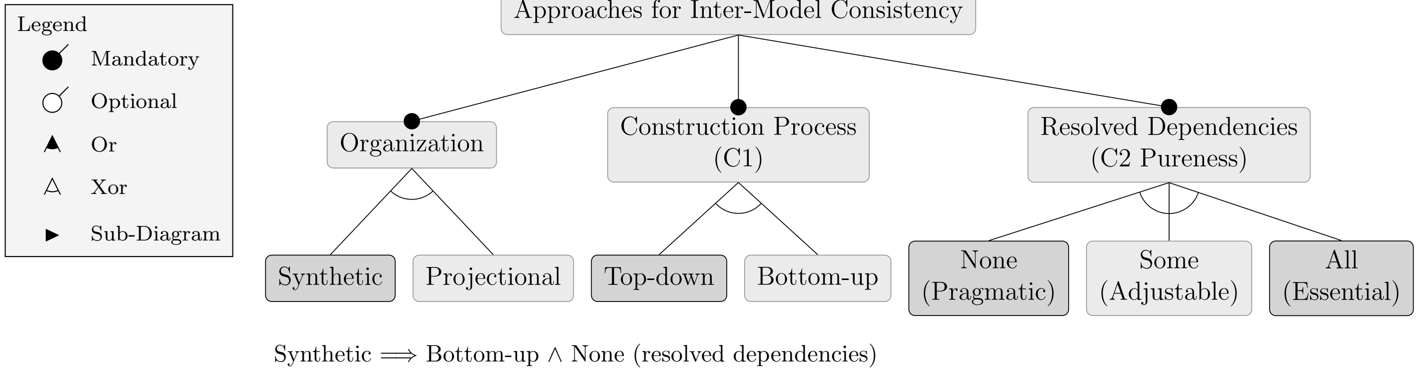


Figure 5.6: Selected Design Choices for conceptual Realization of MoConseMI

as implicit links and is introduced in Section 6.6.4☞225. For change propagation, MoConseMI uses *model synchronization* techniques (Section 5.1.4☞168), which support the asymmetric case for the projectional setting here including the bijective case. The detailed design in Section 6.1☞185 further explains, that the symmetric case is supported. Since the SUM contains all information, no external support for multi-models is required.

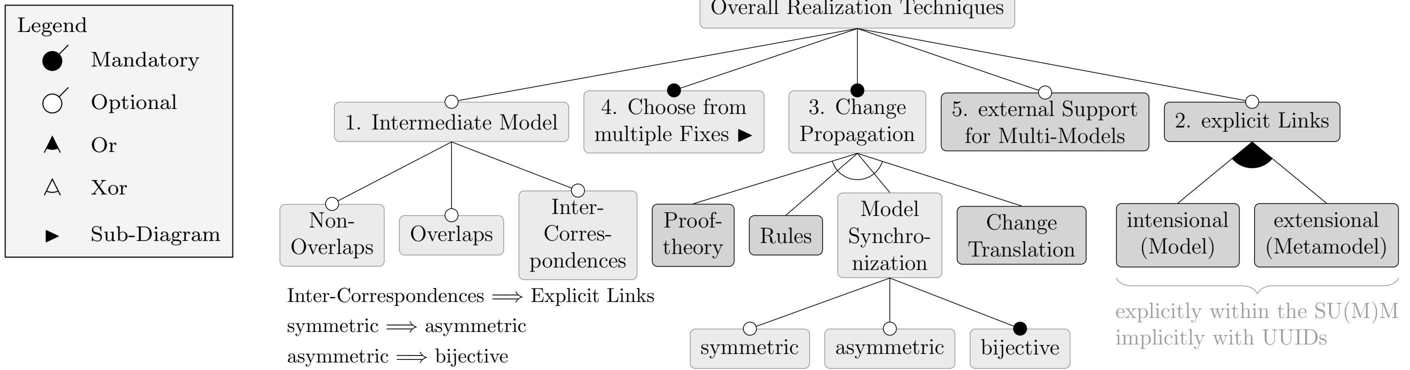


Figure 5.7: Selected Design Choices for technical Realization of MoConseMI

The design for the *choose from multiple fixes* strongly depends on the design of use cases (Section 5.2☞171): Methodologists specify the desired consistency with reusable parts of model transformations for model synchronization. This specification contains also the selection of the final fix for inconsistencies according to the consistency rules. These specifications are done *manually and once by methodologists*, for which MoConseMI provides means for configuration (Section 5.2.1☞171). Afterwards, the initial SUM is created automatically and once by executing these specifications (Section 5.2.3☞176). After these preparations, users can use views and change their models. After having finished these changes, users trigger MoConseMI to *automatically propagate their changes* into all depending views and the SUM (Section 5.2.2☞173). In order to support views realized with different techniques, adapter providers can develop additional adapters to bridge technical spaces (Section 5.2.4☞178).

Summarizing MoConseMI, it follows projectional ideas for change propagation with a single, integrated (meta)model called SU(M)M and provides a bottom-up strategy to develop such SU(M)Ms with the reuse of existing (meta)models as starting point and reduced internal dependencies. For realization, MoConseMI uses model synchronization techniques in order to propagate changes and to create the initial SUM. In order to support project-specific consistency, MoConseMI provides means for configurations, which are used by methodologists to manually specify the desired fixes for inconsistencies according

to the project-specific consistency rules and (meta)models. These configurations are automatically executed by MoConseMI in order to fix inconsistencies, which are introduced by users using single views. These design decisions guide the refinement of the design, which is discussed in the following Chapter 6☞185.

Chapter 6

Design

Basing on the design decisions of Chapter 5[☞163], this section develops the design of MoConseMI in detail. In particular, operators are designed as means to reuse parts of model transformations (Section 5.1.6[☞170]) in Section 6.1. Operators transform metamodels and conforming models in-place and can be configured with metamodel decisions (Section 6.2[☞192]) and model decisions (Section 6.3[☞198]). Operators are selected and combined into a tree of operators in order to describe the transformations between views and the SUM (Section 6.4[☞203]). The execution of this tree of configured operators is designed in Section 6.5[☞213], and transforms models and metamodels, whose technical representations are designed in Section 6.6[☞221]. Additionally, changes in models and metamodels are explicitly represented as differences and are required to control the execution of operators (Section 6.7[☞227]).

The main ideas of the execution of operators and their use for defining new view(point)s are published in this publication:

Related MoConseMI Publication

Johannes Meier, Ruthbetha Kateule and Andreas Winter (2020): *Operator-based viewpoint definition.* In: MODELSWARD 2020 - Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, pp. 401–408.

This publication is cited as Meier, Kateule and Winter (2020) in this thesis.

6.1 Operators as Transformations

Since MoConseMI is model synchronization-based (Section 5.1.4[☞168]) with an explicit SUM (Section 5.1.2[☞165]), *model transformation* definitions must be specified between the (partial) viewpoints and the SUMM. These model transformation definitions are executed (as designed in Section 6.5[☞213]) to propagate changes between the (partial) views and the SUM. This leads to the following two *demands* on model transformations:

Transform Models *and* Metamodels

1. As an impact of the design decision to use model synchronization (Section 5.1.4[☞168]) between reused views (Section 5.1.1[☞164]) and the explicit and optimized SUM (Section 5.1.3[☞167]), model transformations for the model level are not sufficient, since the SUMM (and metamodels for new viewpoints) are not available before MoConseMI is introduced, but are required to define model transformations and to guide conforming models. Therefore, the metamodel level must be supported as well. The need for creation and maintenance of metamodels for model transformations is strongly motivated by Kainz, Buckl and Knoll (2012).

Transformations in two Directions

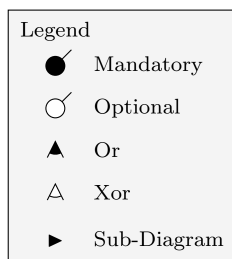
2. Since changes should be propagated from one view via the SUM to other views (Section 5.1.2☞165), the model transformations between views and the SUM (Section 5.1.4☞168) must support transformations in both directions.

According to Section 5.1.6☞170, no single, compact model transformation definition should be specified between a viewpoint and its SUMM, but the required model transformation should be split into parts: These parts allow to fulfill the listed demands in a reusable way are called *operators*. This section specifies only the design of such operators in general, but no concrete operators (except for examples), since the list of operators provided by MoConseMI is documented in Chapter 7☞241.

Before developing the design of operators to fulfill these demands in Section 6.1.3☞189, related work for operators (Section 6.1.1) and model transformations (Section 6.1.2☞188) is investigated as preparation.

6.1.1 Related Work: Operator-based Approaches

This section investigates related approaches which are operator-based, even when other terms like patterns or operations are used. Approaches from various domains beside model transformation are considered. The features found during the analysis of related approaches are summarized in the feature model in Figure 6.1 as result of this investigation. Afterwards, the feature model helps to design the operators of MoConseMI in Section 6.1.3☞189.



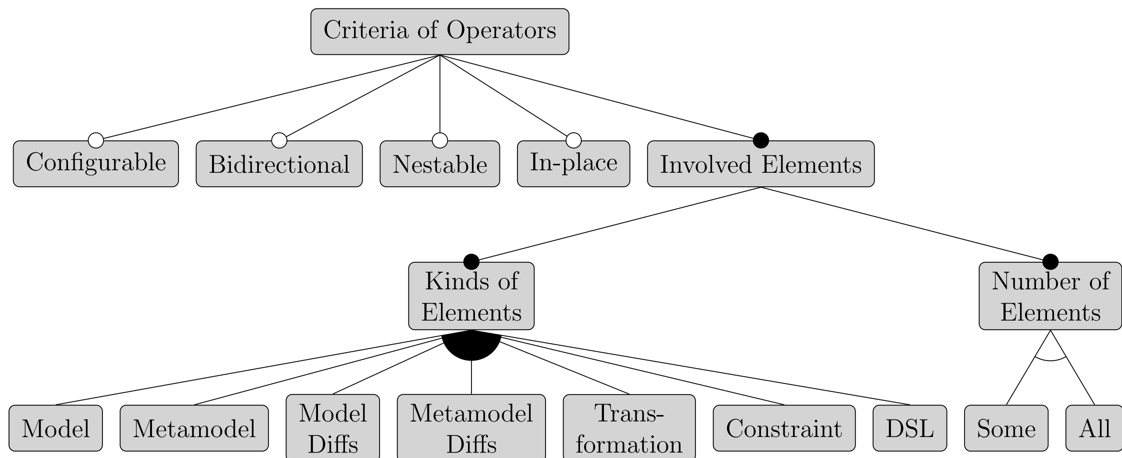


Figure 6.1: Feature Model for classifying Operator-based Approaches

In general and according to Lano, Kolahdouz-Rahimi et al. (2018), patterns are used in model transformations, often unsystematically, but with increasing trend. Lano, Kolahdouz-Rahimi et al. (2014) define multiple formal patterns for model transformation rules. In the area of multi-level modeling, Kainz, Buckl and Knoll (2011) use operators to transform models between meta-levels in order to reuse modeling tools which support only two meta-levels. These operators are configurable in order to cover recurring transformation patterns, which are chained to build whole model-to-metamodel-transformations.

Model Management Operators

Operators for *model management* are not only applied in data base research (Section 3.6.3☞139), but also for models: These operators work on *complete* models (feature "All" in Figure 6.1) and are therefore on the same conceptual level as megamodels (Chapter 2☞51) with exemplary operators like match, merge, diff and slice (Salay, Kokaly et al., 2020). These techniques are often also investigated outside of model management, like model merging (Kolovos, Paige and Polack, 2006). Bernstein (2003) shows model management operators and its application to schema integration, schema evolution and round-trip engineering, but on conceptual level (Section 3.6.3☞139). Chechik, Nejati and Sabetzadeh

(2012) use three of these model management operators (compose, weave, merge) for the integration of models. Kensche and Quix (2007) present the operators export and import to bridge models between different technical spaces. Since these operators are designed to work in a generic i. e. metamodel-independent way on complete models, usually there is less need for configurations of these operators.

more generic and high-level Operators

On granularity level of working with complete models, there are some more operator-based approaches: Persson, Torngren et al. (2013) define comparable operators like composition, projection, extension and synthesis, which are applicable to whole models, which are used for views. Reiter, Kapsammer et al. (2005) wrap model weaving (Chapter 3 ☞ 93) and sewing (a loosely coupling variant of weaving) into operators. Degueule, Combemale et al. (2015) provide similar higher-level operators for the composition of multiple DSLs. Broy, Feilkas et al. (2010) give an overview of related approaches for composing modeling languages with high-level operators. In the enterprise domain, Kühn, Bayer and Karagiannis (2003) present some higher-level patterns similar to the procedure for metamodel integration. Specific for the domain of feature models, Acher, Collet et al. (2010) present different high-level operators for merging feature models.

Metamodel Evolution with Co-Evolution of depending Artifacts

In the area of *X co-evolution*, operator-based approaches are often used: If a metamodel evolves, depending artifacts *X* must co-evolve, too, in order to keep X conform to the changed metamodel. Since the evolution of metamodels is coupled with the evolution of depending artifacts, this challenge is also called the *coupled evolution* problem (Di Ruscio, Iovino and Pierantonio, 2012a). While change propagation aims to keep depending models consistent to each other in *ongoing* processes, co-evolution is done *once* for each change in metamodels, leading to the term *model migration* for model co-evolution. Therefore, co-evolution for depending artifacts is specified only for one direction from the old to the new version of the artifacts and is not bidirectional. Since metamodels and conforming models are on different meta-levels, Visser (2008) uses the term two-level transformation. Since changes in metamodels lead to changes in depending artifacts like models, representing changes explicitly as (meta)model differences is important, but is discussed in its own Section 6.7.1 ☞ 228.

There are lots of different artifacts which depend on metamodels and must be co-evolved with evolving metamodels. Some examples for depending modeling artifacts and co-evolution approaches are given:

Model Co-Evolution targets models conforming to a metamodel: If this metamodel is changed, models conforming to the old version of the metamodel must be co-evolved in order to conform to the new version of the metamodel. Since model co-evolution addresses changes of models *and* their metamodels, as required for MoConseMI as well, related approaches are investigated in more detail in Section 6.2 ☞ 192.

Constraint Co-Evolution targets constraints which complement metamodels and must be fulfilled by models conforming to that metamodel. These constraints explicitly address elements of the metamodel and must be co-evolved according to changes within this metamodel. There are different approaches for constraint co-evolution, including approaches focusing on multiplicity constraints (Taentzer, Mantz et al., 2013) and OCL constraints (Khelladi, Hebig et al., 2016).

Transformation Co-Evolution targets model transformations, which are usually defined on a source metamodel and a target metamodel. Changes in the source metamodel must be reflected by the model transformation. Rutle, Iovino et al. (2018) formalize automatically resolvable transformation co-evolution scenarios (Rutle, Iovino et al., 2020). Other approaches including Kruse (2011), Levendovszky, Balasubramanian et al. (2010), Garcés, Vara et al. (2014) and García, Diaz and Azanza (2013) provide semi-automated solutions for more generic settings. Users of these

approaches have to provide additional information, how to solve certain co-evolution cases.

DSL Co-Evolution targets definitions for DSLs or other concrete syntaxes for model editors. These definitions base on the metamodel of the models to edit and must be changed according to metamodel changes. An example for the co-evolution of the Eclipse Graphical Modeling Framework (GMF) is given by Di Ruscio, Lämmel and Pierantonio (2011).

These definitions are models conforming to metamodels provided by GMF. If GMF and its metamodels change, e. g. due to new versions of GMF, it is necessary to co-evolve DSL definition models developed with earlier versions of GMF accordingly (Herrmannsdoerfer, Ratiu and Wachsmuth, 2010), which falls into the category of model co-evolution (see above).

Granularity of Operators oriented along Metamodel Changes

These depending artifacts are depicted as features of "Kinds of Elements" in Figure 6.1☞186. Most of these examples use operators explicitly or implicitly by patterns or rules, which are oriented at small metamodel changes. Therefore, only *some* elements of metamodels and depending artifacts are changed by these operators. The granularity of operators is often oriented at single metamodel changes and these operators provide the corresponding co-evolution strategy for that metamodel change. Changes in metamodels can be done for the purpose of refactoring, leading to refactoring operators which realize the model co-evolution, as the example of Reimann, Seifert and Aßmann (2010) shows.

Co-Evolution Operators require Configurations

Since co-evolution is also a problem which can be solved with model transformations (Paige, Matragkas and Rose, 2016), operators for co-evolution are a possible choice for model transformations here. Since the desired automation of co-evolution operators reduces the control over the details of the co-evolution (Paige, Matragkas and Rose, 2016), possible configurations for operators are needed (feature "configurable" in Figure 6.1☞186), realized as *model decisions* here and discussed in Section 6.3☞198. Similarily, Di Ruscio, Iovino and Pierantonio (2012b, p. 29f) propose to define reusable patterns with a default co-evolution strategy, which can be extended and customized. Co-evolution cases without automated solution require additional information by users of approaches for co-evolution, but usually do not explicitly distinguish users and methodologists from each other. Since users should be supported with automated decisions, methodologists have to specify additional information or custom co-evolution strategies.

representing Model Changes with Operators

Single (meta)model differences are sometimes called operators, which are investigated in Section 6.7.1☞228. They are not configurable, since they represent only historic changes or can be executed *in-place* to get an updated version. Composite changes are sequences of changes, leading to *nestable* operators in terms of Figure 6.1☞186. The relevant elements of these operators are metamodel differences and model differences.

Summary: Operators often used and varying regarding Granularity

Summarizing, operators vary in the levels of their *granularity*, ranging from generic operators between complete models like model merging over co-evolution operators changing some parts of depending models to model difference operators changing single elements of models. The discussed features of operators are summarized in Figure 6.1☞186. The wide-spread use of operators for different purposes shows, that operators work and can capture recurring tasks. Therefore, following the concepts of operators is a reasonable choice for structuring model synchronization in MoConseMI.

6.1.2 Related Work: Model Transformations

Since MoConseMI uses model synchronization techniques with model transformations for change propagation (Section 5.1.4☞168), this section takes up the discussion in Section 3.3.1☞108 about model transformations and applies the most relevant findings here.

Model transformations for both directions are required (demand 2), which is explicitly addressed by BX (summarized in Figure 3.1 ☞112): EVL+STRACE and JTL do not provide the required automated selection of fixes, while QVT-R and JTL are not incremental. The expressiveness of TGGs is limited in contrast to BXTEND(DSL), which uses imperative definitions in detail. In general, Bucchiarone, Cabot et al. (2020) call bidirectional transformations for model synchronization a grand challenge.

BX for both Transformation Directions

For non-bijective transformation cases, it is important to keep information unchanged which is not covered by the transformation: For out-place model transformations as the discussed BX approaches, this is ensured with incrementality. For in-place model transformations, incrementality is not applicable, since it requires two different models explicitly linked with each other, while in-place model transformations have only one model. Therefore, other means are required to prevent information loss.

Keep non-transformed Information unchanged

Usually, model transformations are applied to transform only models and no metamodels, while transformations of metamodels are required here as well (demand 1). Since metamodels are also models, additional model transformations can be defined which work on source metamodels and target metamodels. But all investigated model transformations can transform either models or metamodels, not both. Also an action-based approach managing models like Mosser and Blay-Fornarino (2013) addresses only the model level and not the metamodels. An exception are model transformation approaches which are designed to work with multi-level models like DEEPATL, which supports only unidirectional transformations. Model transformations supporting multi-level models are not used here, since this thesis is restricted to two-level modeling.

Missing Approaches for transforming Model *and* its Metamodel together

Additionally, reuse and modularization are still a challenge in model transformation approaches (Götz, Tichy and Groner, 2021, p. 480f), which is deepened in Section 6.4.1 ☞203. Here it is sufficient to notice, that the discussed approaches provide no explicit mechanisms for reuse of model transformation parts, in particular, not for reuse in different transformation scenarios.

Limited Support for Reuse

Summarizing, there are no model transformation approaches which support all the required features. More concrete, the joint transformation of models and their metamodels together with reuse and two directions is not supported. Therefore, Section 6.1.3 develops a new approach for model transformations with the desired features.

6.1.3 Design

This section describes the main design of operators along Figure 6.2 ☞190, which depicts the designed signature of operators. The depicted operators are *unidirectional operators*, which are introduced first with its features in Figure 6.3 ☞191. Afterwards, two inverse unidirectional operators are combined as *bidirectional operator*.

Since transformations for models are not sufficient (demand 1), *unidirectional operators* in MOCONSEMI change both a model and its metamodel. Therefore, one model and its metamodel are the input for an operator, and a (changed) model and its (changed) metamodel are the output. This design enables model transformations for change propagation according to model synchronization techniques *and* keeps models conform to their metamodels. Since the data types for input and output are the same, i. e. one model and its metamodel, operators are chainable, as deepened in Section 6.4 ☞203. Since the metamodel is explicitly transformed, operators can create not only the initial SUM, but also the SUMM. Additionally, this design allows operators to internally mesh transformations of models and their metamodels, e. g. first change the metamodel a bit, then change the model and finally change the metamodel again. In contrast, traditional model transformations would need two transformations, one for the models and another one for the metamodel, which are executed at once and independently from each other.

Operators transform Models and their Metamodels together

Since metamodels and conforming models are transformed, operators of MOCONSEMI

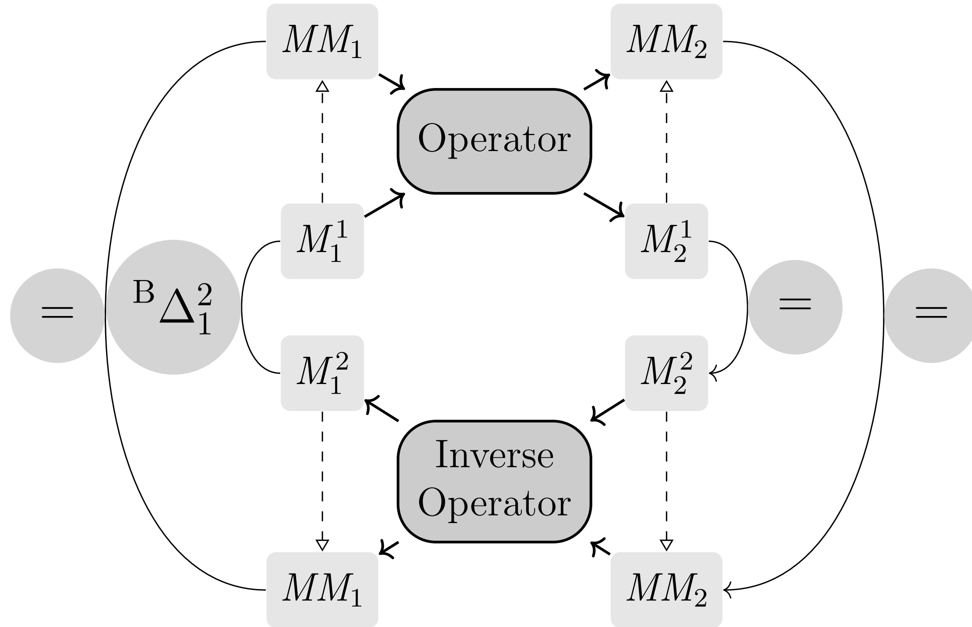


Figure 6.2: Main Design of the Operator Signature

Granularity of Operators: single/few Metamodel Changes

have some similarities with operators for model co-evolution. The investigations in Section 6.1.1[☞186] show, that operators are useful for this purpose. Additionally, their granularity is taken over here with few changes in the metamodel and more changes for the model co-evolution.

Operators work *in-place*!

As deepened in Section 6.5[☞213], operators are executed *in-place* and not out-place: Since operators change only some parts and lots of operators should be chained, executing operators in out-place way would copy all unchanged elements of (meta)models into a new (meta)model for each operator (Kainz, Buckl and Knoll, 2012). In order to improve this unnecessary performance overhead, operators are executed *in-place* by changing only the elements to be changed directly within the input (meta)model. Therefore, the output of operators is the input in changed form. As depicted in Figure 6.2, the model M and its metamodel MM in version 1 (indicated by subscript 1) are changed in-place by the operator into version 2. Afterwards, M_2 must conform to MM_2. Further explanations of Figure 6.2 fill follow further down.

Summary of Unidirectional Operators

This design of unidirectional operators is depicted in Figure 6.3[☞191] as selection of the features of Figure 6.1[☞186]: Unidirectional operators work on *some* elements of a given *model* and its *metamodel* and change them *in-place*. Additionally, unidirectional operators are *nestable* to realize composite changes and to reuse functionalities of other unidirectional operators, if reasonable. In order to be applicable to arbitrary (meta)models (according to Requirement R 1.1 (Generic Metamodels)[☞154]) and to ensure arbitrary consistency goals (according to Requirement R 1.2 (Generic Consistency Goals)[☞155]), unidirectional operators are *configurable*, which is realized with metamodel decisions, as designed in Section 6.2[☞192], and with model decisions, as designed in Section 6.3[☞198].

This design of unidirectional operators is summarized in Definition 22:

> Definition 22: **Unidirectional Operator**
>
> A unidirectional operator is a unidirectional and in-place model transformation definition on one model and its metamodel.

In order to support transformations in two directions (demand 2), one unidirectional operator is combined with its inverse unidirectional operator into one *bidirectional operator*

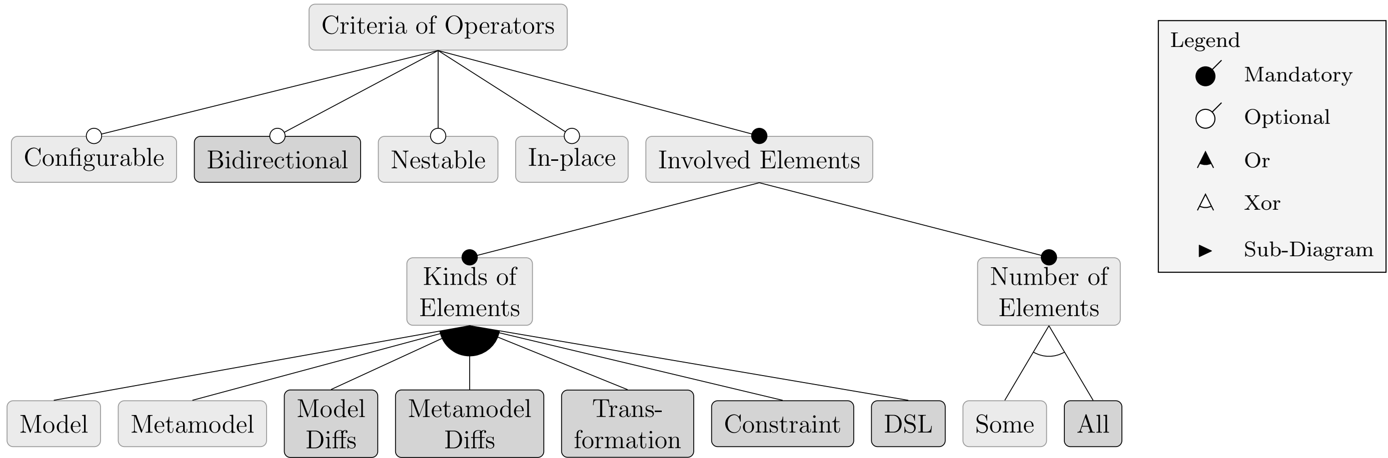


Figure 6.3: Selected Features (light gray) for unidirectional Operators in MoConseMI

(Figure 6.2☞190): Two unidirectional operators are inverse to each other, if the changed metamodel of the first operator (MM_2 in Figure 6.2☞190) can be used as input for the second operator and the (again) changed metamodel is equal to the input metamodel of the first operator (MM_1). With other words, the metamodel changes of the second (inverse) operator exactly revert the metamodel changes of the first operator. This design ensures, that metamodels of viewpoints and the SUMM remain stable, when executing bidirectional operators often and in changing directions. In contrast, model changes of the two unidirectional operators often are *not inverse* to each other. This is required to fix existing inconsistencies in the models and to propagate changes in the model of a view to the SUM and to all other views. Both these cases lead to different models by the two unidirectional operators in consecutive executions. Since these models might be different, their symbols in Figure 6.2☞190 have not only subscript numbers as the metamodels have, but also superscript numbers indicating the current execution by unidirectional operators. In Figure 6.2☞190, superscript 1 indicates the forward execution, while superscript 2 is for the inverse execution (and superscript 3 (not in Figure 6.2☞190) would be the forward execution again and so on). The differences between models of consecutive executions are called *branch differences* ${}^{B}\Delta$, e. g. ${}^{B}\Delta_1^2$ describes the differences between the model M_1^1 and the model M_1^2. Summarizing, Figure 6.2☞190 depicts one bidirectional operator, which consists of two unidirectional operators which are inverse to each other regarding the metamodel changes.

Compose two inverse unidirectional Operators into a bidirectional Operator

With this design, the features of unidirectional operators in Figure 6.3 and bidirectional operators in Figure 6.4☞192 are similar, but not same: By design, only bidirectional operators are *bidirectional*, and unidirectional operators are only unidirectional. While unidirectional operators are *nestable*, bidirectional operators could also be nestable in principle, but here they are designed to be *not nestable*, since nesting of bidirectional operators is not required and provides no significant benefits: Bidirectional operators are directly used "for management" for chaining (only bidirectional) operators and for internally combining two unidirectional operators which are inverse to each other. Unidirectional operators focus on executing unidirectional transformations and can (re)use other unidirectional operators for nesting.

Comparison of unidirectional and bidirectional Operators

This design of bidirectional operators is summarized in Definition 23☞192:

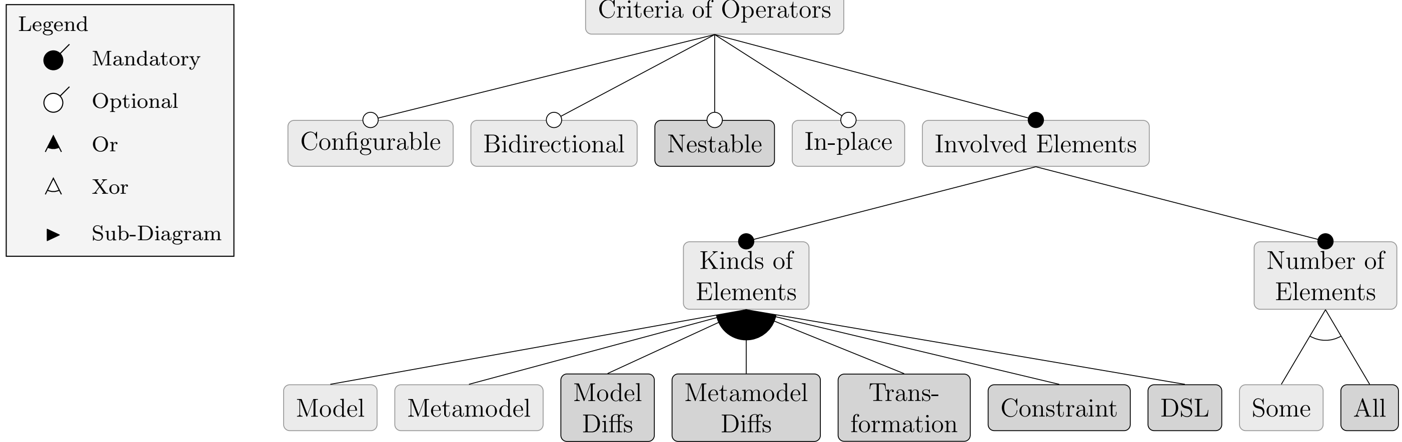


Figure 6.4: Selected Features (light gray) for bidirectional Operators in MoConseMI

> Definition 23: **Bidirectional Operator**
>
> A bidirectional operator is a bidirectional and in-place model transformation definition on one model and its metamodel, which is composed by a corresponding unidirectional operator and its inverse unidirectional operator.

⇄AddDeleteAssociation with →AddAssociation and ←DeleteAssociation

Summarizing, operators are used to define reusable parts of model transformations in MoConseMI (Section 5.1.6 ☞170). The conceptual design of these operators is developed along Figure 6.2 ☞190. As an example, the bidirectional operator ⇄AddDeleteAssociation consists of the unidirectional operator →AddAssociation and its inverse unidirectional ←DeleteAssociation. →AddAssociation creates a new association in the metamodel and allows to create conforming links in the model, controlled by a model decision. ←DeleteAssociation deletes this association in the metamodel and deletes all conforming links in the model in any case. Additionally, ⇄DeleteAddAssociation can be easily developed by switching the unidirectional operators of ⇄AddDeleteAssociation, i. e. by combining →DeleteAssociation and ←AddAssociation as inverse unidirectional operator. Some more examples for operators[12] fulfilling this concept are named in the following Section 6.2, which are collected in a library for reuse and combination by methodologists (Section 6.4 ☞203).

MoConseMI is Operator-based

Summarizing, *bidirectional operators* are composed of two *unidirectional operators* which are inverse to each other in order to enable bidirectional executions (demand 2). Depending on the current direction, one of these unidirectional operators is executed and changes one model and its metamodel in-place (demand 1). Changing both a model together with its metamodel fulfills the challenge to create the initial SUM and its conforming SUMM (last impact of Section 5.1.4 ☞168). Means for configuring unidirectional and bidirectional operators are discussed in the following sections in order to realize project-specific consistency with reusable operators.

6.2 Metamodel Decisions

Since MoConseMI splits definitions for model synchronization into parts for their reuse (Section 5.1.6 ☞170), Section 6.1 ☞185 introduces the general design for bidirectional and uni-

[12]When using the term operator, usually the statement can be applied to both unidirectional and bidirectional operators. Otherwise, the context makes clear, that operators currently mean either unidirectional or bidirectional operators.

directional operators. Their designs are deepened here regarding means for configuration. There are the following two *demands* on both bidirectional operators and unidirectional operators:

1. Since MoConseMI uses operators in order to enable the reuse of parts of definitions for model synchronization (Section 5.1.6 ☞170), these operators must be *generic* to be applicable for different projects on the one hand.

generic Operator for Reuse

2. On the other hand, the consistency goals (Requirement R 1.2 (Generic Consistency Goals) ☞155) and the (meta)models (Requirement R 1.1 (Generic Metamodels) ☞154) are different for each project and therefore require project-specific definitions for the actual use in particular projects, leading to *project-specific* operators.

project-specific Operators for actual Use

In order to fulfill these two demands, the main idea is to design operators which are generic for reuse in general (demand 1) and are configurable for project-specific adaptations in detail (demand 2). This counts for both bidirectional and unidirectional operators (leading to the selected feature "configurable" in Figure 6.3 ☞191 and Figure 6.4 ☞192). These configurations are called *decisions*. Kleppe, Warmer and Bast (2003, p. 78f) call them "transformation parameters", but the term "decision" emphasizes the active role for deciding them more than "parameter".

Decisions ...

Since operators transform metamodels and models, decisions for configuring the transformation of metamodels *and* decisions for configuring the transformation of models are required. Since metamodels specify, which models are conforming to them, and since metamodels are means to specify consistency independently from actual models (Section 2.2.2 ☞60), this Section 6.2 ☞192 investigates metamodel decisions first, before model decisions are investigated in Section 6.3 ☞198. This is also indicated by the research for model co-evolution, which derives changes for models from metamodel changes which occurred before. Therefore, related approaches for model co-evolution are investigated in the following Section 6.2.1.

... for Metamodels and Models

6.2.1 Related Work: Model Co-Evolution

This section investigates related approaches for model co-evolution, since they target the joint evolution of models and metamodels and often use operator-based approaches (Section 6.1.1 ☞186), very similar to the design of operators in MoConseMI (Section 6.1.3 ☞189). The Transformation Tool Contest 2010 (Rose, Herrmannsdoerfer et al., 2012, p. 350), comparing different tools for model co-evolution including generic model transformations, found, that specific tools for model co-evolution are more suited than generic model transformations, in particular in terms of conciseness and understandability, since specific tools can focus on the differences between the (meta)model versions, while generic model transformations must handle also unchanged parts of (meta)models. Depending on their provided features, such behavior could be reconstructed with generic model transformations, as discussed e. g. for the GReTL transformation language (Ebert and Horn, 2014, p. 316). Additionally, generic model transformations handle only one meta-level, as discussed in Section 6.1.2 ☞188. Therefore, only specific approaches for model co-evolution are discussed in this section.

Focus on specific Approaches for Model Co-Evolution

Hebig, Khelladi and Bendraou (2017) classify and survey approaches dedicated to model co-evolution. They classify approaches into five groups, regarding the resolution strategies, how the required model changes depending on the metamodel changes are found:

Strategies to identify the required Model Changes

Resolution Strategy Generation approaches generate multiple possible resolution strategies depending on the current metamodel change and are not investigated here, since they only provide possible candidates for resolution strategies, while the methodologist usually knows, which resolution strategy is desired by users.

Resolution Strategy Learning approaches learn resolution strategies from previously selected resolutions and are not interesting here, since the resolution strategy to realize is already known by the needs of users. An example is provided by Kessentini, Sahraoui and Wimmer (2016), who apply machine learning techniques with multi-objective optimization to search and find useful candidates for co-evolved models.

Constrained Model Search approaches search for possible model changes or updated models, depending on the current metamodel change *and* the current model, and are not interesting here, since these approaches force users to manually select the desired resolution, while users should be provided with automated solutions. An example is provided by Demuth, Riedl-Ehrenleitner et al. (2016), who apply their approaches for rule-based consistency management (Section 3.3.1☞108) to model co-evolution, since the conformance of models to their metamodel can be described also with rules.

Resolution Strategy Languages are approaches which use tailored model transformation languages to co-evolve models from the old to the new metamodel version. These approaches are interesting here, since they allow methodologists to explicitly and once specify the desired model changes.

Predefined Resolution Strategies approaches provide predefined resolution strategies for metamodel changes and are interesting here, since they allow methodologists to reuse resolution strategies.

Therefore, only the last two groups of model co-evolution approaches are deepened with some exemplary approaches.

Epsilon Flock

In the groups of *resolution strategy languages* approaches, Epsilon Flock (Rose, Kolovos et al., 2014) is an exogeneous model-to-model-transformation approach tailored to the transformation of models from a source metamodel to a target metamodel, which is an evolved version of the source metamodel. While the transformation of objects conforming to evolved parts of the metamodel must be explicitly and manually defined according to the current evolution scenario, objects conforming to unchanged or only renamed parts of the metamodel are automatically copied from the source model to the target model. Manual definitions contain mappings between classes in the source metamodel and classes in the target metamodel as well as conditions and statements written with EOL (Paige, Kolovos et al., 2009). Defining more generic co-evolution for special cases is marked as future work (Rose, Kolovos et al., 2014, p. 753).

EMFMigrate

Another example is EMFMigrate (Wagelaar, Iovino et al., 2012), which supports not only model co-evolution, but aims to support the co-evolution of any depending artifacts: Rules can be defined, which combine one metamodel change with actions, which are executed in-place on the depending artifact to co-evolve it according to the metamodel change. These rules can be collected and organized in libraries for their reuse and customization.

Model Change Language (MCL)

The Model Change Language (MCL) (Narayanan, Levendovszky et al., 2009) is a DSL for specifying model co-evolution with graphical support and formalizations in Levendovszky, Balasubramanian et al. (2013). MCL explicitly maps parts of the old metamodel to parts of the new metamodel in graphical way and specifies details in imperative and textual way. Since MCL is a DSL, it supports usual and recurring cases for reuse, but complex and very specific cases require to use other approaches like generic model transformations (Levendovszky, Balasubramanian et al., 2013, p. 821).

enable in-place Model Transformations for Model Co-Evolution

Wimmer, Kusel et al. (2010) provide a strategy to enable the use of generic in-place model transformations for the model co-evolution between two different metamodels. This approach restricts the metamodel evolution by allowing metamodel elements to be only either deleted, created or unchanged, but not to be changed or refactored. Accordingly, model elements in the updated model are either newly created by hand-written in-place

model transformations or kept unchanged, while model elements which do not conform to the new metamodel anymore are deleted automatically.

In the groups of *predefined resolution strategies* approaches, Cicchetti, Di Ruscio et al. (2008) provide lots of change scenarios (similar to operators) for metamodels and predefined automated model co-evolution strategies, which can be refined, if relevant information for the model co-evolution is missing. They propose to chain change scenarios one after another to realize bigger change scenarios.

Edapt

Another, more elaborated example is Edapt: Edapt provides lots of *coupled operators*, explicitly designed for reuse, which couple small changes in the metamodel with automated changes for co-evolution of conforming models (Herrmannsdoerfer, Vermolen and Wachsmuth, 2011). These coupled operators are chained in order to explicitly model the change history of metamodels. Edapt is provided as graphically Eclipse plugin and allows to graphically select and configure coupled operators to actively evolve metamodels (Herrmannsdoerfer, 2010). The corresponding model co-evolution is predefined and cannot be adapted. Custom model co-evolution can be manually realized with explicit code written in Groovy (Herrmannsdoerfer, Benz and E, 2008). Therefore, an easy reuse is possible only for the predefined operators, whose model co-evolution cannot be adapted to project-specific needs, violating Requirement R 1.2 (Generic Consistency Goals)☞155. Additionally, Edapt supports only model co-evolution in one direction, although the predefined operators are introduced with their conceptual inverse operators by Herrmannsdoerfer, Vermolen and Wachsmuth (2011). Initially, Edapt was called COPE (Herrmannsdoerfer, Benz and Juergens, 2009). The approach of Wachsmuth (2007) using QVT-R for co-evolving models according to metamodel changes can be seen as predecessor for COPE/Edapt. Additionally, Wachsmuth (2007, p. 621) explicitly requires in-place transformations for realizing model co-evolution.

Data Base Co-Evolution

Data base co-evolution deals with evolving schemata of data bases (Section 3.6.3☞139) and corresponding co-evolution for the instances within data bases, which is still a challenge in practice (Delplanque, Etien et al., 2018) and in research (Möller, Scherzinger et al., 2020): The counterpart to model co-evolution is called *schema evolution* (Rahm and Bernstein, 2006) in data base research and focuses on evolving data base schemata without information loss. Two examples are CHiSEL using unidirectional operators to transform schemata and instances together (Schuler and Kessleman, 2019), and BiDEL using bidirectional operators to transform schemata and instances together (Herrmann, Voigt et al., 2018). On top of schema evolution, *schema versioning* manages the history of schema versions and enables to access data also with older versions of the schema (Roddick, 1995). These examples show, that operator-based co-evolution is successful also for evolving data bases. Additionally, schema evolution and model co-evolution are very similar, since both target evolution of schema information and co-evolution of conforming instance data (Milovanovic and Milicev, 2015).

Ontology Co-Evolution

Evolution occurs also in *ontologies* (Section 3.6.4☞142) with changes in ontology schemata and co-evolution changes in ontology instances: Noy and Klein (2004) give an introduction to ontology co-evolution and propose to use (unidirectional) operators which are oriented at few changes in the schema and co-evolution changes in the instances. Since ontologies can have more than two meta-levels, co-evolution must cover all lower meta-levels and not only one (Noy and Klein, 2004).

Summary

Summarizing the lessons learned from investigating approaches for model co-evolution, operator-based approaches are often used and are useful, since predefined operators are reusable for recurring metamodel evolution scenarios. Examples for such operators, which are often used by the investigated related approaches, include *creation of single metamodel elements* like add class and add attribute, *deletion of single metamodel elements* like delete class and delete association, *change of single metamodel elements* like change attribute type, change multiplicity and rename, and more *complex scenarios* like to move an attribute from

one class to another class. Since these metamodel changes are small, often consisting of single metamodel changes, complex metamodel evolution scenarios can be split into parts, which are described by such operators. Operators oriented along small metamodel changes are not only used for model co-evolution, but also for the co-evolution of data bases and ontologies.

While all approaches target model co-evolution, only some approaches explicitly perform also the metamodel evolution, like EDAPT, while other approaches perform only the model migration between two different metamodels, like most resolution strategy languages. All investigated approaches target *only one* model and its metamodel, while MoConseMI needs to transform *multiple* (meta)models into the SU(M)M, which is designed in Section 6.4.3☞205.

Adaptations for the Model Co-Evolution required

Another important finding is, that there are multiple possible strategies to co-evolve the same model according to the same metamodel changes. Gruschko, Kolovos and Paige (2007) classify the impact of metamodel changes into non breaking changes, breaking and resolvable changes, and breaking and unresolvable changes:

- Creating new metamodel elements is usually *non breaking*, since they enable new model elements, but do not expect on the one hand, that such model elements exist just after the metamodel evolution. On the other hand, depending on the needs of the current project, some model elements could be initialized.
- Renaming a metamodel element is *breaking and resolvable*, since conforming model elements referring to the renamed metamodel element by their name need to update this name, which can be done automatically without needing any adaptations.
- Changing types of attributes is *breaking and unresolvable* in general, since old values like floating-point values must be converted to the new type, which is unclear e. g. for `boolean` as new type and requires a project-specific co-evolution.
- Deleting metamodel elements could be automated by deleting all conforming model elements (non breaking), but this leads to information loss and might be prevented in some cases by defining custom model co-evolutions.

Summarizing, even for non breaking and breaking and resolvable metamodel changes, model co-evolution provides some degrees of freedom, which can be used for project-specific adaptations. Predefined resolution strategies often support only one variant of the model co-evolution. Therefore, additional configurations for the model level are required, which are discussed in Section 6.3☞198. Additionally, support for custom evolution scenarios is required for special cases.

6.2.2 Design

This sections extends the general design of operators in Figure 6.2☞190 with *metamodel decisions*, as depicted in Figure 6.5☞197. Metamodel decisions are means to control the changes of unidirectional operators on metamodel level. By configuring metamodel decisions, methodologists reuse and adapt generic and reusable operators (demand 1) for the needs of the current project (demand 2). As an example, a generic →ADDASSOCIATION operator creates a new association between two existing classes. This case can occur in different evolution scenarios and can be adapted to the current situation with metamodel decisions for the two existing classes A, B. Accordingly, the methodologist takes the operator provided by MoConseMI and fills these metamodel decisions with the two particular names of the classes in the current metamodel to evolve. Other metamodel decisions control the role names and multiplicities of the new association.

Metamodel Decisions for unidirectional Operators

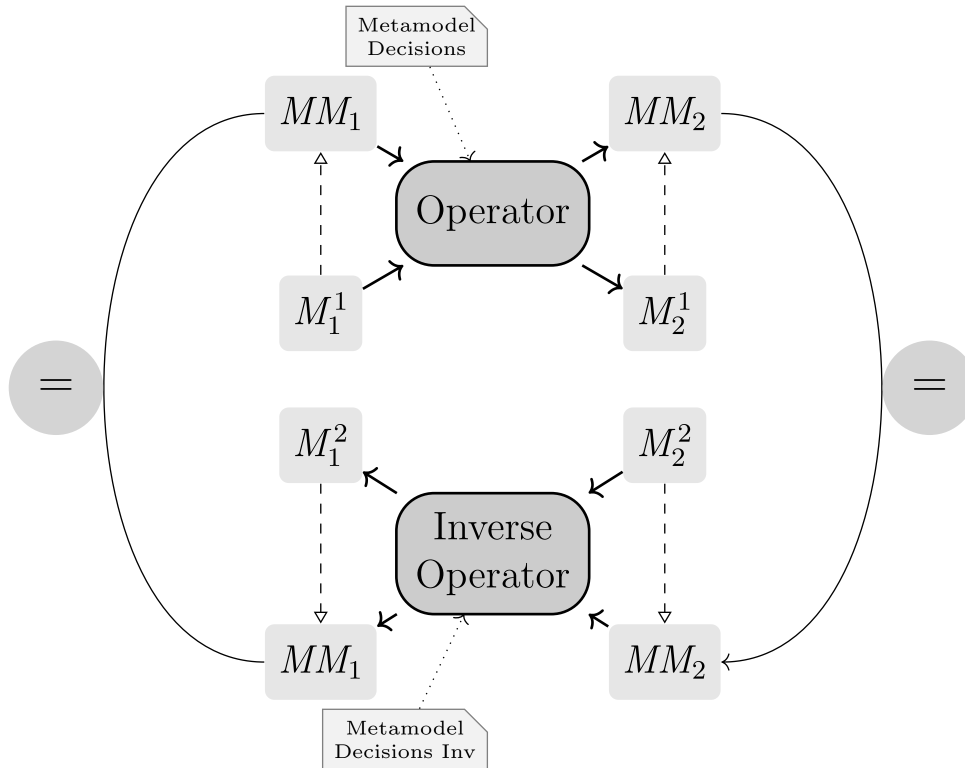


Figure 6.5: Metamodel Decisions for the Operator Signature

This general design for *unidirectional operators* is strongly influenced by EDAPT, as investigated above: EDAPT covers both levels for models and metamodels together and provides reusable operators, which are oriented along small metamodel changes, which can be controlled in detail. Additionally, EDAPT predefines one model co-evolution strategy for each of its operators, while the model co-evolution for MoConseMI is designed later in Section 6.3 ☞198. EDAPT fits here, since it fulfills the needs for explicit transformations of models and their metamodels (Section 6.1.3 ☞189) in a generic (demand 1) and adaptable (demand 2) way. Additionally, EDAPT was successfully evaluated during the Transformation Tool Contest 2010 (Rose, Herrmannsdoerfer et al., 2012), comparing different tools for model co-evolution including generic model transformations.

unidirectional Operators designed according to EDAPT

Since MoConseMI combines unidirectional operators into bidirectional operators, both unidirectional operators might have metamodel decisions: Since the metamodel changes of inverse operators must be inverse to each other, not only the unidirectional operators depend on each other, but also their metamodel decisions and their configurations by methodologists. Therefore, the metamodel decisions of the inverse unidirectional operator are called "Metamodel Decisions Inv" in Figure 6.5. As an example, while →ADDASSOCIATION creates a new association C between the existing classes A and B (see above), its inverse unidirectional operator ←DELETEASSOCIATION must delete exactly this association C. In order to ensure these dependencies, the configurations for the metamodel decisions of the unidirectional operators are given to the bidirectional operator. The bidirectional operator forwards them to the unidirectional operator and *automatically derives* corresponding configurations for the metamodel decisions of the inverse unidirectional operator. Since some information about the metamodel, which is required to derive the inverse metamodel configurations, is initially not known, each bidirectional operator has two methods for this initialization, which are called during the first execution of the operator with the current metamodel as parameter, one of these methods is called before the first execution, the other method is called after the first execution. This design eases the work of methodologists,

Metamodel Decisions for bidirectional Operators

since metamodel decisions need to be configured only once for one direction, and ensures, that unidirectional operators and configurations for their metamodel decisions fit together, as defined in Section 6.1.3☞189.

According to most of the investigated related approaches for operator-based co-evolution, the operators of MoConseMI execute only few changes in the metamodel, since short sequences of changes occur more often than long sequences of changes, i. e. short change sequences can be found more often in evolution scenarios than long change sequences in general. In the extreme case, any metamodel evolution scenario can be split into atomic metamodel changes. Therefore, MoConseMI provides such operators like for creating new, deleting old and changing existing metamodel elements, as documented in Section 7.3☞243. To be reusable, the number of these metamodel changes is fixed for each operator in order to be predictable for methodologists for explicitly specifying the desired metamodel evolution. As an example, →AddAssociation always creates one new association, leading to exactly the metamodel changes for creating the new association and setting its initial values for connected classes, roles names and multiplicities. Therefore, configurations for metamodel decisions are usually single, static values like the names of the two classes (`String` values) and the desired multiplicities (`int` values). Metamodel differences for the representation of the metamodel evolution are discussed in Section 6.7☞227. Since the metamodel changes are small and are done in-place, there is no need to merge different metamodel versions as in Wimmer, Kusel et al. (2010).

Operators oriented along small Metamodel Changes

This design is summarized in Definition 24:

Definition 24: Metamodel Decision

Metamodel decisions are means of unidirectional operators to control their changes on metamodel level. If methodologists apply these operators, they provide metamodel configurations for each of the metamodel decisions.

Summary

Summarizing, the design of operators is oriented along successful co-evolution approaches (Section 6.1.1☞186), while the design of their metamodel decisions is oriented along successful model co-evolution approaches (Section 6.2.1☞193). Metamodel decisions control the changes of generic operators (demand 1) on metamodel level and are means for methodologists to adapt reusable operator to the current project, in particular, to the current metamodel (demand 2). Metamodel decisions are provided by unidirectional operators and configured by methodologists during their reuse. These configurations are given to bidirectional operators, which forward them to the first unidirectional operator and automatically derive corresponding configurations for its inverse unidirectional operator. The configurations for metamodel decisions of operators are done by methodologists during the use case "specify Consistency" (Section 5.2.1☞171).

6.3 Model Decisions

Since MoConseMI uses operators to jointly transform metamodels and models (Section 6.1.3☞189), operators evolve metamodels with small and configurable changes (Section 6.2.2☞196). This section focuses on the design of the required co-evolution of conforming models with the following two *demands*:

project-specific Model Co-Evolution

1. As already motivated in Section 6.2.1☞193, there are multiple variants for the model co-evolution, even for the same metamodel changes and the same current model. Depending on the current needs, different strategies for model co-evolution might be desired in different projects. In particular, it must be ensured, that no still required information is lost by accident i. e. by inaccurate and non-adaptable model co-evolution.

2. Metamodel evolution and model co-evolution are not the main objectives of MoConseMI, but are means to realize model synchronization (Section 5.1.4 ☞ 168) with transformations on model and metamodel level. Instead, the objective is to ensure consistency of models with model synchronization. Therefore, the model transformations must realize the defined consistency rules in order to ensure the consistency goals of the current project.

realize Consistency Rules

In order to fulfill these two demands, the main idea is to make the operators, whose metamodel changes are configurable, configurable by methodologists for the model level as well. These means for configurations are called *model decisions*. Before deepening this design idea in Section 6.3.2 ☞ 200, the next Section 6.3.1 investigates related approaches which allow methodologists to customize model transformations in detail.

6.3.1 Related Work

The general ideas of model co-evolution are already discussed in Section 6.2.1 ☞ 193. An important result is, that model co-evolution can vary even for the same metamodel changes and the same current model. This can be exploited to adapt the model changes according to the needs of the current project, which is done by methodologists (Section 5.1.5 ☞ 169). Therefore, this section investigates related approaches which allow methodologists to customize model transformations in detail, including related approaches for model co-evolution, but also for BX.

For *model co-evolution*, Hebig, Khelladi and Bendraou (2017, p. 405f) explicitly group strategies for model co-evolution regarding the amount of human interaction into so-called "benefit classes": Benefit class $1 : 1$ collects model co-evolution strategies, which require human interaction for each model to co-evolve, and therefore maps to decisions made by users. Benefit class $1 : n$ collects model co-evolution strategies, which require human interaction for each metamodel evolution and handles the co-evolution of all conforming model automatically, and therefore maps to decisions made by methodologists. Benefit class $0 : n$ collects model co-evolution strategies, which require no human interaction at all and therefore maps to situations which can be solved by platform specialists for the whole approach. Here, the benefit class $1 : n$ maps exactly to the cases which should be decided by methodologists and automated afterwards by MoConseMI for users. This shows, that the idea of model decisions for methodologists to configure the desired model co-evolution is reasonable and is already used in practice.

Model Co-Evolution

Additionally, research for *bidirectional transformation* (BX) provides also some examples for customizations of model transformations in details:

BX

- Stevens (2010) motivates in the area of BX to let the methodologist decide once the desired model transformations for the inverse direction.
- Zan, Pacheco and Hu (2014) allow to imperatively customize the desired back-propagation of changes for bidirectional model transformations, written with XML syntax and demonstrated in contrast to QVT-R.
- In the approach of Bank, Buchmann and Westfechtel (2021), a high-level DSL called BXtendDSL is used to declaratively describe the general BX, which is comparable with chaining predefined operators, and is complemented with additional imperative code for the details of the transformation, which is comparable with model decisions.

Similar design ideas can be found also in *data-oriented research*:

Data-oriented Research

- Tran, Kato and Hu (2020) develop an approach to explicitly write code to realize the view-update problem in the data base area, which can be seen as model decisions for the inverse direction of operators here.

GraphQL

- GraphQL (Kress, 2021; Wittern, Cha et al., 2019) is a platform-independent query language for graphs. It is often used as interface in form of an API between server and client in web-based applications, technically often used with REST and JSON. The server has a GraphQL schema describing the structures of the available data, which can be read using queries and changed using mutations by clients. The declaration of allowed queries and mutations as well as their implementation are done on the server, strongly connected with the schema and platform-specific depending on the underlying realization of the data. Here, it is interesting to see, that reading and writing data using GraphQL are completely independent of each other with *two separated specifications*: This design solves the view-update problem by restricting allowed updates and implementing them explicitly.

Summarizing, these investigated related approaches demonstrate, that there is a need for explicit specifications or customizations for model transformations in detail. Although BX research aims to provide model transformations for both directions with a single transformation definition, in practice, there are some approaches which define (parts of) unidirectional transformations explicitly, sometimes in imperative way.

6.3.2 Design

Model Decisions for unidirectional Operators

This section extends the design of operators including metamodel decisions in Figure 6.5 ☞197 with *model decisions*, as depicted in Figure 6.6 ☞201. Model decisions are means to control the changes of unidirectional operators on model level. By configuring model decisions, methodologists adapt the model co-evolution of generic operators (demand 1) according to the needs of the current project, in terms of consistency rules (demand 2). In particular, the possible degrees of freedom during model co-evolution are identified and are made configurable as model decisions. Methodologists use these model decisions to configure the consistency rules in order to ensure the project-specific consistency goals. As an example, after creating a new association in the metamodel, →AddAssociation allows to create additional links in the model by a model decision. Depending on the consistency rules, methodologists can configure to create none or some links, depending on the current model. There are also operators without any model decisions e. g. to rename a metamodel element, since the only reasonable model co-evolution strategy is to update the type name of conforming model elements and this strategy can be automated.

configuring Model Decisions for both unidirectional Operators for Flexibility

Since MoConseMI combines unidirectional operators into bidirectional operators, both unidirectional operators might have model decisions. Since the model changes of such unidirectional operators do not need to be completely inverse to each other (Section 6.1.3 ☞189), in order to fix inconsistencies within the current model, methodologists can explicitly configure the model decisions of both unidirectional operators. While there are technically independent configurations, they have semantic dependencies in order to fulfill the desired consistency rules and to prevent information loss. This design introduces some more additional configuration effort for methodologists, but provides the benefit to explicitly configure the transformations for both directions in unidirectional way, according to the findings of related approaches in Section 6.3.1 ☞199 and Section 3.7 ☞146. This design trades configuration effort for flexibility.

Configurations for Model Decisions take the current Model into account

The configurations for model decisions can depend not only on the current metamodel change and the consistency rules, but also on the current model. Therefore, configurations for model decisions can provide dynamic values, depending on the current model to co-evolve.

While some model decisions are "questions to answer" with predefined values for certain parts of the model, other model decisions require to explicitly change the current model like the model decisions of →AddAssociation to create some links within the model. In

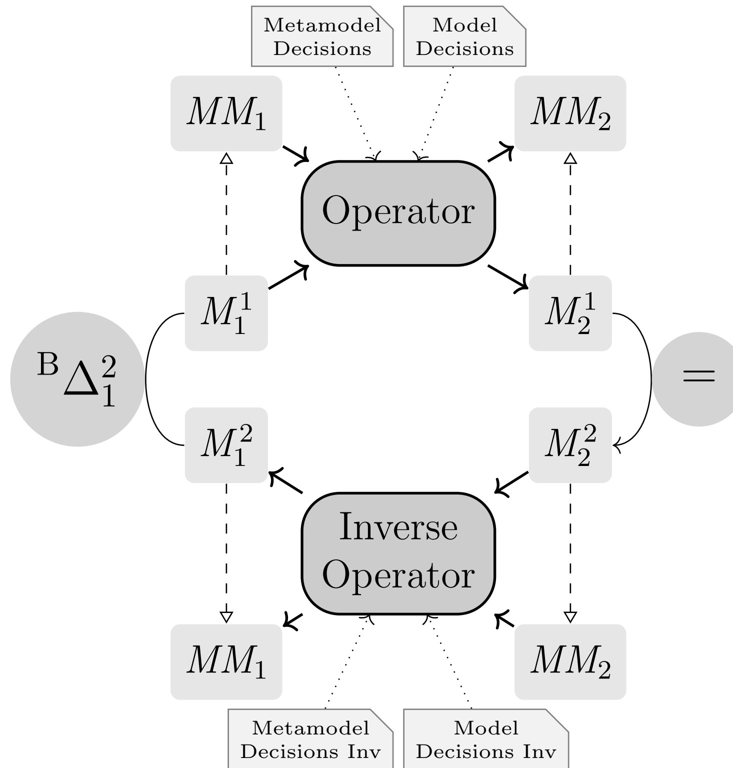


Figure 6.6: Model Decisions for the Operator Signature

order to support methodologists to query the current model and to specify details of model transformations for single directions, *imperative* statements are supported. This fits to the principle of model synchronization with model transformations, which enforce consistency by active overriding of possible inconsistencies with transformations. Since model decisions are configured for each direction, they can be configured in imperative way, while declarative statements of BX languages like TGGs allow to derive transformations for two directions, but decrease flexibility and expressiveness (Section 3.7 ☞ 146). In particular, no formal specifications are used, neither for configuring model decisions nor for formalizing consistency, since model transformations are still required for model synchronization. There is no special DSL for configuring model decisions, since the model decisions for variation points of model co-evolution are diverse. Defining new operators with special model decisions would require to extend the DSL accordingly. Therefore, methodologists use a general-purpose programming language for configuring model decisions. This reduces the learning curve for methodologists, since no new DSL must be learned.

Configurations with imperative general-purpose Programming Languages

This design for model decisions is summarized in Definition 25:

Definition 25: Model Decision

Model decisions are means of unidirectional operators to control their changes on model level for model co-evolution. If methodologists apply these operators, they provide model configurations for each of the model decisions.

In order to ease the configuration work for methodologists again, MoConseMI provides *default configurations* for model decisions, which can be reused in recurring situations. In particular, complex configurations for recurring situations are made reusable with default configurations. As an example, →AddAssociation provides a default configuration for its model decision, which creates no links at all. By using the usual means of the general-purpose programming language like methods, methodologists could create also their own

reusable default Configurations for Model Decisions

reusable and project-specific configurations for model decisions.

custom Model Transformations with ⇄ChangeModel

Since model decisions are configured with imperative general-purpose programming languages, model decisions can be used to support methodologists with ⇄ChangeModel in order to configure custom model transformations scenarios, as it is required for customized handling of specific model co-evolution scenarios in Section 6.2.1☞193. →ChangeModel provides one model decision which allows methodologists to transform the model in arbitrary way, as long as the model remains conform to the metamodel, which is never changed by →ChangeModel. While ⇄ChangeModel provides the whole flexibility for changing models, it comes with the drawback, that the transformation for the inverse direction for ←ChangeModel must be specified accordingly and manually as well.

When configuring, how to transform the model in detail, methodologists can take not only the current model into account, but also the following additional information:

current Branch Differences ${}^{B}\Delta$

Current Branch Differences are model differences ${}^{B}\Delta$, which describe, how the current model which is the input for the unidirectional operator was changed, not by the previous operator, but compared to the previous version of this model. A special case for branch differences are user differences ${}^{User}\Delta$, which are applied by users to the model of a view, leading to an updated version of this model, which is given to the unidirectional operator as input. These branch differences are the input for change translation-based approaches, which work like "if model element x was changed, then change model element y, else do nothing" as reactions on model changes (Kramer, 2017, pp. 80–81). In the ongoing example, realizing Consistency Goal C 3☞77 for renamings of associations and their getter methods requires to know, if either the association or its getter was renamed in order to rename the counterpart to fix contradicting current names. Additionally, using ${}^{B}\Delta$ is motivated by the theoretical findings of Diskin, Xiong and Czarnecki (2011), that delta-based approaches reduce ambiguities compared to state-based approaches (Section 3.7☞146). Accordingly, the branch differences are used in the definition for round-trip engineering by Hettel, Lawley and Raymond (2008). Note, that after executing the current unidirectional operator, a new ${}^{B}\Delta$ must be calculated (requiring the design in Section 6.7.4☞236), which is given to the next unidirectional operator as input.

History Maps

History Maps store key-value pairs and are used by unidirectional operators and configurations for their model decisions. History maps allow to remember information like details of previous transformations or removed information and increases flexibility in general. History maps are deepened in Section 6.5.2☞214. History maps make operators state-full, while they are state-less without history maps. Already Finkelstein, Gabbay et al. (1993) motivate the use of historic data like changes in the past. Only few model transformation approaches using meta-data "*like authoring and versioning information*" (Macedo, Jorge and Cunha, 2017, p. 630) are found in a survey for approaches for ensuring model consistency.

History maps and current branch differences are also used to prevent possible information loss during the execution of operators, which is discussed in detail in Section 6.5.2☞214.

Summary

Summarizing model decisions, model co-evolution ensures the conformance of models to their changed metamodel in the first place. Since there are multiple possible solutions and since the selected solution depends on the semantics of the current model and is project-specific, the degrees of freedom during model co-evolution in general (demand 1) are exploited to ensure semantic consistency of models and to realize the project-specific consistency rules (demand 2). Model decisions allow methodologists to configure the behavior of operators accordingly. Such configurations take the current model, current branch differences and history maps into account and are written with imperative general-purpose programming languages in general. The configurations of model decisions of operators

is done by methodologists during the use case “specify Consistency” (Section 5.2.1 ☞171). Model differences for the representation of the model co-evolution are discussed in Section 6.7 ☞227.

6.4 Operator Combination

Up to now, single operators are designed as parts of model transformations for model synchronization (Section 5.1.6 ☞170). In order to describe the whole transformations between view(point)s and the SU(M)M, operators must be combined into a chain of operators. Since the data types for input and output are the same, i. e. one model and its metamodel, operators are chainable by design (Section 6.1.3 ☞189). The combination of operators into chains from view(point)s to the SU(M)M is done by methodologists during the use case “specify Consistency” (Section 5.2.1 ☞171), together with the selection of operators and the configuration of their metamodel decisions and model decisions. There are the following two *demands*, how to combine operators:

combine *multiple* Data Sources into the SU(M)M

1. In order to create the SU(M)M, it is not sufficient to create it from a single data source, but all reused data sources must be integrated into the SU(M)M, since it should contain all information of the project. Therefore, combining operators must allow to integrate multiple (meta)models into the SU(M)M.

combine Operators for new View(point)s

2. Additionally, the same operators should be combinable also to derive new view(point)s from the SU(M)M, according to Requirement R 3 (Define new View(point)s) ☞156.

Therefore, Section 6.4.1 investigates related approaches for combining parts of model transformations as preparation to design the combination of operators in Section 6.4.2 ☞204. Combining operators is used afterwards to integrate existing data sources into the SU(M)M in Section 6.4.3 ☞205 and to define new view(point)s for the SU(M)M in Section 6.4.4 ☞209. The result of these combinations is summarized in Section 6.4.5 ☞213.

6.4.1 Related Work

There are some related approaches for the reuse in model transformations by composing smaller transformations into bigger transformations or by splitting transformations into smaller parts. This section classifies the combinations of operators of MoConseMI into two classifications for model transformation reuse. By relating the design of MoConseMI to some more selected related approaches, this section will show, that reuse is possible with the general design of MoConseMI. More approaches for reusing model transformations can be found in Kusel, Schönböck et al. (2015) and Bruel, Combemale et al. (2020).

Classifications for Reuse of Model Transformations

According to the *first classification* for model transformation reuse by Kusel, Schönböck et al. (2015), reuse of model transformations can be distinguished into *intra-transformation reuse*, i. e. same parts are reused within the same transformation like module superimposition for ATL (Wagelaar, Van Der Straeten and Deridder, 2010), and *inter-transformation reuse*, i. e. same parts are reused in multiple different transformations (Kusel, Schönböck et al., 2015). In MoConseMI, operators should be reusable for multiple applications *and* also multiple times within one chain of operators between a particular view(point) and the SU(M)M. By configuring the metamodel decisions, the operators are bound to *concrete* parts of metamodels and have a *small* granularity in the classification of Kusel, Schönböck et al. (2015).

According to the *second classification* for model transformation reuse by Bruel, Combemale et al. (2020), MoConseMI provides *systematic* reuse by providing a library of operators, reuses operators as transformation *code* by *referencing* them in applications, since they

describe the whole transformation only *partially*, and transforms *in-place*. Each selected operator is applied once in a $1-1$ way by applying the metamodel changes to the current *metamodel*, defined in *intensional* way like with matching names. Since they are configured with metamodel decisions, they are defined in *explicit* way by methodologists. MoConseMI uses *operators*, which are *bidirectional*. Additionally, the operators provide *syntactic checking*, that they apply to the current metamodel, by *static checking*, as deepened in Section 6.5.2☞214.

Granularity

While small parts of transformations in form of operators should be chained, other research targets chaining of *complete* model transformations like Lúcio, Mustafiz et al. (2013), which allows no configurations as operators in MoConseMI do and decreases reuse due to the too coarse-grained granularity.

Etien, Muller et al. (2015) motivate to modularize model transformations and propose *localized transformations*, which change only small parts of metamodels and are chained to fulfill the complete model transformation. While their localized transformations are similar to operators here, their approach is out-place and not in-place. There are also languages dedicated to the specification of model transformation chains like UniTI (Vanhooff, Ayed et al., 2007).

Summary

Summarizing these related approaches, reuse of model transformations is required and is an active topic of research, in which operator-based approaches are used. Therefore, the design of operators in MoConseMI fits also to the design of model transformation reuse in research. Composing parts into a complete transformation is done here by *containment* (Kusel, Schönböck et al., 2015), i. e. the complete transformation consists of operators, which is designed in the following Section 6.4.2.

6.4.2 Chain of Operators

combine Operators as Chain

This section designs, how to combine operators, which represent reusable parts of transformations, into complete transformations. Two operators can be concatenated, since the output of the first operator is the input for the second operator. That is possible, since the signature of inputs and outputs of all operators is the same, since the operators work in-place. Therefore, methodologists can explicitly select operators and chain them in a self-defined order. This design is depicted in Figure 6.7.

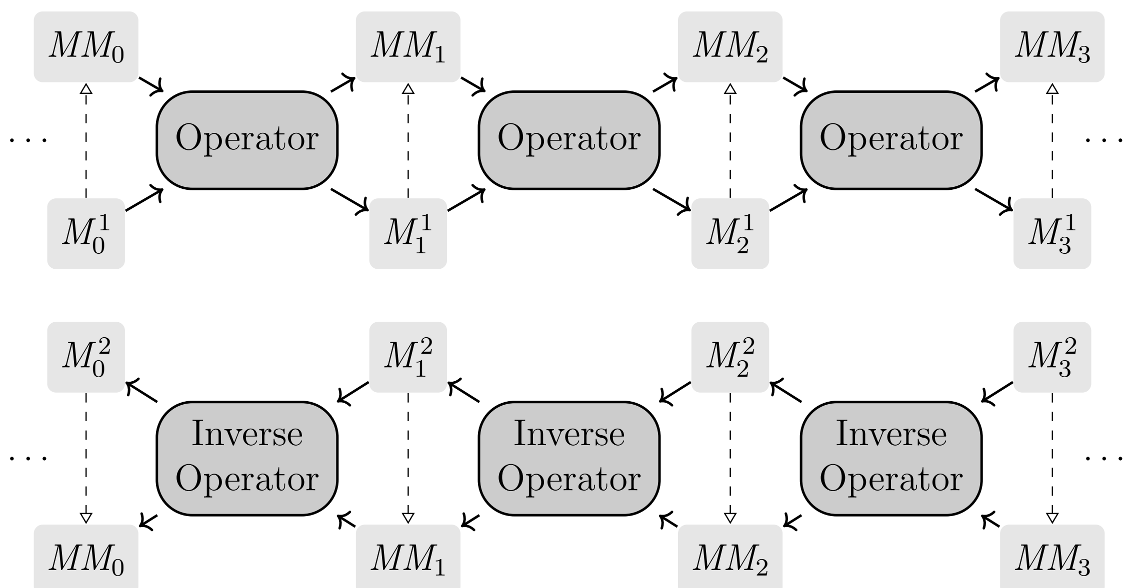


Figure 6.7: Chaining Operators according to the Operator Signature

By selecting operators with particular metamodel changes, methodologists explicitly decide the desired metamodel evolution. Since methodologists configure the metamodel decisions of operators according to the current metamodel, chaining configured operators specifies their order: As an example, if →AddAssociation is selected and configured to create a new association C between the classes A and B, renaming A to D with →RenameClassifier should be done after →AddAssociation, since →AddAssociation is configured with the name of A, not of D. As alternative, the renaming could be done before creating the association, but then →AddAssociation must be configured to create the new association between D and B. Therefore, the order of selected and configured operators is restricted. Even if there are operators within the same chain, which do not depend on each other in this way, their order should be constant and fixed in the way, as the methodologist designed it, otherwise readability and maintainability of operator chains is hampered.

fixed Order of configured Operators

Additionally, unidirectional operators define constraints for the configurations of metamodel decisions in order to ensure, that the operator fits to the current metamodel. As an example, →AddAssociation has constraints, that the two classes must exist, i. e. configuring →AddAssociation with A and B will result in error messages, if A or B do not exist.

Constraints for Metamodel Decisions

After designing the foundations for chaining operators, the next sections design, how multiple data sources can be integrated into the SU(M)M (Section 6.4.3) and how to define new view(point)s (Section 6.4.4☞209), both with chains of operators.

6.4.3 Integration of existing Data Sources

This section designs, how selecting and configuring operators and concatenating them into chains can be used to integrate data sources into the SU(M)M, according to consistency rules. If there is exactly one data source in the project, a single chain of operators according to Section 6.4.2☞204 is sufficient to transform it into the SU(M)M. If there is no data source, this special case is discussed in Section 13.3.2☞474 for using MoConseMI without reuse. This section focuses on the case with two or more data sources to reuse.

If there are two or more data sources to reuse, two independent operator chains could be created, which even might result in equal metamodels for the SUMM, but there are two independent models for the SUM, since the first one contains only the information of the first data source and the second one contains only the information of the second data source. Additionally, it is not possible to link these two models with each other, since they are created by independent chains of operators. Therefore, a special solution is required to enable having information of two models of two data sources within the same chain of operators.

Combination of Information from two Models required

Since operators are designed to work only with one (meta)model (Section 6.1.3☞189), information stemming from two different data sources in form of two models must be combined into one model. This is realized with a special operator called →CombineDataSources, specially designed and only used for this situation: It is the only unidirectional operator which takes *two* (meta)models as input, technically combines them in-place into *one* (meta)model and returns this (meta)model. With this design, it combines two chains of operators, starting at two data sources, into one chain of operators towards the SU(M)M. →CombineDataSources is combined with ←SeparateDataSources into ⇄CombineSeparateDataSources.

special Operator ⇄CombineSeparateDataSources

→CombineDataSources technically combines two (meta)models by moving all their contained elements into one (meta)model. The elements of the two (meta)models are still completely independent of each other (they are only with the same shell), which is exploited by the inverse ←SeparateDataSources for separating the (meta)model into two (meta)models again. This design is also the usual strategy to enable tools without explicit

technical Combination of two (Meta)Models

support for multiple models (Macedo, Jorge and Cunha, 2017, p. 622), as introduced in Section 3.2.5 ☞108. The combination is done in explicit way "by copy"[13] and not by linking e. g. with proxies as provided by EMF or investigated by Amalio, de Lara and Guerra (2015). Just after this combination, the result can be seen as "multi-model" in the sense of Diskin, König and Lawford (2018). This operator →CombineDataSources realizes technical composition of models and metamodels as technical precondition for doing the semantic composition (Kienzle, Mussbacher et al., 2019) afterwards with subsequent operators in order to realize consistency rules and to reduce dependencies for a more essential SUM (Section 5.1.3 ☞167).

Tree of Data Sources

Since →CombineDataSources combines exactly two (meta)models, in order to combine $n > 0$ data sources, this operator must be applied $n - 1$-times. The resulting chains of operators form a "tree of data sources", allowing arbitrary arrangements: After combining the first two data sources, the third data source could be combined, then the fourth data source and so on. A possible alternative is to combine the first and the second data source with each other, then the third and fourth data source and afterwards the first two data sources are combined with the last two data sources. Additionally, both kinds can be mixed.

Related Work

In the area of data base integration, the first kind is called "binary-ladder" and the second kind "binary-balanced" (Batini, Lenzerini and Navathe, 1986, p. 343). The strategy to extend metamodels in step-wise way with more concepts and other viewpoints is also used by Knapp and Mossakowski (2018, p. 43f) for UML, even since there is already a somehow integrated SUMM. In the same step-wise way, Misbhauddin and Alshayeb (2019) integrate viewpoints for UML and for OCL. These related approaches show, that the design for integrating data sources into a SU(M)M is a known and working solution.

combining Data Sources with GraphQL

GraphQL as introduced in Section 6.3.1 ☞199 allows to combine multiple GraphQL schemata into one single schema, which is called "stitching"[14]: It allows to rename existing types, to merge types with the same name automatically and to transform single schemata. In MoConseMI in contrast, transformations, renamings and merges are realized by single operators. Therefore, the (technical) combination of two data sources can be eased with a single operator in MoConseMI, since possible conflicts are solved by previous operators and the integration is done by subsequent operators. Additionally, the operators couple transformations in forward and backward direction, while GraphQL requires completely independent implementations for reading and writing data.

arbitrary, but fixed Order of Data Sources

In principle, each order for the combination of data sources is possible, as described above. But after the methodologist selected and explicitly specified an order, this order is fixed, since the selected operators and their configurations depend on each other, as discussed in Section 6.4.2 ☞204. This can be found also for merging $n \geq 2$ models, where the order matters and leads to different results (Rubin and Chechik, 2013; Boubakir and Chaoui, 2016).

An example for its application is given in the ongoing example now. Walter and Ebert (2009) show another example for chains of step-wise transformations and integrations of three models in the domains of ontologies, DSLs and feature modeling.

Ongoing Example, Part 21: **Integration of Data Sources** ← List →

This box roughly sketches the integration of the three data sources into the SU(M)M. The configured operators are depicted in Figure 6.8 ☞207.

[13] In out-place transformations, elements are copied into the same (meta)model, while the term "moving" is better suited for in-place transformations.

[14] https://www.graphql-tools.com/docs/schema-stitching/stitch-combining-schemas, https://hasura.io/blog/the-ultimate-guide-to-schema-stitching-in-graphql-f30178ac0072/

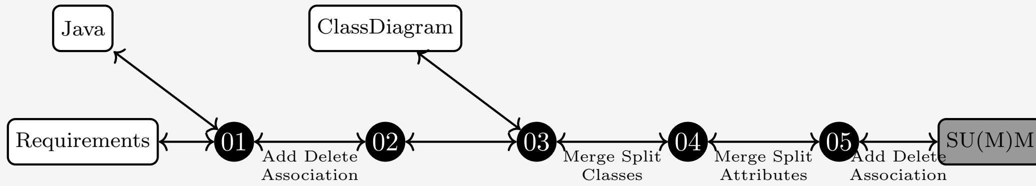


Figure 6.8: Operators for the Integration of Data Sources into the SU(M)M

Starting point are the Requirements and the Java source code, which are technically combined with ⇄CombineSeparateDataSources. Afterwards, ⇄AddDeleteAssociation 01⟷02 realizes C 1, since the new association, as depicted in Figure 6.9, enables users to create traceability links in the model.

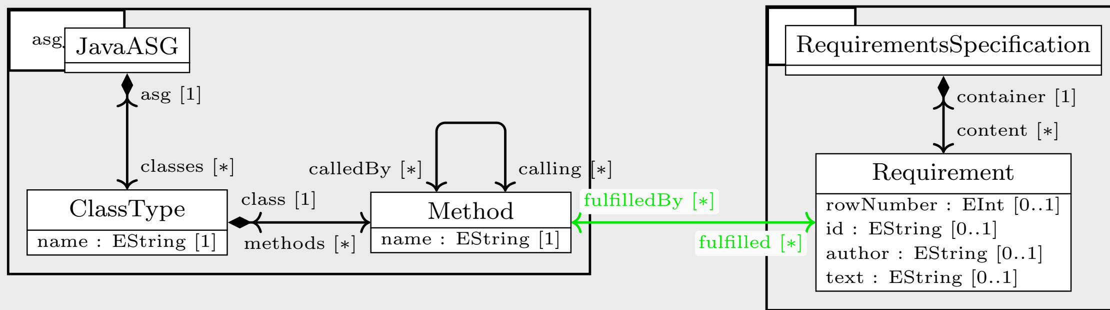


Figure 6.9: Metamodel Changes from 01 to 02

After combining ClassDiagram with the current (meta)model 02 by ⇄CombineSeparateDataSources, the operators ⇄MergeSplitClasses (03↔04) and ⇄MergeSplitAttributes (04↔05) unify the duplicate representations for classes in Java and ClassDiagram for ensuring C 2, as depicted in Figure 6.10 ☞ 208. →MergeClasses merges `umlclasses.Class` into `asgjava.ClassType`, while →MergeAttributes merge the duplicate attributes `asgjava.ClassType.className` and `asgjava.ClassType.name`. The corresponding merges of objects and their slots ensures, that this redundancy is not contained in the SUM anymore. Since the views are updated via the SUM, consistent models are derived from the SUM by design.

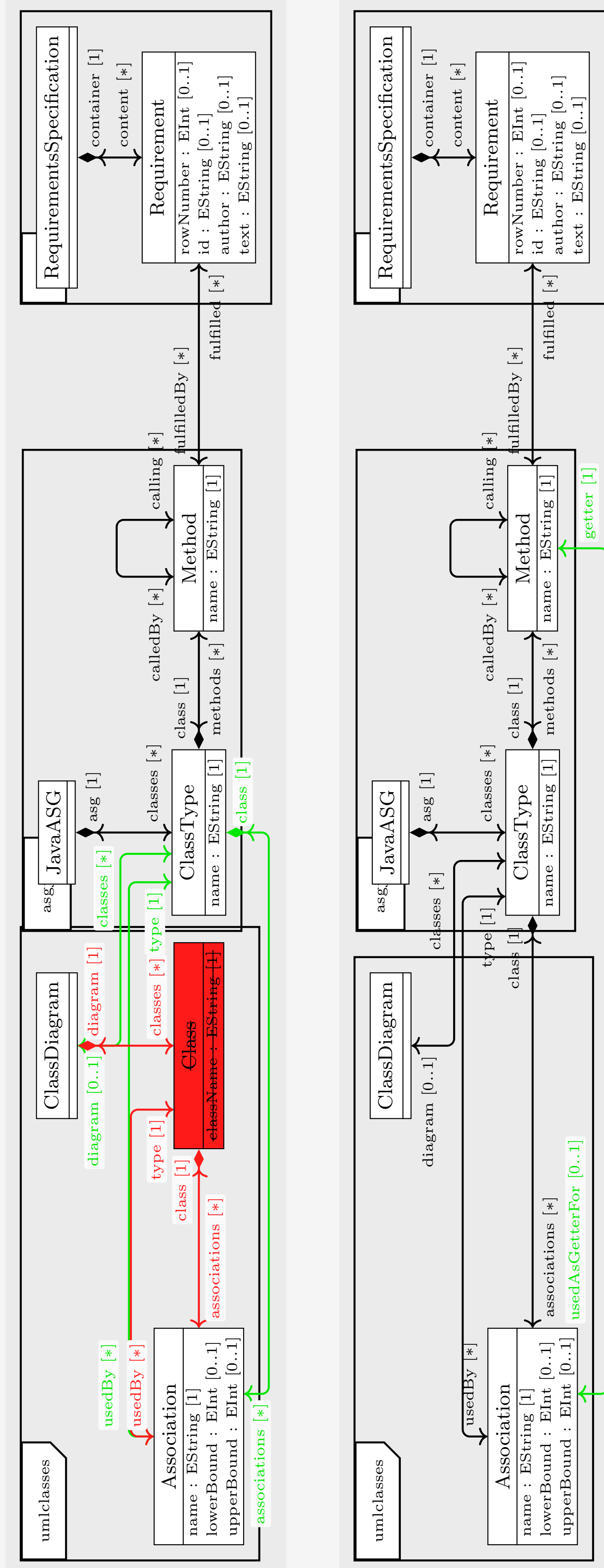


Figure 6.10: Metamodel Changes from 03 to 05 (left/top) and 05 to SUMM (right/bottom)

Finally, ⇄AddDeleteAssociation (05↔SUMM) realizes C 3 by introducing a new association in order to indicate that one `Method` for each `Association`, which serves as getter for the association, as depicted in Figure 6.10☞208. Making this relationship explicit, allows to detect diverging names of associations and their getters and to handle them via renamings configurations in →AddAssociation.

6.4.4 Definition of new View(point)s

new Views represent existing Information differently

This section designs, how chains of operators can be used to define new view(point)s on top of the SU(M)M. The purpose of new view(point)s is to represent only selected parts of the system under development, tailored to the concerns of users (Section 1.2.3☞39). Since the system under development is represented as a whole by the SUM conforming to the SUMM, the SU(M)M is the natural starting point to derive new view(point)s, which present no new information, but already existing information in a different way.

Operator Chain for defining new View(point)s

Since a new view(point) consists of a model and its metamodel, both must be created, since they do not exist initially. Therefore, operators can be reused also for this purpose, since they jointly transform models and metamodels. Chaining bidirectional operators together allows to describe the complete way from the SU(M)M to the new view(point) and to restructure the concepts of the SUMM and information of the SUM.

special Operator ⇄SubSet

Since the new view contains only information which stems from the SUM and contains only some parts of this information, the operator chain reduces the available information during the way from the SUM to the new view, according to the asymmetric case (Figure 3.5☞104). This is supported with the special operator ⇄SubSet, consisting of →SubSetFilter and ←SubSetRecreate: →SubSetFilter removes some elements in the metamodel (selected by metamodel decisions with configurations for a metamodel decision) and all conforming elements in the model. Section 6.5.2☞214 explains the foundations, how ←SubSetRecreate restores the removed information.

define new View(point)s on top of any (Meta)Model

Usually, the starting point for defining new view(point)s is the SU(M)M, since it allows to reuse all represented concepts and information of the system under development in an integrated and improved way, since internal dependencies are reduced during the integration of data sources (Section 6.4.3☞205) according to Section 5.1.3☞167. But in general, it is possible to use any (meta)model as starting point, including the data sources and intermediate (meta)models in operator chains. Therefore, it is also possible to start at another new view(point) or at an intermediate (meta)model in its operator chain.

An example for a definition of a new view(point) is given for the ongoing example:

Ongoing Example, Part 22: **Definition of the new View(point)** ← List →

This box roughly sketches the definition of the new view(point) with a chain of configured operators, starting at the SU(M)M, as visualized in Figure 6.11.

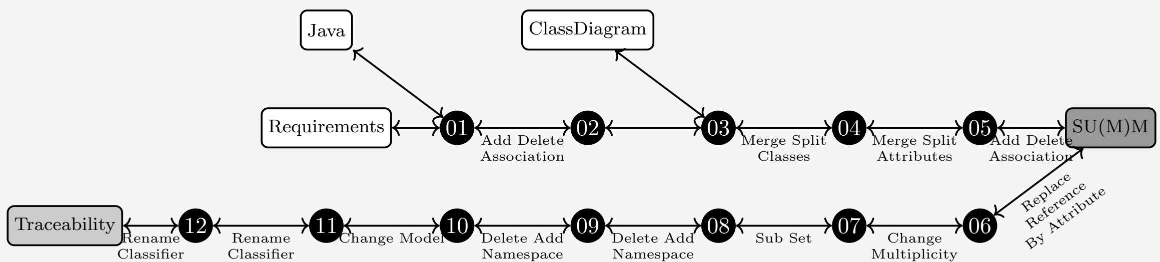


Figure 6.11: All Operators: Integration and Definition of new View(point)s

⇄ReplaceReferenceByAttribute (SUMM→06) replaces the traceability links between requirements and methods, so that requirements store the names of their fulfilling methods as String values, as depicted in Figure 6.12☞211.

⇄ChangeMultiplicity (06→07) switches the multi-value attribute to a single-value attribute and concatenates the names of methods with commas as separators, since the final CSV format supports only one value for each table cell, as depicted in Figure 6.12☞211.

Since the parts for ClassDiagram and Java are no longer required in the new view(point), all their corresponding metamodel and model elements are removed with the operators ⇄SubSet and twice ⇄DeleteAddNamespace (07→10), as depicted in Figure 6.12☞211.

⇄ChangeModel (10→11) adapts the row numbers for Excel, since they stem from CSV originally, without changes in the metamodel.

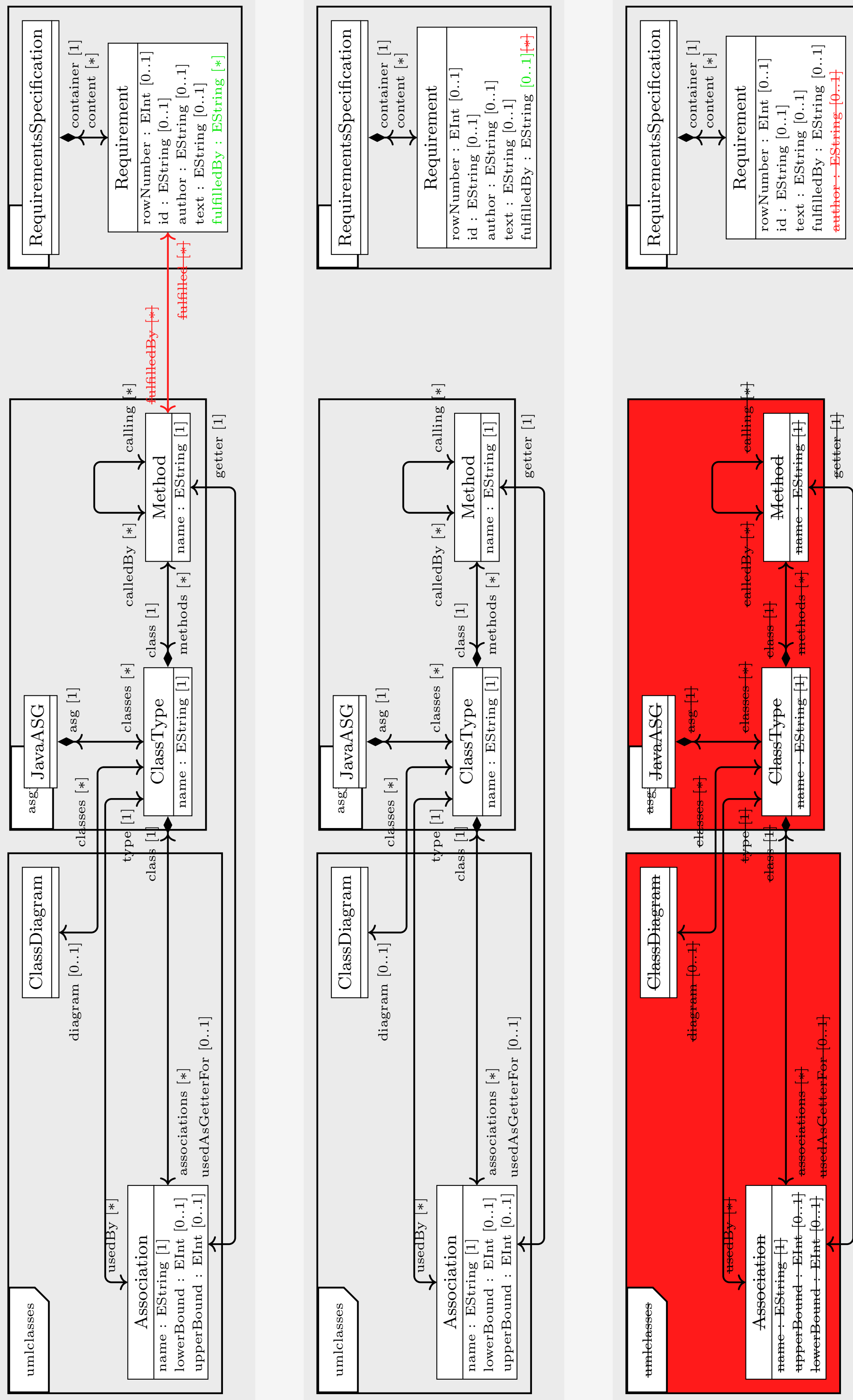


Figure 6.12: Metamodel Changes from SUMM to 06 (left/top), from 06 to 07 (middle) and from 07 to 10 (right/bottom)

Finally, to provide known terminology to the manager, some elements are renamed by applying ⇄RenameClassifier twice (11→Traceability), as depicted in Figure 6.13.

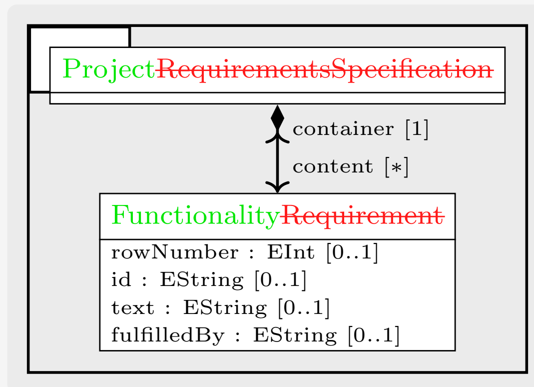


Figure 6.13: Metamodel Changes from 11 to Traceability

The resulting metamodel for the new viewpoint Traceability is shown in Figure 6.14: Requirements are depicted by `Functionality` with the fulfilled methods as String representation in the attribute `fulfilledBy`.

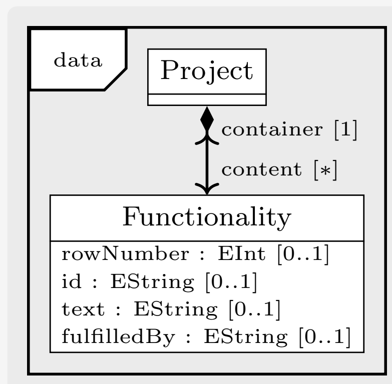


Figure 6.14: Metamodel for the new viewpoint Traceability

The model for the new view Traceability is shown in Figure 6.15, containing the two functionalities `r1` and `r2`. Since the initial data have no relationships between methods and requirements, the slots for `fulfilledBy` are still empty.

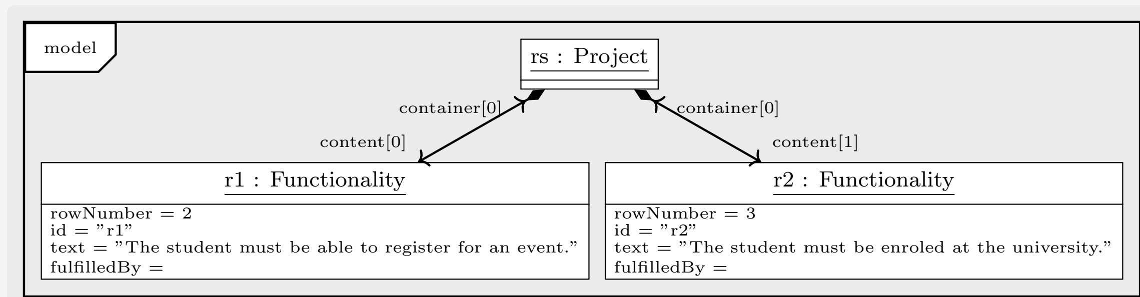


Figure 6.15: Model for the new view Traceability

6.4.5 Final Result: Tree

The design of operators (Section 6.1.3☞189) enables to concatenate operators into chains of operators. The special operator ⇄CombineSeparateDataSources allows to combine two (meta)models into one (meta)model and enables the integration of data sources into the SU(M)M (demand 1, in Section 6.4.3☞205). This design prevents independent operator chains for each data source and leads to a tree. Chains of operators can also be used to define new view(point)s from the SU(M)M (demand 2), possibly supported by ⇄SubSet (Section 6.4.4☞209). Since new view(point)s are defined with additional operator chains, which start at an arbitrary (meta)model (but usually at the SU(M)M) and end at the new view(point), the "tree of data sources" is extended to a "tree of view(point)s" including data sources and new view(point)s. This design is summarized in Definition 26:

Tree of configured Operators

Definition 26: Orchestration

The orchestration consists of the whole tree of selected and configured bidirectional operators as edges, with the SU(M)M as root and the data sources and new view(point)s as leaves. The order of all operators is defined by the methodologist and remains stable.

This tree of configured operators is important for the execution of operators, as designed in the following Section 6.5: Since the tree topology ensures, that there is only one possible path between two nodes (when traversing each edge not more than once), there is only one transformation path between these nodes and no contractions between alternative transformation specifications are possible. In particular, no duplications of model elements due to multiple alternative transformation executions can occur (Klare, Syma et al., 2019). While MoConseMI has a *tree* to organize data sources, which is *directed*, since ⇄AddDeleteAssociation and ⇄DeleteAddAssociation are different, synthetic approaches usually have *directed graphs* of model transformations between them. Another possible topology is a *directed acyclic graph* (DAG), as used by model weaving for depending parts of different models (Kienzle, Al Abed and Jacques, 2009).

Tree prevents contradicting Execution Paths

6.5 Operator Execution

Up to now, the operators, their details and their combination are designed for the use case of the methodologist to specify consistency at *development time* (Section 5.2.1☞171). This section focuses on the *execution of operators* in order to automatically fix inconsistencies for users at *runtime* (Section 5.2.2☞173). Precondition for this use case is to have the orchestration, i. e. the tree of configured operators, which is executed in this use case.

Execution of Operators for fixing Inconsistencies

Since the operators are *in-place* transformations (Section 6.1.3☞189), the main idea for change propagation in MoConseMI is to transform the currently changed model (and its metamodel) along the tree of configured operators to all other models (including the SUM) to update them in-place. During these in-place transformations, elements which are changed by users are propagated to the other models. Since the model transformations of the selected operators are configured in order to fulfill consistency rules, additional model elements might be updated and propagated to all other models.

There are the following two *demands* on executing the tree of operators:

1. Since the chains of operators which start at the SU(M)M and end at the view(point)s conform to the asymmetric case, there is potential for *information loss*: When transforming the current model in-place from the SUM to the view, the amount of information is reduced. But this reduced information must be restored when transforming from the view back to the SUM.

prevent Information Loss

depending Consistency Goals

2. Since consistency goals might depend on each other, the one-time execution of operators might not be sufficient, if the operators which realize the depending consistency goals are placed at different locations in the orchestration.

The second demand is investigated in more detail in related work (Section 6.5.1). The execution of single operators as partial transformations is designed in Section 6.5.2, before designing the execution of the whole model synchronization process in Section 6.5.3☞217. This execution is used for the initialization of the SUM (Section 6.5.4☞219) as well as for the ongoing change propagation (Section 6.5.5☞220).

6.5.1 Related Work

Dependencies and multiple Execution Iterations

Persson, Torngren et al. (2013, p. 5) already predicted the need for multiple iterations for the execution in theory, before reaching a fix-point. Conceptual reasons for that are depending consistency goals and consistency rules (Kramer, 2017, p. 65), concrete data depending on each other, e. g. call-hierarchies of methods, and inconsistency fixes which lead to new inconsistencies (Dam, Egyed et al., 2016, p. 138), which might be fixed already before (Mens and Van Der Straeten, 2007). If these reasons matter, might depend also on the concrete configuration of operators, in terms of their order and their configured model decisions.

Fix-Point

Executing chains of operators multiple times requires also to terminate the execution: Defining a fix-point depends on the understanding, that the models are not changed anymore, if they are (already) consistent to each other (now): Formally, this correspond to the hippocraticness property of BX (Stevens, 2010, p. 14), which is also formalized for symmetric delta-lenses (Diskin, Xiong et al., 2011, p. 314).

6.5.2 Executing single unidirectional Operators

unidirectional Operators transform the current Model and its Metamodel *in-place*

This section designs the execution of *single unidirectional operators* only, while the execution of bidirectional operators is designed in Section 6.5.3☞217. A single unidirectional operator is executed by providing the current model and its metamodel to the operator as input. Usually, the operator executes a small number of changes within the metamodel and more changes in the model to ensure its conformance to the changed metamodel. The details of changes are determined by the implementation of the current operator, depending on the configurations for possible metamodel decisions and model decisions. Note, that metamodel changes and model changes can occur in any order, even mixed. There are operators, which change only the metamodel, e. g. →RENAMECLASSIFIER, or which change only the model, e. g. →CHANGEMODEL. The (changed) model and its (changed) metamodel are provided by the operator as output. This conforms to Figure 6.2☞190 with emphasizing, that the operator transforms *in-place*.

internal Validations

Before executing a unidirectional operator, its *constraints* for the configurations of metamodel decisions are checked in order to make sure, that the operator is applicable, since it is properly configured into the operator chain (Section 6.4.2☞204). Additionally, the execution of a single unidirectional operator is *validated* for internal quality assurance. These validations cross-check, that the implementation of operators by platform specialists, the configurations of operators by methodologists and the implementation of the framework for the execution fit together and conform to the design, described in Chapter 6☞185:

- It is checked, that the metamodel is still a valid metamodel after executing the operator, according to the design of EMF.
- It is checked, that the model is still a valid model after executing the operator, according to the design of EMF and conforming to its metamodel.

- It is checked, that the metamodel changes of a configured unidirectional operator and its configured inverse unidirectional operator are inverse. For that, it is necessary to record the changes done in the metamodel during the operator execution and to remember the recorded metamodel changes, which is deepened in Section 6.7☞227.

History Maps

History maps, as introduced and motivated in Section 6.3.2☞200, allow concrete model decisions to read and write key-value pairs in order to remember them over multiple executions of the same operator: The key-value pairs of history maps use simple text as key, since they allow a speaking designation of the values, and arbitrary serializable content for values, since they maximize flexibility for methodologists (including other maps as content) and ensure, that the values are serializable for shutting down and restarting the framework. The history maps remain the same for each execution of the bidirectional operator, as indicated in Figure 6.16: Each selected and configured bidirectional operator in an orchestration has its own history maps and they are given to each execution of the two unidirectional operators of this bidirectional operator. This design allows to remember important information of the current execution of a unidirectional operator for following executions. There are two different history maps in order to support both stakeholders, the methodologist, who configures model decisions ("History Map Decisions"), and the platform specialist, who implements operators in general ("History Map Operator"), since both stakeholders might need history maps, but should not interfere with each other.

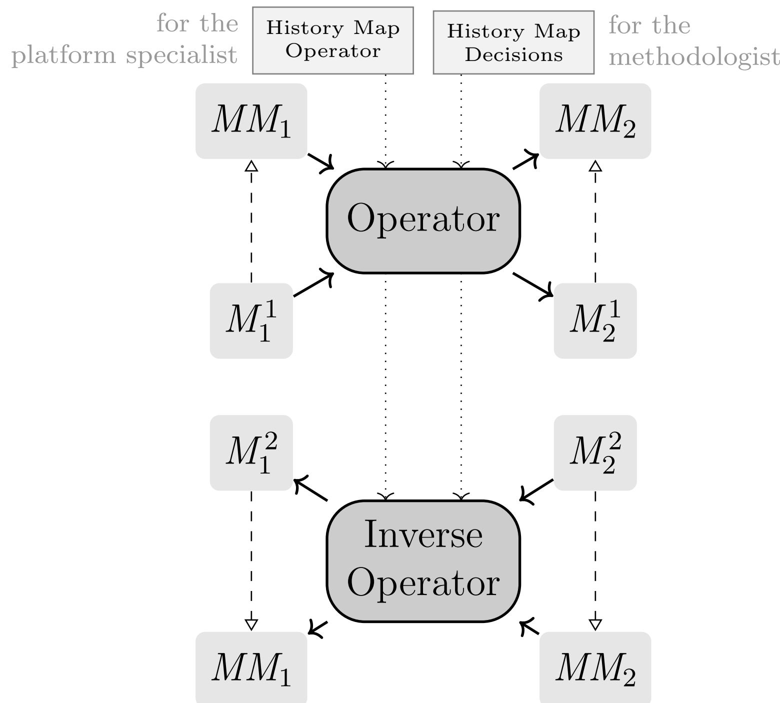


Figure 6.16: Execution Information within the Operator Signature

Branch Differences $^{B}\Delta$

The current *branch differences*, as motivated in Section 6.3.2☞200, allow to change the model depending on the current (user) changes in the current model. When executing the inverse unidirectional operator as visualized in Figure 6.17☞216, the current branch differences are depicted as $^{B}\Delta_2^2$ (abbreviated as "Branch Diff") and are available as read-only input for model decisions. These *branch differences* describe the changes from the model M_2^1 to the model M_2^2, i. e. the model at the same position in the orchestration at different points in execution times. In particular, branch differences are no *operator differences*, which are directly caused by operators, e. g. $^{\overleftarrow{Op}}_{M}\Delta_2^2$ is executed by the inverse unidirectional operator and describes its in-place model changes from M_2^2 to M_1^2. These

changes in the model are directly executed by unidirectional operators and are recorded as ${}^{Op}_{M}\Delta$ by the framework. The following branch differences ${}^{\mathrm{B}}\Delta^2_1$, given as input to the following next operator, are not provided by the unidirectional operator, but are calculated by the framework, as designed in Section 6.7.4 ☞236.

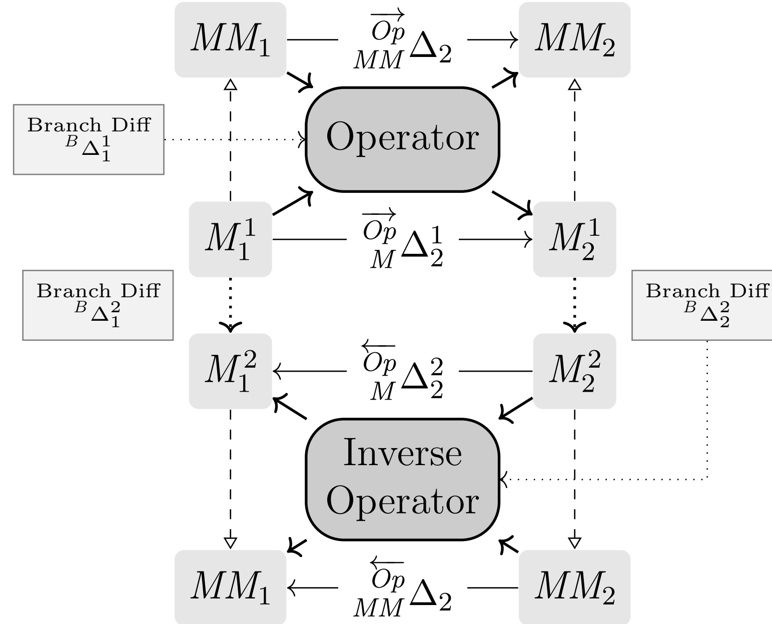


Figure 6.17: Model Differences and Metamodel Differences within the Operator Signature

prevent Information Loss

According to demand 1, in-place executions of unidirectional operators in the outgoing direction, i. e. from the SUM to views, reduce the information in the current model. In-place executions of unidirectional operators in the incoming direction, i. e. from views to the SUM, have to restore these information in the current model, according to the asymmetric case. Therefore, such information must be stored somehow between consecutive executions of unidirectional operators belonging to the same configured bidirectional operator. According to the principle of locality, removed information is stored at the particular configured bidirectional operator, since it manages the executed two unidirectional operators which are inverse to each other. This design is in line with other researchers: Kleppe, Warmer and Bast (2003, pp. 80–82) advocate to store additionally required information directly at the model transformation, which is a concrete execution conforming to the model transformation definition. In similar way, Eramo, Pierantonio and Tucci (2018, p. 38) remember deleted elements in trace information of the out-place JTL approach for BX. Hidaka, Tisi et al. (2016, p. 915f) call lost information "complement" and classify possibilities to store lost information within the transformed model or outside, as it is used here. MoConseMI provides three strategies to remember information which is currently removed in order to update views and which will be restored later to update the SUM (Meier, Kateule and Winter, 2020):

use History Maps

1. The history maps, as designed above, can be used to store information before removing them and to restore them later from history maps. This strategy provides full flexibility for methodologists when configuring model decisions of arbitrary operators.

apply inverted ${}^{Op}_{M}\Delta$

2. Since models are changed in-place by unidirectional operators, these model changes can be recorded and remembered: These model changes are exploited to restore removed information by remembering removed information as model differences and by restoring information by applying the inverted model differences. As an example for ⇄AddDeleteAssociation, →AddAssociation enables to create new links in

the model (but is configured to create no new links itself), which might be introduced manually by new views, but these links are deleted when executing ←DeleteAssociation: When executing →AddAssociation the next time, the removed links cannot be recreated by →AddAssociation directly, since it does not know which links were created manually by users. Instead, the links are remembered in the $\overleftarrow{Op}_{M}\Delta$ of ←DeleteAssociation and can be applied in inverted way to restore the links. This strategy is also used by ⇄SubSet, since it removes multiple metamodel elements and all conforming model elements in the main direction and recreates the metamodel elements and restores the removed model elements in the inverse direction. This strategy is provided in generic form by the framework and can be requested by methodologists during the configuration of bidirectional operators. This strategy works like a post-treatment for the execution of unidirectional operators and is hidden to the unidirectional operators. This design requires to record and provide changes in models as explicit model differences which must be invertible, as designed in Section 6.7☞227.

3. This strategy is a special case to support ⇄CombineSeparateDataSources: When executing the inverse unidirectional ←SeparateDataSources, the current model is split into two models, one for each of the two data sources. One model is provided by the operator as output, the other model is remembered in-memory inside ⇄CombineSeparateDataSources as complete model. Later, →CombineDataSources takes this remembered model and combines it again with the current model.

⇄CombineSeparateDataSources remembers the split Model

In general, this information preservation is also a challenge for related BX approaches. As an example, Ehrig, Ehrig et al. (2014) investigate the conditions for TGGs, so that their "*corresponding forward and backward transformations are inverse to each other in the sense of information preservation*" (Ehrig, Ehrig et al., 2014, p. 72). Using the presented strategies is not required for bijective settings, since there is no information loss. While MoConseMI focuses on the asymmetric case, the symmetric case could be supported with these strategies and operators as well in principle. Summarizing, the three strategies help to prevent information loss (demand 1).

6.5.3 Execution Loop

This section designs, how the tree of operators is executed in order to propagate changes from one view into all other views. First, the execution of a single chain of operators between the SUM and one view is designed. Second, the change propagation between the SUM and all views is designed. Third, this execution is used for the initialization of the SUM in Section 6.5.4☞219 as well as for the ongoing change propagation in Section 6.5.5☞220.

To design the *execution of a single operator chain*, the chain is considered from the SU(M)M in the direction to the view(point): If the SUM is changed (${}^{B}\Delta_{\text{SUM}} \neq \varnothing$), these changes should be propagated to the view, which is done by executing the chain of operators, operator by operator. Since this chain consists of *bidirectional* operators, when executing a bidirectional operator, its unidirectional operator is executed whose direction points from the SUM to the view. For the transformation in the other direction, i. e. from the view to the SUM, the other unidirectional operator is executed. The identified unidirectional operator is executed according to the design in Section 6.5.2☞214. In this way, unidirectional operators are executed one after another according to the chain of bidirectional operators, starting with the first operator connected with the SU(M)M, then its neighbored operator, and so on, until the view(point) is reached. Since the unidirectional operators transform the model in-place, the model of the view is updated now, since it also contains a subset of the changed elements of the SUM (${}^{B}\Delta_{\text{View}}$). The subset of changed information is coming

execute single Operator Chain

from the asymmetric case. If this subset is empty, then the changes in the SUM do not affect the current view. In order to improve the execution performance, the execution of the operator chain is stopped, when there are no branch differences anymore. Again, the details of the calculation of the current ${}^{B}\Delta$ are designed in Section 6.7.4☞236.

execute Operator Chain in inverse Direction

After stopping the execution of the current operator chain, since the view is reached or the current ${}^{B}\Delta$ is empty, the operator chain is executed in *inverse direction back to the SUM* for two reasons: First, the configured model decisions of the unidirectional operators for the inverse direction might fix some inconsistencies and therefore need to be executed. Second, the SUM must be reached again as starting point to propagate changes also to other views by executing other operator chains. For the transformation in the other direction from the view to the SUM, the chain of operators is executed in inverse order and with the other unidirectional operators, until the SUM is reached.

execute the Operator Chain again → Execution Loop

If the executed operators change the model (${}^{B}\Delta_{B} \neq \varnothing$) and these model changes are relevant for the SUM, the SUM is changed (again) with a non-empty ${}^{B}\Delta_{SUM}$. Since these changes might be occurred by unidirectional operators between the view and the SUM, these changes are not yet propagated to the view. Therefore, the current operator chain is executed (again) from the SUM to the new view in order to propagate the new changes in the SUM to the view. This leads to the looped execution of the operator chain between the SUM and the view. This execution loop ends, if neither the SUM nor the view are changed anymore, i. e. the current ${}^{B}\Delta_{SUM}$ and ${}^{B}\Delta_{View}$ are both empty, which defines the fix-point of the execution loop.

execute whole Operator Tree

After updating one view as described up to now, all other views have to be updated as well: Therefore, the other views are updated in the same way as the first view is updated. Regarding the order of views to update, data sources are updated before new views and the most recently integrated views are updated first. If the SUM is changed during updating a view, all other views must be updated afterwards again in order to propagate the changes in the SUM to these views. Summarizing, the execution loop ends, if neither the SUM nor all the views are changed anymore, i. e. the current ${}^{B}\Delta_{SUM}$ and the ${}^{B}\Delta$ of all views are empty, which defines the fix-point of the execution loop. This design fulfills demand 2, since fixing inconsistencies is spread to configured model decisions of multiple unidirectional operators in both directions, in particular, operators fulfilling depending consistency rules could be spread elsewhere in the orchestration. These practical reasons for multiple execution iterations in MoConseMI complement the theoretical motivations in Section 6.5.1☞214.

empty ${}^{B}\Delta$ is the Fix-Point

With this design, termination of the execution loop cannot be guaranteed, as discussed in Section 14.3.1.1☞491, but is flexible and independent of the configured operators. Additionally, the execution order must not be specified by methodologists, but is provided by MoConseMI, while in related approaches the execution order can be implicitly (often in declarative transformations) or explicitly specified (Eramo, Marinelli and Pierantonio, 2014). To detect, if the fix-point of the execution is reached, the branch differences ${}^{B}\Delta$ must be available, which motivates their calculation in Section 6.7.4☞236.

calculate Execution Differences ${}^{E}\Delta$

After finishing the execution loop, the final changes for all views are calculated as *execution differences* ${}^{E}\Delta$: If the current view is not updated during the execution loop, its ${}^{E}\Delta$ is empty. If the current view is updated once, that ${}^{B}\Delta$ is also the ${}^{E}\Delta$. If the current view is updated twice or more, ${}^{E}\Delta$ is the optimized concatenation of all ${}^{B}\Delta$. If the current view is the view which is changed manually by users, the resulting ${}^{User}\Delta$ is the first ${}^{B}\Delta$ to take into account. After calculation, the ${}^{E}\Delta$ are forwarded to the adapters of views in order to update also the concrete renderings of the views with updated models. Note the special case, that the user differences ${}^{User}\Delta$ are already known at the adapter of the view which was changed directly and manually by the user. Therefore, the execution differences ${}^{E}\Delta$ without the ${}^{User}\Delta$ must be given as update to this adapter.

Future Work: **Improved Execution Order**

This design of the execution loop could be improved, in particular, since the design works also for independent operator chains, i. e. one independent operator chain for each view, and does not exploit the tree topology of operators. There following ideas might improve the execution order:

- When transforming the model from the view back to the SUM, this transformation could be stopped, if there are no new changes for the SUM.
- The designed order of views to update could be improved, e. g. operator chains for views which often introduce new changes could be executed first, since the generated follow-up changes could be propagated all together to the other views.
- The details of the configured operators could be analyzed in order to calculate improved execution orders, in particular, knowledge about operators which never fix inconsistencies but do only refactorings could save executions which are only done to be on the safe side.
- The update of new read-only views could be improved by updating them as last views and reaching them only once, if the operators for the new view do not update the SUM.

In general, these first ideas show, that more experiments and theoretical investigations are possible for improving the execution order within the execution loop in terms of performance. Nevertheless, the applications of MoConseMI in Part IV☞283 show, that the change propagation works and leads to the expected results in terms of execution differences $^{E}\Delta$.

6.5.4 Initial Execution

This sections describes the additional use case to initialize the SU(M) and new view(point)s and to fix initial inconsistencies within data sources (Section 5.2.3☞176). Since initially only the data sources exist with the (meta)models, there is one first special execution of the configured operator chains starting at the data sources to the SU(M)M: This transformation is not triggered by users, in particular, there are no user differences $^{User}\Delta$, but it is started by methodologists after finishing the orchestration. *All* operator chains starting at data sources are executed into the direction of the (not yet existing) SU(M)M. When reaching a ⇄CombineSeparateDataSources, its execution combines the (meta)models of two data sources into one (meta)model. Afterwards, the subsequent operators are executed. At the end of this execution, all (meta)models of all data sources are integrated into the SUMM and the conforming initial SUM. There are no branch differences $^{B}\Delta^{0}$, since they represent the differences between the current models, which are executed, and the previous versions of these models, which do not exist, since this is the first execution.

integrate (Meta)Mmodels of Data Sources into the initial SU(M)M

After the creation of the SU(M)M in this way, the execution loop as designed in Section 6.5.3☞217 is started. Since there are no current $^{B}\Delta$, the execution is forced to reach all view(point)s, i. e. all data sources and all new view(point)s. This procedure creates the metamodels and conforming initial models for the new view(point)s. The resulting $^{E}\Delta$ for data sources represent fixes for initial inconsistencies in the reused models. Additionally, the two methods of bidirectional operators are called directly before and after the first execution of one of its unidirectional operators. These calls can be used by bidirectional operators to automatically derive the configurations for the metamodel decisions of the inverse unidirectional operator (Section 6.2.2☞196). After the execution loop reached its

fix initial Inconsistencies

fix-point, MoConseMI is ready for users, who change single views and are supported with automated change propagation, as described in Section 6.5.5.

6.5.5 Ongoing Change Propagation

propagate $^{\text{User}}\Delta$ into all Views

This section describes the use case for fixing inconsistencies automatically as described in Section 5.2.2☞173. A user gets the desired view, changes its model resulting in a non-empty $^{\text{User}}\Delta_{\text{View}}$ and triggers MoConseMI to automatically propagate the changes into all related views. For that, the execution loop as designed in Section 6.5.3☞217 is started and executed. After finishing the execution loop, the $^{\text{E}}\Delta$ is calculated for each view including the SUM and the changed view, which might be empty for those views which have no semantic dependencies with the changed view. Note, that the $^{\text{User}}\Delta$ could be changed (i. e. amended) by the execution loop, which is useful in some cases (Diskin, König and Lawford, 2018, p. 25f), e. g. to revert user changes of read-only information in new views. In theory, the model of any view including the SUM, as proposed by Bork and Sinz (2013, p. 30), and even any intermediate model within the orchestration could be changed by users. In practice, only data sources and new views are changed by users, since these views are tailored to their concerns. The following box presents an execution for the ongoing example.

Ongoing Example, Part 23: **Exemplary Execution Order** ← List →

Figure 6.18☞221 sketches the execution of the orchestration for the ongoing example along Figure 6.18☞221. The execution traverses along the configured operators, which could be summarized as "customized linearization of the tree of operators". Each row depicts the execution of a single operator chain in one direction, i. e. a new row is depicted after the execution direction is reversed. The initialization of the SU(M)M (Section 6.5.4☞219) is already done with the (hidden) rows 01–08.

Now, row 09 provides the desired view by deriving it from the SUM, and the user changes this view, leading to the user differences $^{\text{User}}\Delta^{10}_{\text{ClassDiagram}}$. Afterwards, the execution loop is executed, which propagates the changes along the executed operators in order to update the SUM (in row 10) and Java (in row 15). Nodes with red text depict, that this node was updated, while red operator executions depict new branch differences, which must be propagated to all depending views.

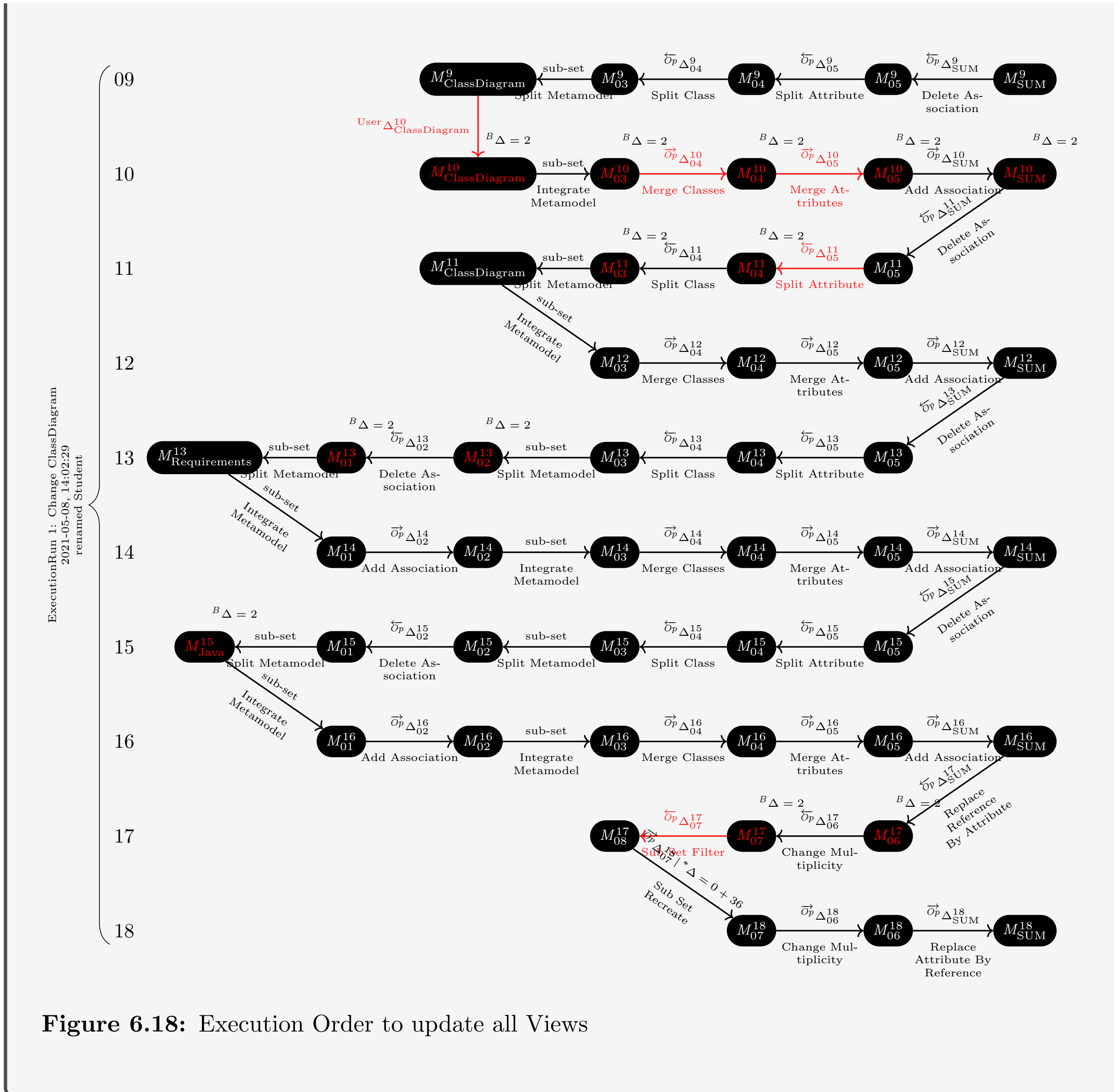


Figure 6.18: Execution Order to update all Views

6.6 Model and Metamodel Representation

This section designs the technical spaces to represent metamodels and models within MoConseMI. In particular, this section introduces the possible concepts for metamodels and conforming models, i. e. the modeling space according to Section 2.5 ☞84. The platform specialist selected EMF as technical space for realizing models and metamodels in MoConseMI, as motivated in Section 2.5.2 ☞86. Operators and model decisions configured by methodologists directly operate on models and metamodels. Since users work on concrete renderings of models, the information and user changes must be transformed to models and model differences, as requested by Requirement R 4 (Technical Spaces) ☞158. The design has to fulfill the following two *demands*:

flexible for in-place Transformations

1. Since (meta)models are transformed *in-place* by unidirectional operators, the technical spaces for technically realizing models and metamodels must allow *flexible* in-place transformations of models and metamodels. In particular, invalid and incompletely initialized objects and classes must be possible during the execution.

uniquely identifiable Model and Metamodel Elements

2. Each element in models and metamodels must be uniquely identifiable. This demand is required for the management of differences in models and metamodels, as designed in Section 6.7[☞227], but must be enabled by the design for models and metamodels in this Section 6.6[☞221]. In general, the identification of elements is a "*crucial requirement for consistency preservation*" (Kramer, 2017, pp. 66).

Section 6.6.1 shortly discusses EMF and some alternatives for technical spaces. Section 6.6.2 designs the realization of metamodels and their concepts. Section 6.6.3[☞223] determines the realization of models and their concepts. Section 6.6.4[☞225] introduces the concept of UUIDs for model and metamodel elements to fulfill demand 2. Section 6.6.5[☞226] introduces the concept of adapters to fulfill Requirement R 4 (Technical Spaces)[☞158].

6.6.1 Related Work

Possible alternatives for EMF as technical space for modeling metamodels and conforming models are already discussed in Section 2.5.1[☞84] and EMF as technical space with ECore as modeling space is selected for MoConseMI in Section 2.5.2[☞86]. Here, some important features of EMF are revived and an alternative for representing EMF models is introduced.

no Use of Profiles

SUM and its SUMM are realized as explicit model and its metamodel, without using profiles, neither UML profiles nor EMF profiles (Section 2.5.1[☞84]): Since profiles add additional information (or restrict existing concepts), but are not able to restructure or to remove already existing concepts, it is not possible to reduce redundant information in the SU(M)M and profiles are not sufficiently flexible. Therefore, profiles are not used for combining (meta)models in this thesis.

multiple Files and Namespaces

In EMF, models and metamodels can be spread over *multiple files* (Jahed, Bagherzadeh and Dingel, 2021). This concept is supported by MoConseMI and its implementing framework, since it improves flexibility. *Namespaces* can be seen as scopes for grouping elements in order to improve scalability in terms of performance and visualization (Jukšs, Verbrugge et al., 2018). By default, EMF supports nestable namespaces for metamodels with nestable `EPackages`, while namespaces for models are not supported. MoConseMI adds limited support for non-nestable namespaces for models by interpreting each file for models as model namespace.

relevant EMF Features

Some important features of EMF (Section 2.5.3[☞87]) to revive here are the use of EMF either in static or in dynamic way. When serializing EMF models and metamodels with XMI, EMF elements can be annotated with IDs. The selection of the concepts for EMF metamodels is motivated in Section 6.6.2.

Edapt Models

Edapt, as already discussed as a related approach for model co-evolution in Section 6.2.1[☞193], whose design ideas are partially reused, provides an alternative modeling space to realize EMF models, but not their EMF metamodels (Herrmannsdoerfer, Benz and Juergens, 2009, Figure 3). These models are called Edapt models in contrast to EMF models. It reuses the default EMF means to realize metamodels and its technical space is realized with EMF itself. Since Edapt models provide more flexibility compared with the default EMF means to realize models, Edapt is selected to realize models and its details are introduced in Section 6.6.3[☞223].

6.6.2 Metamodel Representation

dynamic EMF for Metamodels

According to Section 2.5.3[☞87], using EMF in static way requires to generate static Java source code which replicates the elements of the EMF metamodel. Since the metamodel is slightly changed by most of the operators, which causes to generate according Java source code for each of these unidirectional operators and loading and unloading this Java source code into the JVM, using static EMF does not scale. Instead, dynamic EMF is chosen

to realize metamodels in MoConseMI, since it allows to model metamodels dynamically in-memory and to update them in-place. This realization is the simplest way, since it is directly provided by EMF, and is sufficiently dynamic to fulfill demand 1.

selected Concepts of ECore for Metamodels

The supported amount of concepts of ECore for metamodels is already depicted in Figure 2.21☞88 and already shortly introduced in Section 2.5.3☞87: `EClasses`, `EDataTypes` and `EEnums` are chosen as typical classifiers for data modeling, also known and widely used and accepted from UML class diagrams. Associations and attributes known from UML class diagrams are realized with `EReferences` and `EAttributes` in ECore, generalized as features with `EStructuralFeature`. Nestable `EPackages` are supported as namespaces for structuring metamodels, as discussed in Section 13.3.3.3☞476. Generics are not selected, but could be supported in future extensions. Annotations with `EAnnotation` are not selected, since annotations are a means of EMF to allow practitioners to annotate EMF elements with arbitrary information, which is not required by MoConseMI. Concepts for modeling dynamics like `EOperations` similar to methods are not selected, since this thesis focuses on modeling data and not on their behavior or their business logic. Derived features of ECore are not directly supported, since this would require to explicitly model the functionality to calculate the values of derived features as well. But they can be simulated by configuring operators: Instead of writing Java source code into the generated (static) EMF code or using model queries (Ráth, Hegedüs and Varró, 2012) for derived features with EMF, operators can be used for calculating (and updating) additional values, e. g. →AddAttribute. The Java source code for calculating the derived values is integrated into the configured model decision of the operator.

Metamodel Concepts influence List of concrete Operators

Since unidirectional operators are designed to slightly and directly change metamodels, the design of concrete operators depends on the supported concepts of metamodels. Therefore, concrete operators depend on the supported concepts for metamodels of ECore, e. g. simple UML associations and UML compositions are supported with `EReferences`, but no UML aggregations. Therefore, the supported concepts for metamodels are selected in order to be as generic as possible. Summarizing, the technical details of EMF are important for the implementation of operators, the concepts for metamodels influence the list of concrete operators (Section 7.3☞243), and the general design of operators (Section 6.1☞185) is independent from selected metamodel concepts.

6.6.3 Model Representation

Edapt for Models

In contrast to Section 6.6.2☞222, dynamic EMF is not used for realizing models, since the EMF implementation restricts the management of possible models to its current metamodel, e. g. adding values is possible only for slots with a conforming attribute in the metamodel. This is beneficial for managing models with stable metamodels, but burdens the coupled transformations of models and their metamodels (demand 1). As alternative, means of Edapt are reused for realizing models in MoConseMI, since Edapt is successfully used for this purpose. Additionally, parts of the existing infrastructure for managing EMF models with Edapt can be reused for the implementation of MoConseMI. A disadvantage of using Edapt is, that the reuse of existing transformations in model decisions becomes harder. As look-ahead for Section 6.6.4☞225, another advantage of Edapt is, that objects in models directly store their UUID (with `Instance.uuid`).

Concepts of Edapt for Models

Figure 6.19☞224 depicts the most important concepts to realize EMF models conforming to EMF metamodels: In the upper part of the graphic, some concepts of ECore for metamodels are depicted, selected from Figure 2.21☞88. In the lower part of the graphic, the concepts of Edapt to realize models are depicted. Each object in the model is represented as an instance of `Instance`. The type of the object is a `EClass` in the metamodel, modeled via the `Type` container of the `Instance` (`(Instance.type).eClass`). Values of objects are stored in `Slots`, which point to their conforming `EStructuralFeature` in the metamodel,

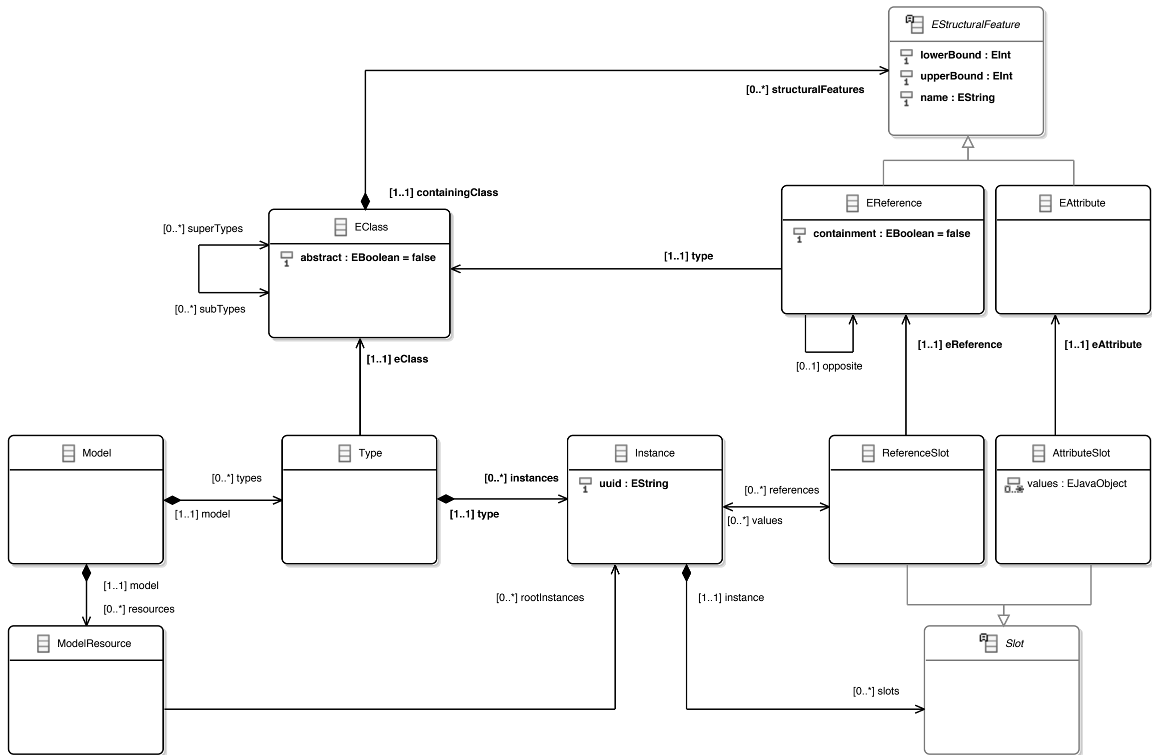


Figure 6.19: Concepts of EDAPT to represent EMF Models, similar to Herrmannsdoerfer, Benz and Juergens (2009, Figure 3)

distinguished into primitive values for `EAttributes` which are arbitrary Java objects, and into objects for `EReferences` which are instances of `Instance`. This design for model concepts allows to explicitly model types of objects and values of their slots in a flexible way.

Objects in a `Model` are organized into one or multiple files, represented by `ModelResources`, which point to their contained root objects according to the ECORE containment tree (Section 2.5.3☞87). These `ModelResources` are interpreted as (non-nestable) namespaces within EMF models and allow operators to manage objects of models in multiple namespaces.

combine EMF for Metamodels with EDAPT for Models

After showing the concepts for models in Figure 6.19, this graphic is extended by Figure 6.20☞225 in order to show, how metamodels realized with EMF and models realized with EDAPT are managed together[15] and provided to unidirectional operators: `MigrationInformation` is given to the operators for execution and serves as container for one metamodel and one conforming model. The metamodel is represented as a `ResourceSet` consisting of multiple `XMIResources` which represent files to store parts of metamodels in them in XMI format. The metamodel is structured by nestable `EPackages` as namespaces containing the classifiers. Each root `EPackages` is allocated to one `XMIResources`, which realizes the distribution of the metamodel over multiple files. `Models` in EDAPT refer to their current `Metamodels` by using `MetamodelResources`, which point to the root `EPackages` of the metamodel in EMF. `ResourceSet`, `XMIResource` and `EPackage` are provided by EMF, `MigrationInformation` is provided by MOCONSEMI, and all other concepts are provided by EDAPT.

[15]Note, that the means of EMF are depicted in a strongly simplified way by focusing on the relevant aspects, e. g. `XMIResource` is a `XMLResource` is a `Resource`, which is contained by the `ResourceSet` in reality.

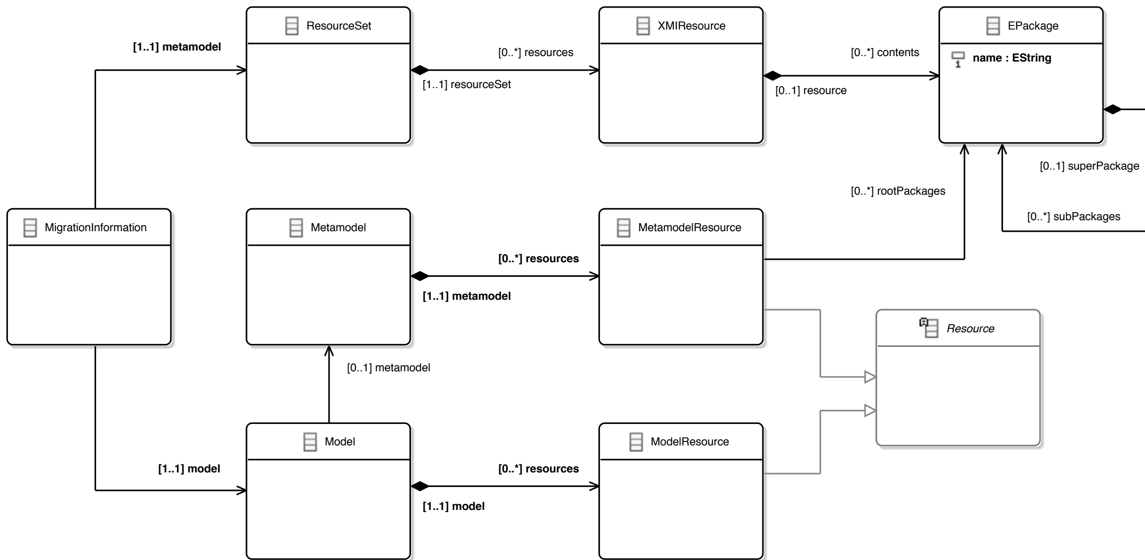


Figure 6.20: Combining EMF for representing Metamodels and EDAPT for representing Models

6.6.4 UUIDs

In order to fulfill demand 2, all elements in models and metamodels must be identifiable. The main motivation stems from the management of differences for models and metamodels, which must unambiguously point to the related model or metamodel element. This motivation will be deepened in Section 6.7☞ 227. For this identification, each element gets its own UUID, which is defined in Definition 27:

> Definition 27: **UUID**
>
> An UUID is a universally unique identifier which is stable over multiple executions of operators.

UUIDs are universally i. e. *globally unique* for all elements of models *and* metamodels, i. e. all instances of `Instance` in models and all instances of `EPackage`, `EClassifier`, `EStructuralFeature` and `EEnumLiteral` in metamodels. This counts for all metamodels and models at all positions in the orchestration, e. g. objects which are newly created and derived from the SUM for different new views must not have already used UUIDs.

UUIDs are globally unique

UUIDs must be *stable* over multiple executions of operators in order to identify same elements in different execution runs of operators (vertical) at the same position within the orchestration (horizontal). In contrast, elements which represent same concepts might have different UUIDs at different positions within the orchestration (horizontal), e. g. since they are changed by operators like ⇄MERGESPLITCLASSES from one model (position) to another model (position) in the orchestration. This refers to (vertical) *alignment* of objects in two different model versions (Anjorin, Buchmann et al., 2020), as in lenses (Diskin, Xiong and Czarnecki, 2011). As an alternative to UUIDs, explicit links could be used also for this vertical case (Diskin, König and Lawford, 2018, p. 24) in order to mark corresponding elements in different versions, but explicit links are not possible in MoConseMI, since the transformations are in-place.

UUIDs are stable

In order to get and to store the UUID of each element, the element could be annotated with its UUID. In particular, it is neither necessary nor desired to extend domain-specific metamodels with domain-independent concepts like UUIDs, as done by Goldschmidt (2010, pp. 283–284) and Kuryazov (2019, p. 116). Here in MoConseMI, for objects in models, the `uuid` attribute of `Instance` is exploited, which is designed by EDAPT for this purpose.

storing UUIDs

For elements in metamodels, ECore annotations could be used to store UUIDs, but EMF supports IDs for serialization with `XMIResources` (Figure 6.20[☞225]), since `XMIResources` internally manage maps to map elements to their ID for serialization. This approach is used, since it allows to directly serialize and persist UUIDs as well with the concept, which is dedicated by EMF for this purpose. In order to support the implementation of operators and the configuration of model decisions by methodologists, `MigrationInformation` provides supporting getters for retrieving UUIDs of metamodel elements.

6.6.5 Adapters

MoConseMI internally uses EMF for realizing metamodels (Section 6.6.2[☞222]) and Edapt for realizing models (Section 6.6.3[☞223]). But users usually work on concrete renderings of views realized with other technical spaces than the technical spaces used for (meta)models of these view(point)s. Therefore, bridges between EMF and other technical spaces are required and are designed in this section as *adapters* to fulfill Requirement R 4 (Technical Spaces)[☞158]. Adapters are developed by adapter providers during the use case "develop adapter" (Section 5.2.4[☞178]). This section discusses the general design of adapters, while Section 8.4[☞271] presents some implementations of adapters for concrete technical spaces. A concrete example for an adapter supporting the CSV format is given in Part 24[☞276] of the ongoing example.

Related Work

Adapters are special projectors in terms of Bézivin and Kurtev (2005) for EMF respectively Edapt as used in MoConseMI for importing and exporting metamodels and models. Mens and Van Gorp (2006) motivate the use of special transformations for switching between different technical spaces. Anjorin, Saller et al. (2013) provide a formal framework for adapters and motivate to use multiple transformation steps. Note the different meanings of the term *adapter* in Jelschen (2024) and in this thesis: While adapters realize the transformation between the original data in their concrete renderings and the Edapt format required by MoConseMI here, Jelschen (2024) calls this functionality *transformer*, since adapters are used to provide a uniform way to interact with tools there. On the other hand, adapters in MoConseMI ensure also the correct loading and saving of data at required points in time (including the possible sending of information to other locations), so that they provide also starting points for interaction with existing tools, while transformers in Jelschen (2024) are restricted to data conversation only.

Adapter vs Transformer

provide the Metamodel

Since the models of views conform to metamodels, adapters have to provide the *metamodel* once as precondition: Depending on the particular technical space for concrete renderings, the schema is explicit or implicit (Jin, Cordy and Dean, 2002). Explicit schemata can be transformed into EMF metamodels, while implicit schemata need to be derived from the concrete renderings or require other means like to transform parts of models into metamodels (Bézivin, 2005, p. 184), as for CSV, where the top row is interpreted as schema information (Section 8.4.4[☞275]).

transform Models

Having the metamodel, conforming *models* can be transformed from the concrete renderings realized with the external technical space: The relevant information depicted with concrete renderings must be transformed into Edapt models ("import") and vice versa ("export"). Since MoConseMI requires UUIDs for all (meta)model elements, UUIDs must be managed by adapters as well. In particular, the stability of UUIDs must be ensured, when importing and exporting models.

handle Model Differences

But adapters not only provide the metamodel in EMF and transform models in Edapt, but also handle *model differences*, i. e. $^{User}\Delta$ done on concrete renderings must be provided for MoConseMI, and $^{E}\Delta$ can be used to update the concrete renderings instead of completely recreating them. In order to provide user changes $^{User}\Delta$ as model differences (as designed in Section 6.7[☞227]), the adapter itself chooses and realizes one of the following possible strategies:

- If the concrete rendering is integrated in an appropriate tool or other environment, recorded user actions can be translated into model differences for the $^{User}\Delta$.
- If there is no dedicated strategy for directly identifying the $^{User}\Delta$, it can be calculated by comparing the previous complete model and the changed complete model e. g. with EMF Compare (Brun and Pierantonio, 2008).

A comparable discussion is done by Cicchetti, Ciccozzi and Leveque (2012) and more formally by Diskin, Xiong and Czarnecki (2010, p. 64).

Future Work: **UUID Mapping in Adapters**

Since model elements must have *globally* unique UUIDs, it is not sufficient, if the model elements of a single view have IDs which are unique within this view ("locally" unique). Usually, model elements can be uniquely identified by some domain-specific IDs like matriculation numbers for students or inventory numbers for products, but these "external IDs" are only locally unique and do not fulfill the requirements for real UUIDs, as used internally within MoConseMI. In order to handle (possible) collisions of internal and external UUIDs, adapters must map internal to external UUIDs and vice versa. This mapping must be maintained during import and export of models. While the MoConseMI framework provides some rudimentary support for this problem, e. g. with using view-specific prefixes to extend IDs to UUIDs, more concepts might be required for generic solutions.

6.7 Model and Metamodel Differences

Model differences respectively metamodel differences are modeling artifacts which describe changes in models respectively in metamodels. They are required to control and validate the execution of operators (Section 6.5☞213) and therefore are a central part of the design of MoConseMI. In particular, differences for models and metamodels have to fulfill the following three *demands*:

1. Representing and managing differences are required for elements in metamodels *and* models: Metamodel differences are required to validate, that unidirectional operators are inverse to each other regarding metamodel changes. Model differences are required to control the execution of operators, as deepened by the following demands. Model differences are influenced by changes in metamodels, which is deepened later for model difference co-evolution (Section 6.7.3☞235) and requires the integrated representation of model differences and metamodel differences.

   integrated Differences for Models and Metamodels

2. As discussed in Section 6.5.3☞217, in order to detect the fix-point, the loop to execute operators requires branch differences $^{B}\Delta$. These branch differences must be calculated.

   calculate Branch Differences $^{B}\Delta$

3. Differences must be *invertible* in order to restore removed information to prevent information loss in operators (Section 6.5.2☞214). Additionally, some special details of the implementing MoConseMI framework require to invert differences. In the future, transaction management would require to roll back transactions, i. e. user changes and automated follow-up changes of MoConseMI, by inverting and applying the occurred changes, as it will be discussed in Section 12.3☞462.

   invertible Differences

Section 6.7.1☞228 investigates related approaches for model and metamodel differences, with the focus on the representation of model differences, which is designed in Section 6.7.2☞229. Dependencies between model differences and metamodel differences

are needed for the valid and joint representation of model differences and metamodel differences and are handled in Section 6.7.3☞235. The calculation of the required $^{B}\Delta$ is designed in Section 6.7.4☞236.

6.7.1 Related Work

There are lots of related approaches for the representation of differences in models and metamodels as well as for the management of differences. Stephan and Cordy (2013) survey approaches for model comparison and model differences in various research areas including UML models, EMF models, product lines and process models. Kolovos, Di Ruscio et al. (2009) survey approaches to calculate differences between models based on model matching. Altmanninger, Seidl and Wimmer (2009) classify and survey approaches for model versioning and provide some features to represent differences in the following.

overriding vs operational Representation

Focusing on the *representation* of actual differences, there are two fundamental approaches to realize differences (Diskin, Xiong and Czarnecki, 2011): *Overriding* representations specifies the updated elements as additional, small (meta)model containing the updates, which can be applied to the previous version by overriding the old elements, like a special model merging. *Operational* representations encode the real differences explicitly, which can be applied directly on the previous version in order to update it to the new version. Here, operational representations are used and investigated only, since operational representations usually need less data, since overriding representations are valid (meta)models conforming to a (meta)metamodel. Additionally, operational representations could be designed to jointly represent differences for models and metamodels, as designed in Section 6.7.2☞229.

The features for *representing* differences, which are relevant for MoConseMI, are depicted in Figure 6.21☞229. According to demand 1, changes can be described for metamodels as *metamodel differences* or for models as *model differences* (or both). According to demand 3, differences might be *invertible* without additional information. This feature is called "reversibility" by Herrmannsdoerfer and Koegel (2010). In order to identify same elements in different versions, either *heuristics* or *UUIDs* can be used (Altmanninger, Seidl and Wimmer, 2009). Another alternative are *traces*, which connect same elements in different versions explicitly with each other, like in Van Der Straeten, Mens et al. (2003) or similar to trace links of out-place model transformations. The concepts (i. e. the abstract syntax) used to represent differences in models conforming to the same metamodel can be either *specific* for this metamodel or *generic* i. e. independent from the metamodel. Note, that approaches with metamodel-specific representations might be metamodel-generic as a whole, if means for the metamodel-specific representation are automatically generated for the actual metamodel. Changes are conceptually represented as sequences of single changes, which might be primitive or composite changes, by composing multiple changes. The technical representation of differences in-memory can be *text*-based, usually with lines, or *model*-based, usually with graphs or trees (Altmanninger, Seidl and Wimmer, 2009). A *textual representation* could be used for the visualization of differences and is directly available, if differences are represented as text in-memory.

Model Differences

There are several related approaches to represent *model differences*, including the following examples: Cicchetti, Di Ruscio and Pierantonio (2007) represent model differences model-based with a (automatically generated) metamodel-specific difference metamodel, which is applied for incrementally updating views (Cicchetti, Ciccozzi and Leveque, 2012). The DL approach (Kuryazov, 2019) represents model differences text-based with a (automatically generated) metamodel-specific syntax, supported with different services working on such model differences including difference calculator and difference optimizer (Kuryazov and Winter, 2014). Le Noir, Delande et al. (2011) encode model differences as statements for Prolog, which can be seen as text-based representation, which uses IDs and is

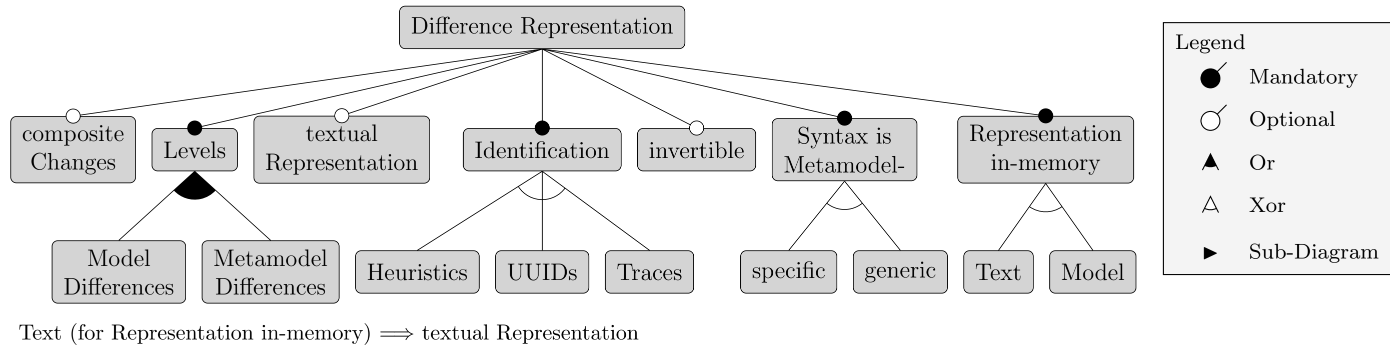


Figure 6.21: Feature Model for classifying functional Features of Difference Representation Approaches

metamodel-independent.

Metamodel Differences

There are several related approaches to represent *metamodel differences*, including the following examples: Vermolen, Wachsmuth and Visser (2012) present possible primitive and some composite changes in metamodels. Bruneliere, Perez et al. (2015) introduce a textual DSL for primitive metamodel changes, which is metamodel-independent. Burger and Gruschko (2010) present changes for MOF metamodels in form of a change metamodel.

There are few related approaches to represent *model and metamodel differences*: For the concurrent evolution of models and their metamodels, Cicchetti, Ciccozzi et al. (2011) apply the same techniques for model differences *twice* in order to represent model differences and metamodel differences by treating metamodels as models conforming to the meta-metamodel. This leads to two concurrent difference representations, but not to an integrated difference representation. In similar way, the model-based difference representation of Burger (2014, pp. 121ff, 138) can be applied twice to represent model differences and metamodel differences for EMF. Kramer (2017, pp. 152–155) presents a classification for changes in ECore-based models with a similar intention.

derive Findings by analyzing Differences

Other approaches analyze available differences to derive new findings from them, e. g. Vermolen, Wachsmuth and Visser (2012) reconstruct composite changes from primitive metamodel changes, while Maoz and Ringert (2018) relate syntactic model differences to semantic differences in the application domain of the models. Such ideas are not necessary for MoConseMI, since semantic consistency is defined on models and not on model differences.

6.7.2 Model Difference Representation

Basing on the identified features for differences in Section 6.7.1 ☞ 228, this sections designs the representation of differences for models and metamodels (demand 1). The main features of this design are depicted in Figure 6.22 ☞ 230 by marking the selected features of Figure 6.21.

general Design

Due to demand 1, both *model differences* and *metamodel differences* are jointly represented and might occur within the same change sequence, since unidirectional operators jointly change a model and its metamodel with an arbitrary order of model and metamodel changes. Changes are represented in *model*-based way, in order to relate model changes and metamodel changes directly to each other, which is required for Section 6.7.3 ☞ 235. The model-based difference representation is metamodel-*generic*, since metamodel-specific representations would require a different representation for most of the nodes in the orchestration, i. e. the number of difference representations would be linear to the number of configured operators, which usually change metamodels. A *textual representation* of changes is provided for compact documentations in human-readable way.

In order to detect same elements in different (meta)model versions, *UUIDs* are selected:

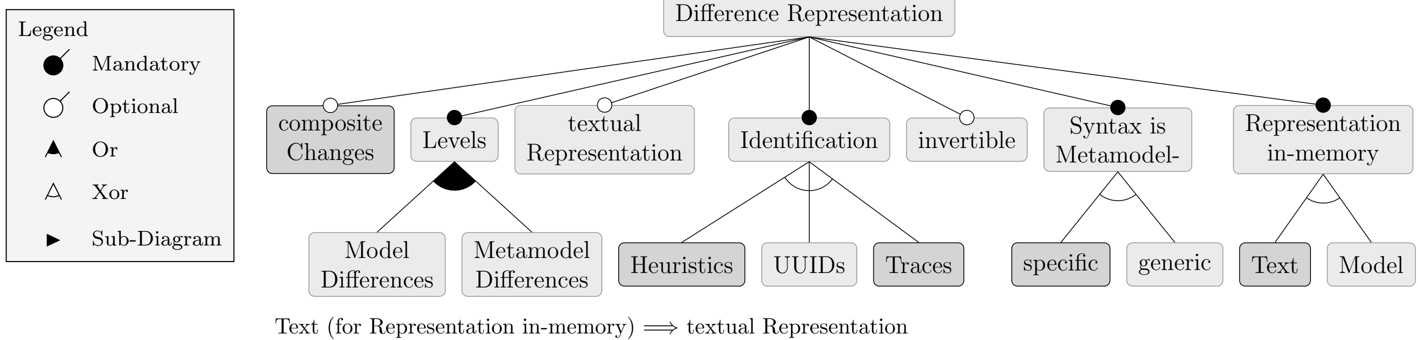


Figure 6.22: Selected Features for the Difference Representation in MoConseMI

UUIDs

Globally unique identifiers for elements enable to "*identify them over time*" (Wenzel, 2014, p. 679), since highly changed elements can still be matched (Altmanninger, Seidl and Wimmer, 2009). UUIDs provide reliable identifications in contrast to heuristics, which come with some degree of uncertainty. Therefore, the framework must support UUIDs (Altmanninger, Seidl and Wimmer, 2009, p. 281f), i. e. model elements and metamodel elements must have UUIDs, as designed in Section 6.6.4[☞225]. Van Der Straeten, Mens et al. (2003) present an UML profile to express evolution *traces* between different versions of the same UML model. Traces following this idea are not used here, since it would require to keep two complete versions of the same (meta)model in memory. This is not applicable, since the operators work in-place and update the current (meta)model, but do not provide new, additional (meta)models.

While Figure 6.22 shows the main features of difference representation in MoConseMI, the designed concrete differences are depicted for models and metamodels now. Afterwards, their completeness is discussed and the design in detail to realize the required functionalities is motivated.

Model Changes

The following kinds of changes are designed to represent relevant differences in Edapt models (Figure 6.19[☞224]). The arguments for these changes are discussed later.

Instance `createInstance`, `deleteInstance`, `changeInstanceType`

Slot `addValue`, `removeValue`

Changes in Slots

When representing added and removed *values in slots*, slots for `EAttributes` and slots for `EReferences` do not need to be dinstinguished, since values for `EReference` slots point to an objects as value. The slot to change is identified by the UUID of the object and by the feature the slot is conforming to. Both information are given as arguments to `addValue` and `removeValue`. With this design, slots for multi-value and single-value `EStructuralFeatures` can be addressed in uniform way. Slots themselves do not need to be created, since slots exist in Edapt by design only when there are some relevant values in the slot and therefore can be automatically created and deleted. All changes are delta-based, i. e. they describe only changed information and do not repeat unchanged information in models, in particular, unchanged values in multi-value slots are not repeated.

Changes in Objects

Objects in models are called instances by Edapt, therefore, the names of relevant changes for objects contain the term `Instance`. To change the type of an object would be possible also by `deleteInstance` with the old type and `createInstance` with the new type, in order to save one change kind for a more minimal set of change kinds, but then the recreated object would have a different UUID, according to Section 6.6.4[☞225]. This is disadvantageous, if the semantics of the current change scenario is to keep the object stable, but to change only its type, e. g. within the same type hierarchy. Changing types of objects is done also by other research: De Lara and Guerra (2017) explicitly propose to use dynamic

types to increase flexibility and reusability. When merging models, Westfechtel (2014, p. 768) explicitly cover the case to merge objects, whose types are different in different versions of the model. The "dynamic classification" is allowed by the OMG for an extended version of the MOF (Object Management Group, 2013). The Transformation Tool Contest 2010 (Rose, Herrmannsdoerfer et al., 2012, p. 351), comparing different tools for model co-evolution including classical model transformations, found, that retyping is beneficial for model co-evolution approaches, in particular for conciseness, as already motivated above. Therefore, the UUID is treated here as stable object identifier, while the type of the object is changable with `changeInstanceType`. Note, that the `Type`-class is used by EDAPT as a container to group all objects having the same type. Therefore, the type of objects can be changed in this easy way without creating or deleting types, since types are defined in the metamodel as `EClasses` with EMF.

The following kinds of changes are designed to represent relevant differences in EMF metamodels (Figure 2.21 ☞88):

Metamodel Changes

Namespace `createNamespace`, `deleteNamespace`

Enum `createEnum`, `deleteEnum`

Literal `createLiteral`, `deleteLiteral`, `changeLiteralValue`, `changeLiteralLiteral`

DataType `createDataType`, `deleteDataType`

Class `createClass`, `deleteClass`, `changeClassPotency`, `addSuperType`, `removeSuperType`

Feature `createFeature`, `deleteFeature`, `changeFeatureLowerBound`, `changeFeatureUpperBound`, `changeFeatureOpposite`, `changeFeatureKind` (i. e. association vs containment), `changeFeatureType`, `changeFeatureOwner`

Additionally, the name of namespaces, enums, literals, data types, classes and features can be changed with the generic `changeName` change. The namespace of (other) namespaces, enums, data types and classes in metamodels as well as of objects in models can be changed with the generic `changeNamespace` change.

These presented kinds of changes are complete to describe changes in models (Figure 6.19 ☞224) and metamodels (Figure 2.21 ☞88), since the following strategies are applied to derive change kinds for all relevant concepts of models and metamodels (i. e. the meta-metamodel):

Completeness of Differences

- For each non-abstract class X, there is a `createX` and a `deleteX` change, with the desired UUID as argument. The exceptions are `Slot` and `Type` in EDAPT, as discussed above. The X is part of the name for the change in order to indicate the wanted type of the newly created element.

- For each single-value attribute and association X, there is a `changeX` change with the previous value and the new value as well as the UUID for the element to change as arguments. The X indicates the attribute or association. The exceptions are the UUID attribute, since it is unchangable by design as discussed above, and the `instanceTypeName` of `EDataTypes`, since their values are assumed to be constant.

- For each multi-value attribute and association X, there is an `addX` and `removeX` change with the added respectively removed value as well as the UUID for the element to change as argument. This design allows to specify only the changes within a multi-value slot and avoids to repeat unchanged values. The X indicates the attribute or association.

If attributes or associations belong to super-classes or are generalizable, corresponding changes are defined only once and are valid for all affected classes. All these changes are primitive changes and no *composite* changes are designed, since they are not required by MoConseMI.

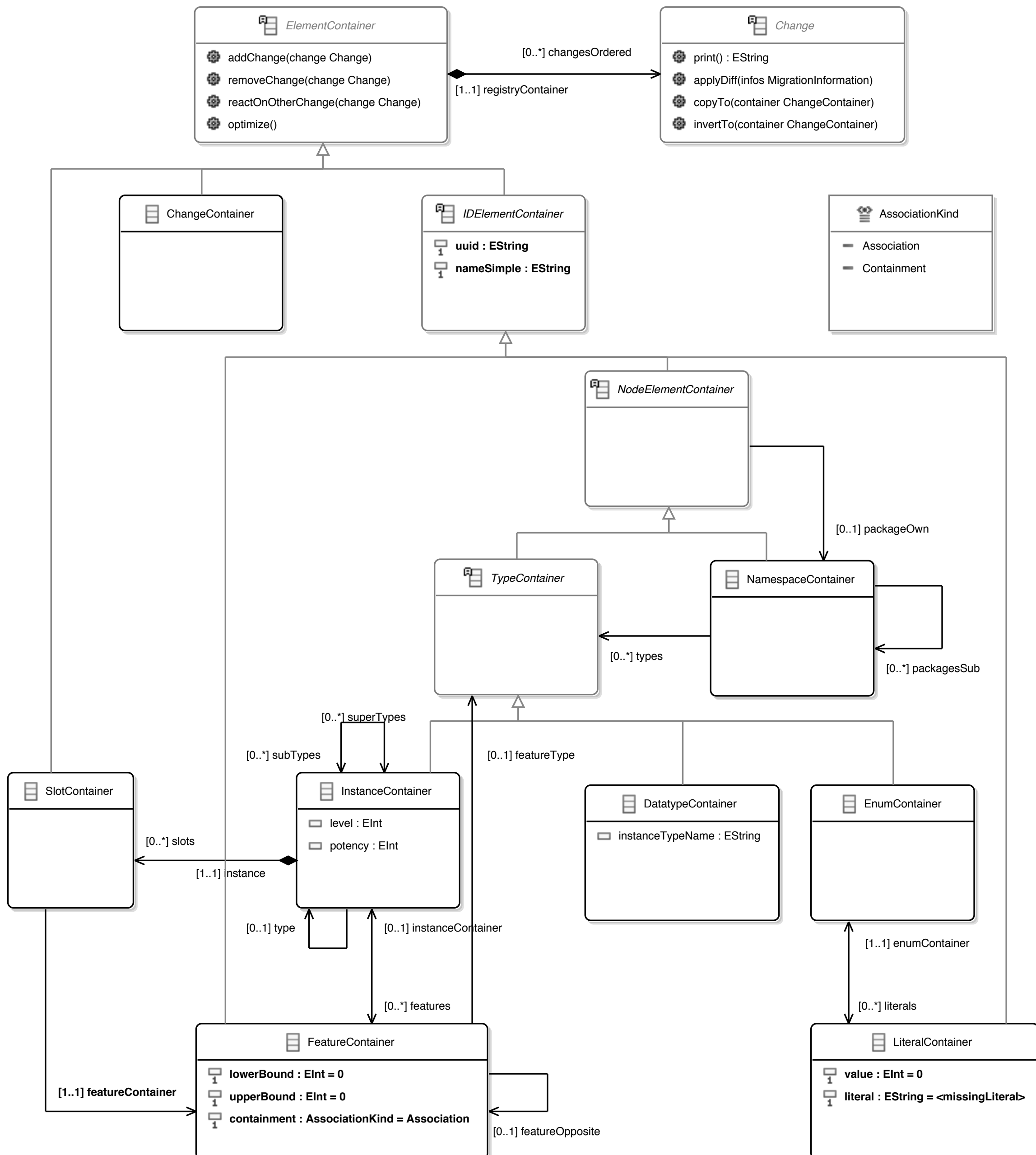


Figure 6.23: Overview about the Model-based Representation of Differences

Containers for Differences

These changes are managed in *model-based* way, as depicted in Figure 6.23 in simplified way: All changes for the same element are collected by a dedicated container, leading to `NamespaceContainer` for namespaces, `DatatypeContainer` for data types, `EnumContainer` for enums, `LiteralContainer` for literals, `FeatureContainer` for features including `EAttributes` and `EReferences`, `SlotContainer` for slots, and `InstanceContainer` for objects in models *and* classes in metamodels. These containers reflect (only) changed parts of the models and metamodels and collect all changes for them (`changesOrdered` in `ElementContainer`). The possible kinds of changes are listed above in the text and are skipped in Figure 6.23. The required containers for changed elements in models and metamodels are

managed by the `ChangeContainer`, as depicted in Figure 6.24.

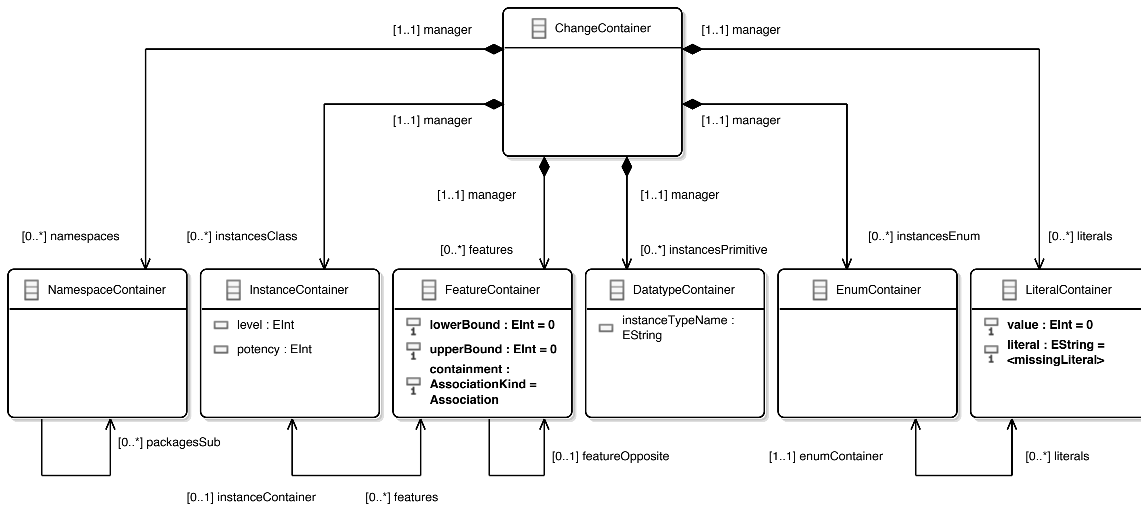


Figure 6.24: `ChangeContainer` as central Point to manage Model-based Differences

In order to integrate model differences and metamodel differences, this design can be seen as a first attempt for multi-level model differences: This counts for objects in models (`Instance` in EDAPT) *and* classes in metamodels (`EClass` in EMF), which are unified as `InstanceContainer`, having super types (within the same meta-level) and a type (in a higher meta-level), which are other `InstanceContainers`. This design fulfills demand 1, since a changed object could directly navigate to its class as type and analyze its differences. Accordingly, the distinction between abstract and non-abstract classes is done by `potency` values, while classes and objects are distinguished by different `level` values refering to different ontological levels. `EPackage` for EMF metamodels and `ModelResource` for EDAPT models are unified as `NamespaceContainer` and changes for namespaces work for both models and metamodels.

unified Concepts for Models and Metamodels

This tight integration is a precondition for *model difference co-evolution*, as designed in Section 6.7.3 ☞235. As an example, changes for adding values to a slot (in the model) can detect, if the feature of the slot is currently renamed (in the metamodel), since the `addValue`-change is stored by the corresponding `SlotContainer`, which points to its `FeatureContainer`.

Integration of Model and Metamodel Differences

This design of differences ensures, that differences are *invertible* and fulfill demand 3, i. e. the inversion of differences is possible without additional information like the current (meta)model: For the inversion of a sequence of changes, each change is inverted *and* the order of changes is reversed in the sequence as well. The inversion of the designed kinds of changes is possible, since all kinds of changes have an inverse change kind and the required arguments are information preserving, therefore, for each change an inverse change can be derived, which reverts the impact of the change. Applying a change and its inverse change to the same (meta)model in version i results in the version $i + 1$ which is the same as version i.

invertible Differences

- `createX` and `deleteX` are invertible to each other, since the UUID as argument is given to both changes and indicates the elements to delete respectively to create. This counts also for the type X of the element, which indicates the desired type of the element, i. e. `delete` is not invertible, since the desired type of the element to create is missing, but `deleteX` specifies this type.

- `changeX` for single-value attributes and associations is invertible, since the change gets the new *and* the previous value as arguments. Therefore, `changeX(oldValue, newValue)` can be inverted to `changeX(newValue, oldValue)`.

- `addX` and `removeX` for multi-value attributes and associations are invertible to each other, since they describe the previous state of the slot: `addX(v)` indicates, that the added value v did not exist before the change, therefore, this change can be reverted with `removeX(v)`, and vice versa.

optimize Differences

Additionally, this design of differences ensures, that differences can be *optimized*, as required for Section 6.7.4☞236, in order to validate, that metamodel changes of inverse unidirectional operators are really inverse to each other, and to optimize user changes in views. Difference optimization means, that those changes are removed from a valid sequence of changes, which do not impact differences in the (meta)model before and after applying the whole, non-optimized change sequence. In other words, all changes in an optimized change sequence have an impact which is still "visible" after applying the whole optimized change sequence.

`createX`, `deleteX` ⇒ ∅; `deleteX`, `createX` ⇒ ∅

- Elements might be created and deleted in alternating way: Since `createX` and `deleteX` are inverse to each other, applying both together have no effect. Therefore, pairs of `createX`/`deleteX` are removed for optimization.

`changeX(a,b)`, `changeX(b,c)` ⇒ `changeX(a,c)`; `changeX(a,a)` ⇒ ∅

- The value of single-value attributes or associations might be changed multiple times. All these changes can be optimized to a single change from the old value of the first `changeX` change to the new value of the last `changeX` change. Additionally, `changeX` changes, where the old value and the new value are equal, can be removed, since they have no impact at all.

`addX(v)`, `removeX(v)` ⇒ ∅; `removeX(v)`, `addX(v)` ⇒ ∅

- Adding and removing the same value for multi-value attributes or associations might occur: Since `addX` and `removeX` for the same value are inverse to each other, such pairs are removed for optimization.

Note, that changes for single-value attributes like "$a \rightarrow b \rightarrow a$" are not completely optimized by some related approaches, if the difference representation is not accurately designed, e. g. the DL approach (Kuryazov, 2019) describes these changes with `changeX(b)`, `changeX(a)`, resulting in `changeX(a)` after optimization, which is an incomplete optimization, since the initial value before the first change is unknown. The design in MoConseMI enables to optimize `changeX(a,b)`, `changeX(b,a)` into ∅ as the expected result, since the change sequence has no impact at the end. Note, that only changes targeting the *same* element are removed by difference optimization: In order to identify changes for the same element, the difference optimization needs UUIDs for all model elements (Section 6.6.4☞225), which are also used for the difference representation (Figure 6.22☞230).

validate Metamodel Changes of inverse unidirectional Operators

This difference optimization is used for, among others, the validation, that metamodel changes of inverse unidirectional operators are really inverse to each other. According to Figure 6.17☞216, the first unidirectional operators changes the metamodel with ${}^{\overrightarrow{Op}}_{MM}\Delta_2$, while its inverse unidirectional operator should revert these metamodel changes with ${}^{\overleftarrow{Op}}_{MM}\Delta_2$. Therefore, after these two changes, the initial metamodel MM_1 should be reached again, which is formalized as following:

$$\text{optimize}\left({}^{\overrightarrow{Op}}_{MM}\Delta_2\,,\ {}^{\overleftarrow{Op}}_{MM}\Delta_2\right) \stackrel{!}{=} \varnothing \tag{6.1}$$

There are some more functionalities for managing differences, which are provided by most of the difference management approaches and which are shortly explained now, since they are required for MoConseMI as well:

Application of represented differences on the current (meta)model is done by the represented changes themselves: Changes are applied directly to `MigrationInformation`

containing the current model and its metamodel, i. e. the concrete change determines the related element(s) in the model or the metamodel and change them accordingly. This is depicted by the method applyDiff of Change in Figure 6.23☞232. Applying differences is required by MoConseMI to restore removed information by applying model differences in inverted form (Section 6.5.2☞214).

Recording of differences listens to (meta)models which might be changed in-place and provides these changes as differences. Difference recording is required in MoConseMI to identify the changes of unidirectional operators in models and metamodels, which work in-place and therefore enable listening. Difference recording is realized by registering listeners, one listener for the current EMF metamodel and one listener for the current Edapt model. These two listeners are synchronized with each other, since both listeners write their detected changes into the same ChangeContainer. Herrmannsdoerfer and Koegel (2010) provide a metamodel-independent listener for EMF models in the context of Edapt, which is not reused here, since MoConseMI uses its own representation for differences and therefore needs custom listeners.

Calculation of differences takes two (meta)models conforming to the same (meta)metamodel as input and provides the calculated differences between them as model differences as output. Difference calculation is required by adapters in MoConseMI in order to detect user changes (Section 6.6.5☞226). Here, EMF Compare (Brun and Pierantonio, 2008) is used to calculate differences in the EMF Compare representation, which are translated into differences according to the representation in MoConseMI.

6.7.3 Model Difference Co-Evolution

Motivation for Model Difference Co-Evolution

In traditional *model* versioning approaches, the *metamodel* is stable. Since operators change models and metamodels, in MoConseMI the metamodel is changed as well. While the operators ensure *model co-evolution*, i. e. that models conform to their metamodels after the complete execution of operators, the representation of model differences must be adapted to evolving metamodels as well, leading to *model difference co-evolution*.

Since the operators ensure model co-evolution, the recorded differences for models fit to the evolved metamodel and its metamodel differences by design. But there are some other impacts of evolving metamodels, which are shortly discussed here:

rename Metamodel Elements

- Renaming attributes and associations in metamodels influences the representation of model differences for changed values in slots conforming to the renamed attributes or associations. As an example, if the value for the attribute “name” of an object o is changed from a to b with o.removeValue(name, a), o.addValue(name, b) and afterwards this attribute is renamed to “key”, then the model change should look like o.removeValue(key, a), o.addValue(key, b) according to the evolved metamodel. This co-evolution of model differences is solved by the model-based design of differences in MoConseMI, since addValue and removeValue do not explicitly encode “name” (or “key”) as pointer to the feature, but these changes directly point to the affected SlotContainer, which points to its FeatureContainer, which collects the changeName change. Using this design, the old name and the new name of the feature can be retrieved, e. g. for printing changes with textual representation. Similarily model difference co-evolution is realized for renamed classes and renamed enum literals. Summarizing, the model-based representation of model and metamodel differences as designed in Section 6.7.2☞229 fulfills this case for model difference co-evolution by design.

Metamodel Evolution changes default Values in Models

- ECore stipulates default values for slots conforming to `EAttributes` under certain conditions, but these default values are not explicitly stored in slots by Edapt. As an example, single-value attributes with Java primitive data types as attribute type have the Java-specific default value for this primitive data type as default value in EMF. Changing the attribute type in the metamodel, e. g. from `EBoolean` (Java default value for `boolean`: false) to `EInt` (Java default value for `int`: 0), might lead to changes in the model like `removeValue(attribute, false)`, `addValue(attribute, 0)`. Since these default values are not explicitly stored in Edapt models, no explicit changes occur in the model, which therefore cannot be detected by difference recorders. But when looking at the actual models, these changes occurred and must be made explicit by adding the mentioned model differences.

changing Containments changes Namespaces of Objects

- The containment tree in EMF models organizes all objects in one or multiple trees, i. e. each object (except for root objects) is contained in another object, forming trees (Section 2.5.3 ☞87). These containments are indicated by usual links conforming to `EReferences` which are marked as containment in the metamodel. Changing some of these containment flags in the metamodel might lead to changes of the containment tree of objects in the model, even without explicit model changes. Since the namespace of an object is the namespace of its parent object (`ModelResources` point only to root objects in Figure 6.19 ☞224), the namespaces of objects could be changed implicitly. Again, these model changes must be made explicit by adding the mentioned model differences.

Summary

Summarizing, metamodel evolution mostly affects models, but also model differences are affected in some special cases: While the renaming of metamodel elements affects mostly the *representation* of model differences and not the model differences directly in the first case, the other two cases depend on special features of the technical spaces EMF and Edapt, where metamodel evolution changes conforming models implicitly without changing the elements of model explicitly. The technical realization of model difference co-evolution for the second and the third case is done by the listener for Edapt models, which reacts on metamodel changes identified by the listener for EMF metamodels and adds some more model differences. Having correct model differences, even in special and rare situations, is important for the calculation of branch differences (Section 6.7.4), since the execution loop depends on them (Section 6.5.3 ☞217).

6.7.4 Branch Difference Calculation

$^{\mathrm{B}}\Delta$ vs $^{Op}\Delta$

This section designs the *calculation of branch differences* $^{\mathrm{B}}\Delta$ in order to fulfill demand 2. Executing operators in-place allows to navigate the current model between views and the SUM in order to update them. When executing operators, they produce model differences called operator differences $^{Op}_{M}\Delta$ in "horizontal direction", which describe, how models for views and the SUM can be transformed into each other. In contrast, branch differences describe the model changes between consecutive versions of the model at the same position within the orchestration "in vertical direction". These versions occur, when operators are executed multiple times (Section 6.5.3 ☞217). The user differences $^{\mathrm{User}}\Delta$ are special branch differences, since $^{\mathrm{User}}\Delta$ are manually done by users. Branch differences (as well as execution differences) occur only on model level, since the metamodel level is unchanged by design (Figure 6.2 ☞190).

Motivation for $^{\mathrm{B}}\Delta$

Branch differences are required in Section 6.5.3 ☞217 in order to detect the fix-point of the execution loop for operators. Additionally, execution differences $^{\mathrm{E}}\Delta$, which are optimized concatenations of branch differences, are useful for adapters in order to update their concrete renderings for changed views.

possible Strategies for ${}^{\text{B}}\Delta$ Calculation

Operators should not calculate the branch differences themselves, since it is possible to calculate branch differences in a generic way, as designed in this section. In general, there are *two possible strategies* to calculate branch differences, i. e. by difference calculation between two complete versions of the same model (Section 6.7.2 ☞ 229) or by comparing the model differences which led to the different model versions (Altmanninger, Seidl and Wimmer, 2009, p. 281). Both strategies are discussed in the following, along the example for a very simple orchestration in Figure 6.25: There is one view(point), which is derived by one operator from the SU(M)M. The user requested the view, got the corresponding model M^0_{View} as projected by the unidirectional operator from the current M^0_{SUM}, and changed the model by applying ${}^{\text{User}}\Delta^1_{\text{View}}$, resulting in the model M^1_{View}, which is transformed by the inverse unidirectional operator back into the SUM resulting in the updated M^1_{SUM}. The model changes ${}^{\overleftarrow{Op}}_{M}\Delta^0_{\text{View}}$ and ${}^{\overrightarrow{Op}}_{M}\Delta^1_{\text{View}}$ of the unidirectional operators are recorded (Section 6.7.2 ☞ 229) and therefore known. The user differences ${}^{\text{User}}\Delta^1_{\text{View}}$ are also known, since they are directly applied by users. While the updated SUM is known as M^1_{SUM}, it is unclear, how the SUM is changed compared to the previous version M^0_{SUM} of the SUM. Therefore, a reasonable design to calculate these differences as ${}^{\text{B}}\Delta^1_{\text{SUM}}$ is required.

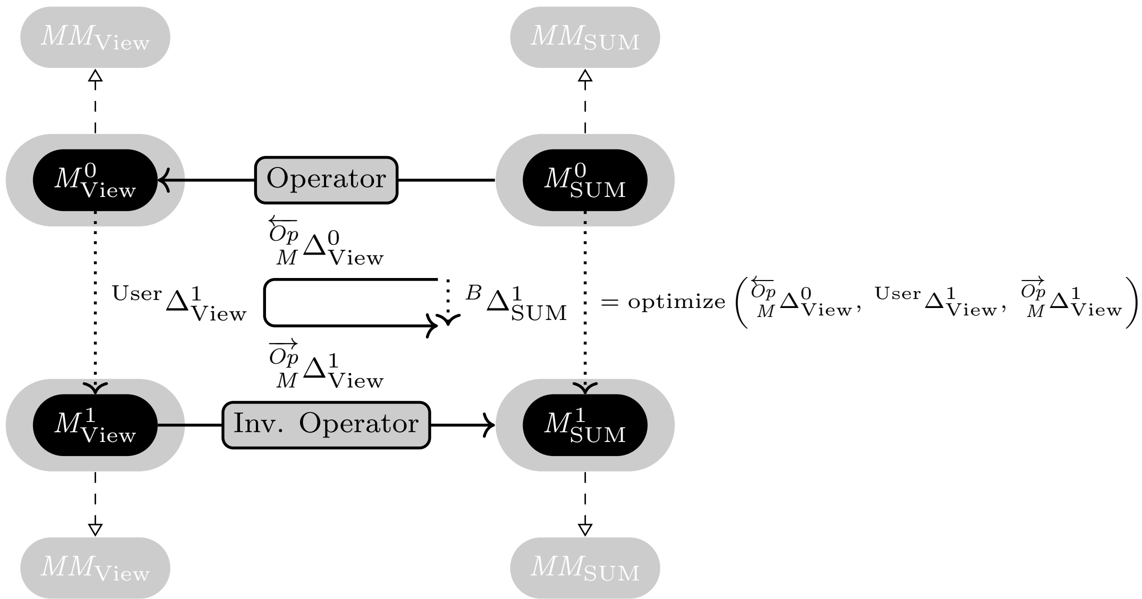


Figure 6.25: Conceptual Idea for the Calculation of Branch Differences

1. Calculation by Model Comparison

As *first strategy*, the calculation of ${}^{\text{B}}\Delta^1_{\text{SUM}}$ by difference calculation between M^0_{SUM} and M^1_{SUM} is possible in general, but comes with some drawbacks: While M^1_{SUM} is available as final result of executing the inverse unidirectional operator, M^0_{SUM} is not available anymore, since it was transformed *in-place* into M^0_{View}, then into M^1_{View} and finally into M^1_{SUM}. Therefore, M^0_{SUM} must be remembered for difference calculation. While this would be possible for this small orchestration, bigger orchestrations need to remember all intermediate models for all operators as well, leading to high memory consumption. Additionally, the difference calculation needs to take the whole models into account, not only the model differences. Since the second strategy can be realized as designed in the following and decreases these disadvantages, the second strategy is used by MoConseMI.

2. Calculation by Difference Comparison

The *second strategy* does not compare the whole models with each other, but compares relevant model differences with each other: As depicted by the solid black arrow in Figure 6.25, the M^0_{SUM} is changed three times in order to become the M^1_{SUM}. First, the unidirectional operator transformed M^0_{SUM} into M^0_{View} leading to ${}^{\overleftarrow{Op}}_{M}\Delta^0_{\text{View}}$, second, the user changed the model of the view to M^1_{View} with ${}^{\text{User}}\Delta^1_{\text{View}}$, third, M^1_{View} is transformed by

the inverse unidirectional operator to the M^1_{SUM} leading to ${}^{\overrightarrow{Op}}_{M}\Delta^1_{\text{View}}$. The concatenation of these three change sequences, i. e. ${}^{\overleftarrow{Op}}_{M}\Delta^0_{\text{View}}$, ${}^{\text{User}}\Delta^1_{\text{View}}$ and ${}^{\overrightarrow{Op}}_{M}\Delta^1_{\text{View}}$, describes the changes between the SUM in version 0 and the SUM in version 1. By *optimizing these concatenated changes*, unnecessary differences are removed and the differences, which really changed the SUM in version 1 in comparison to version 2, remain as ${}^{\text{B}}\Delta^1_{\text{SUM}}$. This calculation of branch differences is done *after* executing the inverse unidirectional operator, since it has to produce the required current operator differences *before*.

improved Performance

This strategy needs to remember all these model differences, which are usually smaller than whole models and therefore require less memory than the first strategy. Additionally, the optimization of model differences (Section 6.7.2 [☞229]) requires less calculation effort than the comparison of models in the first strategy, since the complexity depends on the number of model differences and not on the number of model elements.

generalized ${}^{\text{B}}\Delta$ Calculation

This idea, depicted along an example in Figure 6.25 [☞237], is generalized to calculate the branch differences after each execution of each unidirectional operator (fulfilling demand 2). In general, a unidirectional operator transformed the model from previous position P within the orchestration to the current position C and produced the model differences ${}^{\overrightarrow{Op}}_{M}\Delta^i$ during the current execution i. Comparing it with the previous execution $i-1$, which resulted in ${}^{\overleftarrow{Op}}_{M}\Delta^{i-1}$, and the current branch differences ${}^{\text{B}}\Delta^i_{\text{P}}$ at the previous position P results in the new branch differences ${}^{\text{B}}\Delta^i_{\text{C}}$, formalized as following:

$$ {}^{\text{B}}\Delta^i_{\text{C}} = \text{optimize}\left({}^{\overleftarrow{Op}}_{M}\Delta^{i-1},\ {}^{\text{B}}\Delta^i_{\text{P}},\ {}^{\overrightarrow{Op}}_{M}\Delta^i\right) \tag{6.2} $$

Exceptions

This formalizations counts also for switched directions of operators, i. e. with ${}^{\overrightarrow{Op}}_{M}\Delta^{i-1}$ and ${}^{\overleftarrow{Op}}_{M}\Delta^i$. One exception for this general calculation is ⇄CombineSeparateDataSources, whose unidirectional operators do not change the content of models at all, resulting in unchanged branch differences ${}^{\text{B}}\Delta^i_{\text{P}} = {}^{\text{B}}\Delta^i_{\text{C}}$. Another exception is the first execution of each bidirectional operator leading to ${}^{Op}\Delta^0$, since no branch differences can be calculated for its unidirectional operator, since the calculation needs the previous operator differences ${}^{Op}\Delta^{-1}$, which do not exist in this situation, since it is the first execution. This situation occurs during the initialization of the SUM and of the new views, as described in Section 6.5.4 [☞219].

Motivation for the Design of Differences

This design for the calculation of branch differences requires the proper difference optimization and motivates its design in Section 6.7.2 [☞229]. In order to identify changes for the same element, which are not relevant, the difference optimization needs UUIDs for all model elements (Section 6.6.4 [☞225]), which are also used for the difference representation (Section 6.7.2 [☞229]). Additionally, the model differences must be accurate, even with evolving metamodels, leading to the need for model difference co-evolution in Section 6.7.3 [☞235].

6.8 Summary

MoConseMI is Operator-based

This Chapter 6 [☞185] established the design of MoConseMI with all its concepts, basing on the main design decisions made in Chapter 5 [☞163]. MoConseMI uses *operators*, which are generic for *reuse* in recurring transformation scenarios and can be adapted to the project-specific metamodels by *metamodel decisions* (overcoming structural heterogeneity) and to the project-specific consistency rules by *model decisions* (overcoming semantic heterogeneity). There are bidirectional operators and unidirectional operators: While each unidirectional operator transforms one model and its metamodel in-place in one direction, each bidirectional operator couples one unidirectional operator with its inverse unidirectional operator for both directions. Bidirectional operators are selected, configured and combined into a tree, with the SU(M)M as root and the views, including data sources and new views, as leafs.

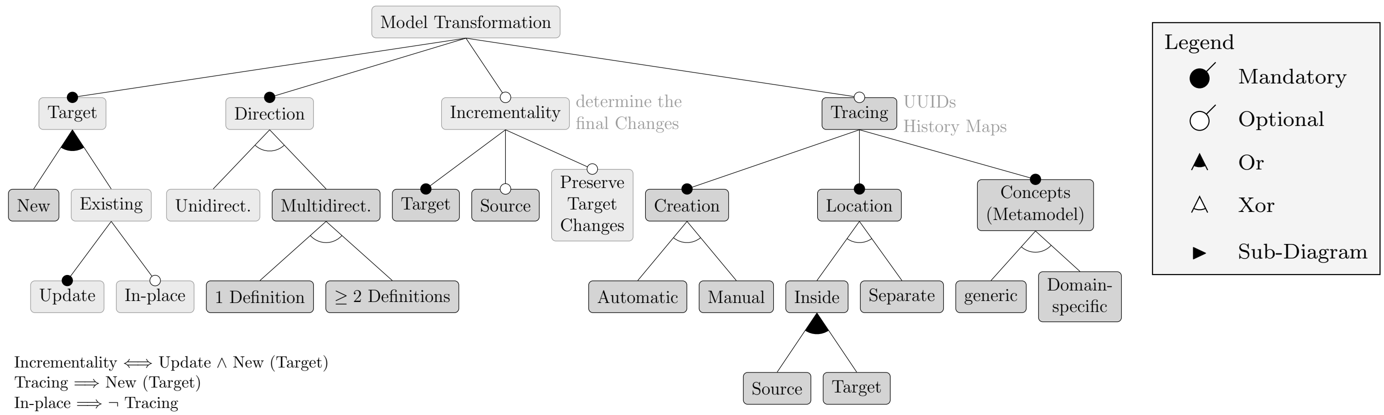


Figure 6.26: Selected Features of Model Transformations for unidirectional Operators

unidirectional Operators as (Model) Transformations

With this design, unidirectional operators represent small model transformations, as investigated in Section 2.2.3☞67. The selection of features of model transformations (Figure 2.14☞68) for unidirectional operators is depicted in Figure 6.26: Unidirectional operators can be seen as *unidirectional* and *in-place* model transformations with two models as source and target, while one model is a metamodel for the other model. While unidirectional operators do not improve performance by incrementality, they *preserve target changes* by in-place transformations and additional means to prevent information loss. There is no explicit tracing, since the transformation is in-place, but history maps can be used for storing trace information and stable UUIDs of elements can be used for some tracing.

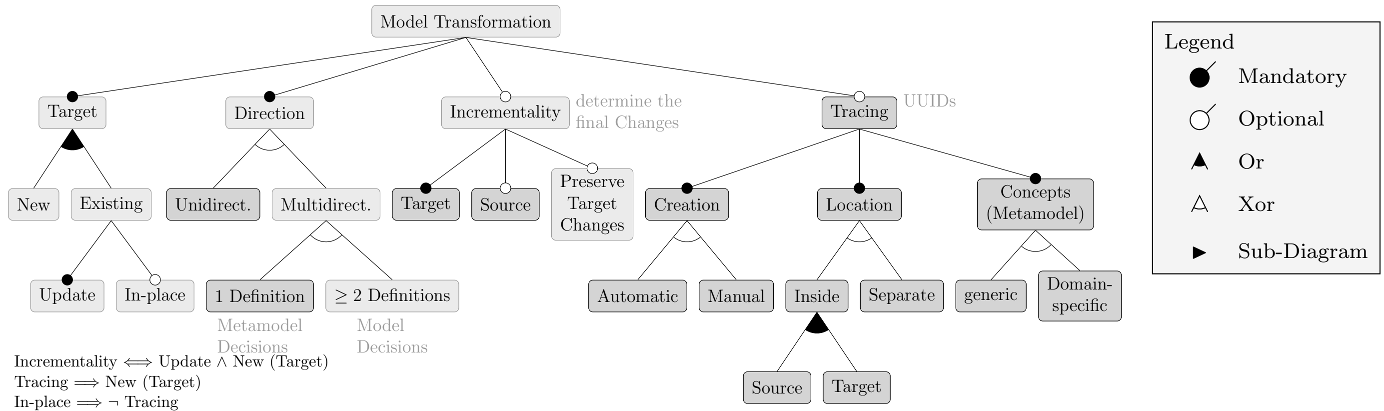


Figure 6.27: Selected Features of Model Transformations for the whole MoConseMI approach

MoConseMI as (Model) Transformation Framework

Since MoConseMI as a whole realizes change propagating by model synchronization with model transformations, MoConseMI is aligned to features of model transformations in Figure 6.27: While most of the selected features for MoConseMI are inherited from the selected features of unidirectional operators in Figure 6.26, MoConseMI is bidirectional as a special case of *multidirectional* transformations, while the bidirectional operators require one definition for metamodel decisions and *two definitions* for model decisions. Since history maps are an internal concept, history maps are not visible outside of MoConseMI. While the model synchronization is done in-place in general, *new* (meta)models for the SU(M)M and new view(point)s can be created by storing these (meta)models once during the initialization.

Execution of Operators

The general design of unidirectional operators including metamodel decisions is taken from Edapt and is extended here with model decisions and inverse unidirectional operators coupled into bidirectional operators. Additionally, the execution of operators is designed for bidirectional operators: The current (meta)models is transformed by chains of operators

in-place between the views and the SUM in order to propagate changes. This execution is looped in order to handle depending consistency goals and flexible orders of operators, which might result in automatically generated follow-up changes, which must be propagated to all views as well, until a fix-point is reached. For the execution order, a conservative strategy is chosen and needs more investigations in the future in order to improve the execution performance, while the results of applications will show that the execution works and provides the expected results.

EMF, EDAPT and Differences

Metamodels are realized by EMF and conforming models are realized by EDAPT as technical spaces. For the validation and controlling of the execution loop, differences in models and their metamodels are jointly represented. The designed difference representation allows to invert and optimize differences, which is required to calculate branch differences and execution differences, which define the fix-point of the execution loop.

Concrete operators conforming to this design are presented in Chapter 7 ☞241. While most of the concrete operators realize small, recurring metamodel evolution scenarios with configurable model co-evolution, ⇄ChangeModel enables custom model transformations for cases, where the other predefined operators are not beneficial. Limitations of this design are discussed in Section 14.3.1 ☞490. This design is realized as framework in Chapter 8 ☞263 to be reusable for several application examples in order to evaluate this design.

Chapter 7

Operators

After establishing the general design of operators in Section 6.1☞185, this section presents designed and implemented operators, which are applied in the evaluation in Part V☞467. Objectives of this section are the following:

- list the provided operators as overview for methodologists
- provide details about the operators, in particular metamodel and model decisions, which must be configured by methodologists, together with examples
- serve as reference for details of the operators, so that these details can be reduced in Part V☞467 during application

The next Section 7.1 presents some related work in which some of the operators are already introduced or discussed. Section 7.2 introduces the template, which is used in Section 7.3☞243 to describe the operators. Section 7.4☞262 summarizes the operators.

7.1 Related Work

This section shows some related work which propose or motivate operators, which target metamodel and model level and which are documented in details in the following Section 7.3☞243.

Burger and Gruschko (2010) analyze possible atomic changes for elements and their properties in MOF-based metamodels and classify their impact on conforming models. From these atomic changes, corresponding operators like →AddClass or →RenameClassifier can be derived.

Herrmannsdoerfer, Vermolen and Wachsmuth (2011) present a catalog of operators for EMOF-based metamodels like ECore in the context of Edapt (Section 6.2.1☞193) and include also more complex operators like →MergeClasses. Some of these operators are taken and adapted here.

Similar operators are also sketched by Cicchetti, Di Ruscio et al. (2008), by Meyers, Wimmer et al. (2012) and by Wachsmuth (2007), which are explicitly designed to support the model co-evolution. Some of these operators are taken and adapted here.

7.2 Template to describe Operators

This sections describes the template, which is used in Section 7.3☞243 to document operators: Each bidirectional operator is documented in its own sub-section. Following the design of operators as motivated in Section 6.1☞185, each bidirectional operator consists of

documentation template for bidirectional operators

two unidirectional operators, which are documented as sub-sub-sections. The following sub-sub-sections describe an abstract example for the current bidirectional operator, including a visualization of the impact on metamodels and models, the corresponding configurations of decisions and the execution details. The used exemplary metamodels and models are artificial and tailored to the current operator to be small and to present the specific impact of the operator at the same time, which is hard to realize with the same metamodel and model for all operators. This application is only an example and does not demonstrate all features of the operator.

Usually, an inverse bidirectional operator can be designed by swapping the involved unidirectional operators. Here, such inverse operators are named, but not documented, since its details are nearly the same, since the same unidirectional operators are involved. In rare cases, such an inverse bidirectional operator does not make sense, which is documented, too. In analogous way, unidirectional operators, which are inverse to each other, e. g. RenameClassifier, are documented only once.

documentation template for unidirectional operators

To document unidirectional operators, the following structure is used: At the beginning of the sub-sub-section, a longer text motivates the operator and describes some details of the operator. Afterwards, its details are documented using the following structure:

Metamodel Decisions lists all possible decisions for the metamodel level with its name, its data type and a description for its purpose.

Model Decisions lists all possible decisions for the model level with its name, its data type and a description for its purpose. Since model decisions allow to decide for each current object or value individually, they are rendered like methods: Parameters specify the current situation in form of current objects, values and so on, while the return type specifies possible answers.

Enumeration types Some decisions introduce new enumerations to define the possible values selectable by the methodologist. Such enumerations are listed with their possible literals here. If there are no new enumerations for this operator, this part is not shown.

Unidirectional Operators mentions, which other unidirectional operators are used internally by the current operator, and adds the purposes of all reused operators. If no sub operators are used by this operator, this part is not shown.

Preconditions lists all preconditions, which must be fulfilled, before the operator is executable. These preconditions target the input metamodel (and therefore also the input model) or the configurations for metamodel decisions done by the methodologist.

Default Configurations lists some configurations for model decisions for situations, which occur often. They can be reused (and adapted, if necessary) by the methodologist and ease his work. If there are no provided default configurations for this operator, this part is not shown.

hidden details

For brevity, some technical aspects are hidden in this section:

- the parameters `infos : MigrationInformation` and `decisionInfos : DecisionInformation`, since they are given to most model decisions
- metamodel and model decisions, which control UUIDs for created or deleted elements

7.3 List of Bidirectional Operators

This section lists the most important bidirectional operators, which are often used in the ongoing example or in the application examples in Part V☞467. Some rarely used operators are skipped here in order for brevity. This includes special operators like ⇄CombineSeparateDataSources for combining two data sources presented in Section 6.4.3☞205, and ⇄SubSet for filtering out unnecessary information presented in Section 6.4.4☞209, but also rarely used operators for creating and deleting rarely used concepts of EMF (see Section 6.6.2☞222) including the creation and deletion of data type, enums and enum literals, which are developed for completeness (as discussed in Section 13.2.2☞470), but which are not directly used for the presented application examples.

Each bidirectional operator is documented only for one direction, the bidirectional operator for the inverse direction simply swaps the two unidirectional operators.

7.3.1 AddDeleteOppositeRelation

As bidirectional operator, ⇄AddDeleteOppositeRelation consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →AddOppositeRelation in Section 7.3.1.1
- backward unidirectional operator ←DeleteOppositeRelation is described in Section 7.3.1.2☞244

The inverse bidirectional operator is ⇄DeleteAddOppositeRelation with swapped undirectional operators.

7.3.1.1 AddOppositeRelation

If the used technical space supports unidirectional and bidirectional associations (like EMF), modelers can choose for each association, if it should be unidirectional or bidirectional, depending on their needs. Using bidirectional association links allows the explicit navigation in both directions, while unidirectional associations support navigation only in one direction. On the other hand, unidirectional associations require less memory for their representation compared with bidirectional associations. If bidirectionality for navigation is not required anymore, this operator can be applied.

This operator is used to change an existing unidirectional association into a bidirectional one in the metamodel. In the model, all existing unidirectional links are enhanced to bidirectional links.

Metamodel Decisions of →AddOppositeRelation:

- existingReferenceFullyQualified : `String` [1]

  The fully-qualified name of the reference for which the opposite is created.

- newFeatureName : `String` [1]

  The name of the new opposite reference.

- newFeatureLowerBound : `int` [1]

  The lower bound of the new opposite reference. The selected bound must cover the number of links in the model. Otherwise, a wider bound has to be selected.

- newFeatureUpperBound : `int` [1]

  The upper bound of the new opposite reference. The selected bound must cover the number of links in the model. Otherwise, a wider bound has to be selected.

Model Decisions of →AddOppositeRelation: This operator has no individual model decisions.

Preconditions to be fulfilled before executing →AddOppositeRelation:

- The reference must not already have an opposite. (restricts existingReferenceFullyQualified)
- If the existing reference is a containment reference, the upper bound of the new opposite reference must be 1.

7.3.1.2 DeleteOppositeRelation

If the used technical space supports unidirectional and bidirectional associations (like EMF), modelers can choose for each association, if it should be unidirectional or bidirectional, depending on their needs. Using bidirectional association links allows the explicit navigation in both directions, while unidirectional associations support navigation only in one direction. On the other hand, unidirectional associations require less memory for their representation compared with bidirectional associations. If unidirectionality is enough for the initial (meta)model, but the integration would benefit from navigation also in the missing direction, this operator can be applied.

This operator is used to change an existing bidirectional association into an unidirectional one in the metamodel, which restricts the navigability to only one direction. In the model, all existing bidirectional links are reduced to unidirectional links.

Metamodel Decisions of →DeleteOppositeRelation:

- referenceFullyQualified : `String` [1]

  The fully qualified name of the reference whose opposite reference should be deleted.

Model Decisions of →DeleteOppositeRelation: This operator has no individual model decisions.

Preconditions to be fulfilled before executing →DeleteOppositeRelation:

- The opposite reference must not be containment. (restricts referenceFullyQualified)
- The reference needs to define an opposite reference. (restricts referenceFullyQualified)

7.3.2 AddDeleteAttribute

As bidirectional operator, ⇄AddDeleteAttribute consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →AddAttribute in Section 7.3.2.1 ☞245
- backward unidirectional operator ←DeleteAttribute in Section 7.3.2.2 ☞246

The inverse bidirectional operator is ⇄DeleteAddAttribute with swapped undirectional operators.

7.3.2.1 AddAttribute

In order to store additional information for objects, this operator enables to create an attribute and to determine the desired initial value for each object. A typical application scenario is also to calculate values and to provide this derived information in read-only way.

This operator creates a new attribute to the specified class in the metamodel and allows to specify the initial values in conforming attribute slots in the model.

Metamodel Decisions of →AddAttribute:

- classWithNewAttributeFullyQualified : `String` [1]

  The fully-qualified name of the class with the new attribute.

- attributeName : `String` [1]

  The name of the new attribute.

- attributeLowerBound : `int` [1]

  The lower bound of the new attribute.

- attributeUpperBound : `int` [1]

  The upper bound of the new attribute.

- attributeDataTypeFullyQualified : `String` [1]

  The fully qualified name of the type for the new attribute.

Model Decisions of →AddAttribute: Model decisions are encoded in decision : `AddAttributeDecision` [1], whose type is defined in `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸AddAttribute▸AddAttributeDecision` ("Determines the initial value(s) for this new attribute for each instance."). All their individual model decisions are listed here:

- computeInitialValue (instanceToFix : `Instance`, newAttribute : `EAttribute`) : `Object`

  This decision is called for each object conforming to the class with the new attribute and allows to specify the initial value for the attribute slot. Use null to indicate, that there is no initial value. Return a List with the values for a new multi-value attribute.

Preconditions to be fulfilled before executing →AddAttribute:

- Ensures, that the value for the lower bound is not negative. (restricts attributeLowerBound)

- Ensures, that the value for the upper bound is not zero. (restricts attributeUpperBound)

- Ensures, that the value for the lower bound is not bigger than the value for the upper bound.

- Ensures, that the class does not yet have a feature with the desired name. (restricts attributeName)

Default Configurations for model decisions for reuse in recurring situations are bundled in the following classes:

- `de`‣`unioldenburg`‣`se`‣`mmi`‣`framework`‣`operator`‣`unidirectional`‣`decisions`‣`AddAttributeIdUuid`: Uses the UUID of the instance as initial value for the new attribute.
- `de`‣`unioldenburg`‣`se`‣`mmi`‣`framework`‣`operator`‣`unidirectional`‣`decisions`‣`AddAttributeNothing`: Leaves the slot empty, i.e. no value is added.

7.3.2.2 DeleteAttribute

In order to remove values which are not required anymore, this operator allows to delete an attribute and all values conforming to this attribute.

This operator deletes an attribute in the metamodel. In the model, all values in conforming attribute slots are deleted.

Metamodel Decisions of →DeleteAttribute:

- fullyQualifiedAttributeName : `String` [1]

  The fully-qualified name of the attribute to delete.

Model Decisions of →DeleteAttribute: This operator has no individual model decisions.

Preconditions are not existing for →DeleteAttribute.

7.3.3 DeleteAddNamespace

As bidirectional operator, ⇄DeleteAddNamespace consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →DeleteNamespace in Section 7.3.3.1
- backward unidirectional operator ←AddNamespace in Section 7.3.3.2 ☞247

The inverse bidirectional operator is ⇄AddDeleteNamespace with swapped unidirectional operators.

7.3.3.1 DeleteNamespace

Since an empty namespace is not used anymore, this operator deletes such an empty namepace.

This operator deletes an existing, empty namespace either in the metamodel or in the model.

Metamodel Decisions of →DeleteNamespace:

- namespaceFullName : `String` [1]

  The fully-qualified name of the namespace to be deleted.

Model Decisions of →DeleteNamespace: This operator has no individual model decisions.

Preconditions are not existing for →DeleteNamespace.

7.3.3.2 AddNamespace

In order to improve the structure and grouping of elements, this operator creates a new namespace.

This operator creates a new namespace, either in the metamodel or in the model. Nothing more is changed.

Metamodel Decisions of →AddNamespace:

- parentNamespaceFullQualifiedName : `String` [0..1]

  The fully-qualified name of the parent namespace (might be null).

- namespaceName : `String` [1]

  The name of the new namespace.

- kind : `NamespaceKind` [0..1]

  Configures, if the new namespace is used for model elements or for metamodel elements (or if this information is unknown).

Model Decisions of →AddNamespace: This operator has no individual model decisions.

Enumeration types used for decisions in →AddNamespace:

- The enumeration `NamespaceKind` allows `METAMODEL`, `MODEL` and `UNDEFINED` as possible values.

Preconditions are not existing for →AddNamespace.

7.3.4 ChangeAttributeType

As bidirectional operator, ⇄ChangeAttributeType consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →ChangeAttributeType in Section 7.3.4.1
- backward unidirectional operator ←ChangeAttributeType in Section 7.3.4.1

The inverse bidirectional operator is ⇄ChangeAttributeType, because ⇄ChangeAttributeType is inverse to itself. Therefore, no additional inverse bidirectional operator needs to be introduced in this case.

7.3.4.1 ChangeAttributeType

If an inaccurate type is used for an attribute, this operator enables to change the type of the attribute and provides the customizable migration of values in attribute slots in the model. Inaccurate attribute type might stem from adapters for technical spaces which do not support all desired types, as an example, only String values are represented in the CSV format for spreadsheets.

This operator changes the type of an attribute in the metamodel. In the model, the values in corresponding attribute slots conforming to the old attribute type are replaced by values conforming to the new attribute type.

Metamodel Decisions of →ChangeAttributeType:

- fullyQualifiedAttributeName : `String` [1]

  The fully-qualified name of the attribute whose type should be changed.

- fullyQualifiedNewTypeName : `String` [1]

  The fully-qualified name of the new type for the attribute.

Model Decisions of →ChangeAttributeType: Model decisions are encoded in decision : `ChangeAttributeTypeDecision` [1], whose type is defined in `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸ChangeAttributeTypeExtended▸ChangeAttributeTypeDecision` ("Converts the current value conforming to the old attribute type into a new value conforming to the new attribute type."). All their individual model decisions are listed here:

- convert (input : `Object`) : `Object`

  This decision is called for each existing value conforming to the old attribute type in order to calculate a new value conforming to the new attribute type.

Preconditions to be fulfilled before executing →ChangeAttributeType:

- Ensures, that the new type is different than the current type of the attribute. (restricts fullyQualifiedNewTypeName)

Default Configurations for model decisions for reuse in recurring situations are bundled in the following classes:

- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertBooleanToString`: Uses the usual Java way to convert a boolean to String.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertDoubleRoundedToString`: Converts a double value after rounding to the desired number of decimals to String.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertDoubleToInteger`: Rounds a Double value to a Long value (i.e. removes the decimals) and converts it to String.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertDoubleToLongToString`: Rounds a Double value to a Long value (i.e. removes the decimals) and converts it to String.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertDoubleToString`: Uses the usual Java way to convert a double to String.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertFloatToString`: Uses the usual Java way to convert a float to String.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertIntegerToDouble`: Converts the integer value to double by Java casting.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertIntegerToString`: Uses the usual Java way to convert an integer to String.

- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertStringToBoolean`: Converts the String value to boolean by using the default Java parsing method.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertStringToDouble`: Converts the String value to double by using the default Java parsing method.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertStringToFloat`: Converts the String value to float by using the default Java parsing method.
- `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertStringToInteger`: Converts the String value to integer by using the default Java parsing method.

7.3.5 ChangeModel

As bidirectional operator, ⇄ChangeModel consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →ChangeModel in Section 7.3.5.1
- backward unidirectional operator ←ChangeModel in Section 7.3.5.1

The inverse bidirectional operator is ⇄ChangeModel, because ⇄ChangeModel is inverse to itself. Therefore, no additional inverse bidirectional operator needs to be introduced in this case.

7.3.5.1 ChangeModel

There are two motivations for this case: First, this operator allows to change the model in situations which do not require changes in the metamodel, because the metamodel (or at lease its relevant parts) is already sufficient. Second, complex evolution scenarios which require changes in metamodels and models and which are not realizable with single (or some chained) provided operators, could be realized with this operator, by doing (only) the metamodel evolution first without changing the model and by doing (only) the model evolution afterwards and as a whole with this operator.

This operator allows to change the model in arbitrary way, but does not change the metamodel.

Metamodel Decisions are not specified for →ChangeModel.

Model Decisions of →ChangeModel: Model decisions are encoded in decision : `ChangeModelDecision` [0..1], whose type is defined in `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸ChangeModel▸ChangeModelDecision` ("Allows to change the model in arbitrary way."). All their individual model decisions are listed here:

- changeModel ( )

  This decision allows to change the current model in arbitrary way. Changes in the metamodel are forbidden.

Preconditions are not existing for →ChangeModel.

Default Configurations for model decisions for reuse in recurring situations are bundled in the following classes:

- `de ▸ unioldenburg ▸ se ▸ mmi ▸ framework ▸ operator ▸ unidirectional ▸ decisions ▸ ChangeModelNothing`: Nothing is changed in the model.

7.3.6 ChangeMultiplicity

As bidirectional operator, ⇄ChangeMultiplicity consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →ChangeMultiplicity in Section 7.3.6.1
- backward unidirectional operator ←ChangeMultiplicity in Section 7.3.6.1

The inverse bidirectional operator is ⇄ChangeMultiplicity, because ⇄ChangeMultiplicity is inverse to itself. Therefore, no additional inverse bidirectional operator needs to be introduced in this case.

7.3.6.1 ChangeMultiplicity

In order to concretize multiplicities of existing features, it is necessary to ensure, that slots of objects in the model have the desired number of values, which is done by this operator.

This operator changes the multiplicities of a structural feature in the metamodel. In the model, this multiplicity is ensured for each affected object with the help of the model decision.

Metamodel Decisions of →ChangeMultiplicity:

- fullyQualifiedFeatureName : `String` [1]

  The fully-qualified name of the structural feature whose multiplicities should be changed.

- newLowerBound : `int` [1]

  The new multiplicity for the lower bound

- newUpperBound : `int` [1]

  The new multiplicity for the upper bound

Model Decisions of →ChangeMultiplicity: Model decisions are encoded in decision : `ChangeMultiplicityDecision` [0..1], whose type is defined in `de ▸ unioldenburg ▸ se ▸ mmi ▸ framework ▸ operator ▸ unidirectional ▸ ChangeMultiplicity ▸ ChangeMultiplicityDecision` ("Allows to fix slots whose number of values hurt the new lower or upper bounds. If there is no need to fix slots, no decision is required to configure."). All their individual model decisions are listed here:

- handleInstanceWithHurtLowerBound (slot : `Slot`, hurtLowerBound : `int`)

  This decision is called for each Slot which hurts the new lower bounds and allows to add some additional values to the slot.

- handleInstanceWithHurtUpperBound (slot : `Slot`, hurtUpperBound : `int`)

  This decision is called for each Slot which hurts the new upper bounds and allows to remove some existing values from the slot.

- handleInstanceWithValidBounds (slot : `Slot`, newLowerBound : `int`, newUpper-Bound : `int`)

  This decision is called for each Slot which does not hurt the new bounds and allows to adapt non-critical slots as well. There is a default configuration for this model decision ("By default, nothing is changed, since slots which do not hurt the new bounds do not need to be changed.").

Preconditions to be fulfilled before executing →CHANGEMULTIPLICITY:

- Ensures, that the value for the lower bound is not negative. (restricts newLower-Bound)
- Ensures, that the value for the upper bound is not zero. (restricts newUpperBound)
- Ensures, that the value for the lower bound is not bigger than the value for the upper bound.
- Ensures, that at least one bound of the multiplicity is changed.

Default Configurations for model decisions for reuse in recurring situations are bundled in the following classes:

- `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣ChangeMultiplicityNothingTodo`: Nothing is changed, due to the assumption of this configuration, that all bound are never hurt.

Additionally, one model decision provides a default configuration, directly together with its introduction, as documented above.

7.3.7 MergeSplitClasses

As bidirectional operator, ⇄MERGESPLITCLASSES consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →MERGECLASSES in Section 7.3.7.1
- backward unidirectional operator ←SPLITCLASS in Section 7.3.7.2☞254

No inverse bidirectional operator is defined for ⇄MERGESPLITCLASSES, because it was not required for application examples up to now.

7.3.7.1 MergeClasses

If the same concept is represented twice by two different classes, those classes can be unified by using this operator. This case occurs often after combining two data sources having overlaps in form of two classes describing the same concept.

This operator merges two classes on the metamodel level and merges corresponding instances on model level. The merging of instances is done 1-to-1, i.e. one source and one target instance are merged (exactly: 0..1-to-0..1), while merging 1-to-n or m-to-n is impossible with this operator.

Metamodel Decisions of →MERGECLASSES:

- targetClassName : `String` [1]

  The target class (fully qualified) which will be filled with the features of the source class.

- sourceClassName : `String` [1]

  The source class (fully qualified) which will be removed afterwards.

- fullyQualifiedFeatureNamesToMakeNonContainment : `String` [*]

  A list of fully qualified feature names (before(!) the merge on metamodel and model level) which are containments, but should be made non-containments during the metamodel evolution (without any additional model changes).

- fullyQualifiedFeatureNamesContainmentToChangeMultiplicity : `String` [*]

  A list of fully qualified feature names (before(!) the merge on metamodel and model level) which have an opposite containment reference, whos multiplicities should be changed (lower bound: 1 $\longleftrightarrow$ 0) during the metamodel evolution (without any additional model changes).

Model Decisions of →MERGECLASSES: Model decisions are encoded in decision : `MergeClassesDecision` [1], whose type is defined in `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸MergeClasses▸MergeClassesDecision` ("Decides mainly, which instances are merged into each other."). All their individual model decisions are listed here:

- mergeAgain (sourceInstance : `Instance`, targetInstance : `Instance`) : `MergeAgainDecision`

  After merging one source instance and one target instance into each other in a previous execution, this decision controls, if this merge should be done again, whichout checking their matching again.

- mergedSourceHasNoTargetAnyMore (sourceInstance : `Instance`) : `MergedNowMissingDecision`

  After merging one source instance and one target instance into each other in a previous execution, that target instance is missing now, while the source instance still exists, which raises the question, how to deal with the remaining source instance.

- mergedTargetHasNoSourceAnyMore (targetInstance : `Instance`) : `MergedNowMissingDecision`

  After merging one source instance and one target instance into each other in a previous execution, that source instance is missing now, while the target instance still exists, which raises the question, how to deal with the remaining target instance.

- areMatching (sourceInstance : `Instance`, targetInstance : `Instance`) : `MatchDecision`

  Decides if one given source object and one given target object should be merged.

- initializeTargetFeatures (instance : `Instance`)

  Allows to initialize features of the target class for (direct or indirect) instances of the source class. There is a default configuration for this model decision ("Here, no initialization for slots of target features is required.").

- initializeSourceFeatures (instance : `Instance`)

  Allows to initialize features of the source class for (direct or indirect) instances of the target class. There is a default configuration for this model decision ("Here, no initialization for slots of source features is required.").

- handleSourceWithoutMatch (sourceInstance : `Instance`) : `SearchedNoMatchDecision`

  Decides, how source instances without matching target instance should be treated. There is a default configuration for this model decision ("The source instance without target match is kept.").

- handleTargetWithoutMatch (targetInstance : `Instance`) : `SearchedNoMatchDecision`

  Decides, how target instances without matching source instance should be treated. There is a default configuration for this model decision ("The target instance without source match is kept.").

- determineMissingMatchForSource (sourceInstance : `Instance`) : `Instance`

  If requested, this decision allows to specify a matching target instance manually. It can be used to create a new instance for that, as example. There is a default configuration for this model decision ("Here, this case does not occur.").

- determineMissingMatchForTarget (targetInstance : `Instance`) : `Instance`

  If requested, this decision allows to specify a matching source instance manually. It can be used to create a new instance for that, as example. There is a default configuration for this model decision ("Here, this case does not occur.").

- mergeValuesOfAttributeOfCommonSuperClasses (sourceSlot : `AttributeSlot`, targetSlot : `AttributeSlot`)

  For the special case, that source instance and target instance to be merged have same super classes, this decision controls, how to merge their slots for attributes of such joint super classes. There is a default configuration for this model decision ("Realizes a simple merging without duplicate values.").

- mergeValuesOfReferenceOfCommonSuperClasses (sourceSlot : `ReferenceSlot`, targetSlot : `ReferenceSlot`)

  For the special case, that source instance and target instance to be merged have same super classes, this decision controls, how to merge their links for references of such joint super classes. There is a default configuration for this model decision ("Realizes a simple merging without duplicate links.").

Enumeration types used for decisions in →MergeClasses:

- The enumeration `MergeAgainDecision` allows `MERGE_AGAIN`, `MERGE_NOT_BUT_SEARCH_ONLY_SOURCE`, `MERGE_NOT_BUT_SEARCH_ONLY_TARGET`, `MERGE_NOT_BUT_SEARCH_BOTH` and `MERGE_NOT_NO_SEARCH` as possible values.
- The enumeration `MergedNowMissingDecision` allows `SEARCH_MATCHING`, `NO_SEARCH_NO_MERGE` and `DELETE_THIS_INSTANCE` as possible values.
- The enumeration `MatchDecision` allows `MATCH_FOUND`, `NO_MATCH` and `NEVER_MATCH_THIS_TARGET` as possible values.
- The enumeration `SearchedNoMatchDecision` allows `MIGRATE_WITHOUT_MATCH`, `DELETE_THIS_INSTANCE` and `DETERMINE_MISSING_MATCH` as possible values.

Unidirectional Operators which might be used internally by →MergeClasses:

- →ChangeContainmentsOfClass: TODO
- →ChangeMultiplicity: TODO

Preconditions to be fulfilled before executing →MergeClasses:

- The source and the target class have to be different.
- The source class and the target class must be both either abstract or non-abstract.
- The source class and the target class must be both either an interface or non interface.
- The source class must not be a (direct or indirect) sub class of the target class.
- The target class must not be a (direct or indirect) sub class of the source class.
- All features of Source and Target Classes have to have unique names.
- All features to change have to be containers.
- All features to change their multiplicities must have 0 or 1 as lower bound, since these two values will be switched.
- All features to change have to point to the source or target class (or to one of their super classes).

Default Configurations for model decisions for reuse in recurring situations are bundled in the following classes:

- `de` ▸ `unioldenburg` ▸ `se` ▸ `mmi` ▸ `framework` ▸ `operator` ▸ `unidirectional` ▸ `decisions` ▸ `NeverMergeClass`: Using this decision, all objects are kept, but no objects are merged.

Additionally, eight model decisions provide a default configuration, directly together with their introductions, as documented above.

7.3.7.2 SplitClass

If the same concept is represented by one class, but it is required to describe it with two classes, this operator can be used.

This operator splits one class into two classes on the metamodel level. On model level, each instance can either be split or become an instance of the new class or remains an instance of the existing class.

Metamodel Decisions of →SplitClass:

- classToSplitName : `String` [1]

  The fully qualified name of the class which should be split.

- outgoingFeaturesToMoveNames : (`String` → `ReferenceKind`) [*]

  The outgoing (not fully qualified) feature names (which should be moved to the new class) of the class which should be split. The ReferenceKind specifies the wanted direction of the feature in the new class.

- incomingReferencesToMoveNames : (`String` → `ReferenceKind`) [∗]

  The incoming (fully qualified) feature names (which should be moved to the new class) of the class which should be split. The ReferenceKind specifies the wanted direction of the feature in the new class.

- superClassesToMoveNames : (`String` → `Boolean`) [∗]

  The (fully qualified) names of the direct super classes (which should be moved to the new class) of the class which should be split. The Boolean encodes, whether the class should be a super class only for the existing class (null/not specified) or only for the new class (false) or for both classes (true).

- subClassesToMoveNames : (`String` → `Boolean`) [∗]

  The (fully qualified) names of the direct sub classes (which should be moved to the new class) of the class which should be split. The Boolean encodes, whether the class should be a sub class only for the existing class (null/not specified) or only for the new class (false) or for both classes (true).

- newFullClassName : `String` [1]

  The (fully qualified) name of the new class.

- fullyQualifiedFeatureNamesToMakeContainment : `String` [∗]

  A list of fully qualified feature names (before(!) the split on metamodel level) which are non-containments, but should be made containments during the metamodel evolution (without any additional model changes).

- fullyQualifiedFeatureNamesContainmentToChangeMultiplicity : `String` [∗]

  A list of fully qualified feature names (before(!) the split on metamodel level) which have an opposite containment reference, whos multiplicities should be changed (lower bound: 1 ⟷ 0) during the metamodel evolution (without any additional model changes).

Model Decisions of →SplitClass: Model decisions are encoded in decision : `SplitClassesDecision` [1], whose type is defined in `de` ▸ `unioldenburg` ▸ `se` ▸ `mmi` ▸ `framework` ▸ `operator` ▸ `unidirectional` ▸ `SplitClasses` ▸ `SplitClassesDecision` ("Decides mainly, how instances are split."). All their individual model decisions are listed here:

- shouldSplit (instanceToSplit : `Instance`) : `SplitOption`

  Decides, if an existing instance either is split (i.e. a new instance of the new class is created) or becomes an instance of the new class or remains an instance of the existing class.

- recreateContainmentForInstanceOfExistingClass (instanceOfExistingClass : `Instance`, instanceOfNewClass : `Instance`)

  This allows to ensure the containment for instances of the existing class. Reasons for the missing containment can be, that this instance is created newly or that the previous containment feature is moved to the new class. There is a default configuration for this model decision ("Does nothing, since this case is not relevant here.").

- recreateContainmentForInstanceOfNewClass (instanceOfExistingClass : `Instance`, instanceOfNewClass : `Instance`)

  This allows to ensure the containment for instances of the new class. Reasons for the missing containment can be, that this instance is created newly or that the previous

containment feature remains at the existing class. There is a default configuration for this model decision ("Does nothing, since this case is not relevant here.").

- recreateContainmentForSplitSubInstance (subInstance : `Instance`)

  This allows to recreate the (now missing) containment for instances of sub-classes which remain sub-classes only for the existing class. By moving existing (containment) features to the new class, an existing containment can be removed and must be fixed now. There is a default configuration for this model decision ("Does nothing, since this case is not relevant here.").

- recreateContainmentForNewSubInstance (subInstance : `Instance`)

  This allows to recreate the (now missing) containment for instances of sub-classes which will be sub-classes only for the new class. Since existing (containment) features might remain at the existing class, that existing containment is removed and must be fixed now. There is a default configuration for this model decision ("Does nothing, since this case is not relevant here.").

- splitValuesOfAttributeSlotOfCommonSuperClasses (slotToSplit : `AttributeSlot`, newSlot : `AttributeSlot`)

  If an instance is split, this allows to handle values for an attribute which belongs to a super-class which becomes a super-class of the new and(!) the existing class. This decision can be used e.g. to distribute the existing values to the two slots or to create some more new values. There is a default configuration for this model decision ("Does nothing, since this case is not relevant here.").

- splitValuesOfReferenceSlotOfCommonSuperClasses (slotToSplit : `ReferenceSlot`, newSlot : `ReferenceSlot`)

  If an instance is split, this allows to handle values for a reference which belongs to a super-class which becomes a super-class of the new and(!) the existing class. This decision can be used e.g. to distribute the existing values to the two slots or to create some more new values. There is a default configuration for this model decision ("Does nothing, since this case is not relevant here.").

- splitIncomingValue (instanceWhichWasSplitted : `Instance`, instanceWhichIsNew : `Instance`, incomingOppositeSlot : `ReferenceSlot`) : `SplitOptionIncomingLink`

  If an instance is split, this allows to decide for (unidirectional!) links pointing to the existing instance, whether the link should point to the existing instance XOR the new instance XOR to both. There is a default configuration for this model decision ("Does nothing, since this case is not relevant here.").

Enumeration types used for decisions in →SplitClass:

- The enumeration `ReferenceKind` allows `LOOP`, `BETWEEN_SOURCE_TO_TARGET`, `BETWEEN_TARGET_TO_SOURCE` and `OTHER` as possible values.

- The enumeration `SplitOption` allows `SPLIT`, `ONLY_NEW` and `ONLY_EXISTING` as possible values.

- The enumeration `SplitOptionIncomingLink` allows `BOTH`, `ONLY_NEW` and `ONLY_EXISTING` as possible values.

Unidirectional Operators which might be used internally by →SplitClass:

- →ChangeContainmentsOfClass: TODO
- →ChangeMultiplicity: TODO

Preconditions to be fulfilled before executing →SplitClass:

- The name of the new class name must be unique inside the wanted package.
- Each super classes to move must be a (direct or indirect) super class of the existing class.
- Each sub classes to move must be a (direct or indirect) sub class of the existing class.
- The features to be moved to the new class must exist at the existing class. (restricts outgoingFeaturesToMoveNames)
- The incoming references to be moved to the new class must exist at the existing class. (restricts incomingReferencesToMoveNames)
- All features to become containers must be no containers before. (restricts fullyQualifiedFeatureNamesToMakeContainment)
- All features to change their multiplicities must be used as containers. (restricts fullyQualifiedFeatureNamesContainmentToChangeMultiplicity)
- All features to change their multiplicities must have 0 or 1 as lower bound, since these two values will be switched.
- All features to change have to point to the existing class (or to one of its super classes).

Default Configurations for model decisions for reuse in recurring situations are bundled in the following classes:

- `de`‣`unioldenburg`‣`se`‣`mmi`‣`framework`‣`operator`‣`unidirectional`‣`decisions`‣`SplitClassesAlways`: Each object is always split into two objects.

Additionally, seven model decisions provide a default configuration, directly together with their introductions, as documented above.

7.3.8 RenameClassifier

As bidirectional operator, ⇄RenameClassifier consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →RenameClassifier in Section 7.3.8.1 ☞258
- backward unidirectional operator ←RenameClassifier in Section 7.3.8.1 ☞258

The inverse bidirectional operator is ⇄RenameClassifier, because ⇄RenameClassifier is inverse to itself. Therefore, no additional inverse bidirectional operator needs to be introduced in this case.

7.3.8.1 RenameClassifier

If an existing name is misleading or the name does not reflect the meaning of its element during the integration anymore, the current name can be changed with this operator. Another reason for a renaming is to ensure the uniqueness of names as preparation for following operators.

This operator renames a classifier (class, enumeration, data type), i.e. its name is changed, while the model remains unchanged.

Metamodel Decisions of →RenameClassifier:

- elementFullyQualified : `String` [1]

  The fully qualified name of the existing classifier in the metamodel to be renamed

- name : `String` [1]

  The new name for the classifier

Model Decisions of →RenameClassifier: This operator has no individual model decisions.

Preconditions to be fulfilled before executing →RenameClassifier:

- The new name must not be empty. (restricts name)
- The new name must be different than the existing name. (restricts name)
- The new name must not be already used by children of the element's parent. (restricts name)

7.3.9 RenameFeature

As bidirectional operator, ⇄RenameFeature consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →RenameFeature in Section 7.3.9.1
- backward unidirectional operator ←RenameFeature in Section 7.3.9.1

The inverse bidirectional operator is ⇄RenameFeature, because ⇄RenameFeature is inverse to itself. Therefore, no additional inverse bidirectional operator needs to be introduced in this case.

7.3.9.1 RenameFeature

If an existing name is misleading or the name does not reflect the meaning of its element during the integration anymore, the current name can be changed with this operator. Another reason for a renaming is to ensure the uniqueness of names as preparation for following operators.

This operator renames a feature (attribute, unidirectional reference), i.e. its name is changed, while the model remains unchanged.

Metamodel Decisions of →RenameFeature:

- elementFullyQualified : `String` [1]

  The fully qualified name of the existing classifier in the metamodel to be renamed

- name : `String` [1]

  The new name for the classifier

Model Decisions of →RenameFeature: This operator has no individual model decisions.

Preconditions to be fulfilled before executing →RenameFeature:

- The new name must not be empty. (restricts name)
- The new name must be different than the existing name. (restricts name)
- The new name must not be already used by children of the element's parent. (restricts name)

7.3.10 ReplaceAttributeByReference

As bidirectional operator, ⇄ReplaceAttributeByReference consists of the following two unidirectional operators, which are inverse to each other:

- forward unidirectional operator →ReplaceAttributeByReference is described in Section 7.3.10.1
- backward unidirectional operator ←ReplaceReferenceByAttribute is described in Section 7.3.10.2☞261

The inverse bidirectional operator is ⇄ReplaceReferenceByAttribute with swapped undirectional operators.

7.3.10.1 ReplaceAttributeByReference

If there is an attribute whose values can be interpreted as pointers to other objects (like for example foreign keys in data base terminology), this operator enables to replace these values by explicit links to the corresponding objects.

Migrates the values 1-to-1 in the model by replacing each attribute value to one object.

Metamodel Decisions of →ReplaceAttributeByReference:

- sourceClassName : `String` [1]

  The fully-qualified name of the source class containing the attribute to replace.

- attributeName : `String` [1]

  The simple name of the attribute to remove.

- targetClassName : `String` [1]

  The fully-qualified name of the target class as type of the new reference.

- oppositeReferenceName : `String` [0..1]

  The simple name of the opposite reference to create.

- oppositeLowerBound : `int` [0..1]

  The lower bound of the opposite reference to create.

- oppositeUpperBound : `int` [0..1]

  The upper bound of the opposite reference to create.

Model Decisions of →ReplaceAttributeByReference: Model decisions are encoded in decision : `ReplaceAttributeByReferenceDecision` [1], whose type is defined in `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸ReplaceAttributeByReference▸ReplaceAttributeByReferenceDecision` ("Calculates for each value in all attribute slots the corresponding object for the link to create instead."). All their individual model decisions are listed here:

- replaceValue (oldAttributeSlot : `AttributeSlot`, oldValue : `Object`, operator : `ReplaceAttributeByReference`) : `Instance`

  This decision is called for each existing value in each attribute slot and provides the object to link to as replacement for the old value. If no object is returned (null), the old value is removed and no new link is created.

- determineOppositeLinkOrder (newOppoSlot : `ReferenceSlot`, newValueForOppositeSlot : `Instance`) : `int`

  If the new association is bidirectional and its inverse association allows many values (i.e. upper bound is higher than one), this decision can be used to determine the wanted index for the inverse link for a replaced value. There is a default configuration for this model decision ("The index of the previous execution is reused. If this is the first execution, an arbitrary index is used.").

- determineOppositeLinkIndexInterpretation ( ) : `IndexInterpretation`

  This method is called exactly once before replacing values (if there is a multi-value opposite reference) and informs, how the indices (which are provided by 'determineOppositeLinkOrder(...)') should be interpreted. There is a default configuration for this model decision ("Ensures, that the indices of the previous execution are reused. If this is the first execution, the given indices are directly used.").

- handleInstanceWithoutValues (sourceWithoutValues : `Instance`, newReferenceSlot : `ReferenceSlot`, operator : `ReplaceAttributeByReference`)

  This decision is called for each object which as no old values in the attribute slot in order to allow to create links nevertheless. There is a default configuration for this model decision ("Nothing is done for objects without old attribute values.").

Enumeration types used for decisions in →ReplaceAttributeByReference:

- The enumeration `IndexInterpretation` allows `INDICIES_ARE_VALID_NOW`, `INDICIES_REPRESENT_FINAL_STATE` and `INDICIES_DONT_MATTER_DO_NOT_ASK` as possible values.

Unidirectional Operators which might be used internally by →ReplaceAttributeByReference:

- →AddAssociation: Used to create the new association.

- →DeleteAttribute: Used to delete the existing attribute.

Preconditions to be fulfilled before executing →ReplaceAttributeByReference:

- Ensures for the new opposite reference, that the value for the lower bound is not negative. (restricts oppositeLowerBound)
- Ensures for the new opposite reference, that the value for the upper bound is not zero. (restricts oppositeUpperBound)
- Ensures for the new opposite reference, that the value for the lower bound is not bigger than the value for the upper bound.
- The name of new opposite reference has to be unique, i.e. there is no feature with this name in the target class yet. (restricts oppositeReferenceName)

7.3.10.2 ReplaceReferenceByAttribute

When reducing the available information, there are cases, where linked information in terms of linked objects should not be used anymore, but only a name or another identifier for the linked object should be shown only instead (like for example the foreign key in data base terminology). Therefore, this operator enables to replace explicit links by values, whose interpretations refer indirectly to the previously linked objects.

Migrates the values 1-to-1 on model level.

Metamodel Decisions of →ReplaceReferenceByAttribute:

- sourceClassName : `String` [1]

  The fully-qualified name of the source class containing the reference.
- referenceName : `String` [1]

  The simple name of the reference to remove.
- newAttributeType : `String` [1]

  The type of the new attribute.

Model Decisions of →ReplaceReferenceByAttribute: Model decisions are encoded in decision : `ReplaceReferenceByAttributeDecision` [1], whose type is defined in `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸ReplaceReference-ByAttribute▸ReplaceReferenceByAttributeDecision` ("Calculates for each existing link the new value for the attribute slot."). All their individual model decisions are listed here:

- replaceLink (oldReferenceSlot : `ReferenceSlot`, oldValue : `Instance`, operator : `ReplaceReferenceByAttribute`) : `Object`

  This decision is called for each existing link in each reference slot and provides the attribute value as replacement for the old link. If no value is returned (null), the old link is removed and no new value is added.
- handleInstanceWithoutLinks (sourceWithoutValues : `Instance`, newAttributeSlot : `AttributeSlot`, operator : `ReplaceReferenceByAttribute`)

  This decision is called for each object which as no old links in the reference slot in order to allow to add values nevertheless. There is a default configuration for this model decision ("Nothing is done for objects without old links.").

Unidirectional Operators which might be used internally by →REPLACEREFERENCEBYATTRIBUTE:

- →ADDATTRIBUTE: Used to create the new attribute.
- →DELETEASSOCIATION: Used to delete the association.

Preconditions are not existing for →REPLACEREFERENCEBYATTRIBUTE.

7.4 Summary

Discussions of operator properties

While this section focuses on introducing the designed bidirectional and unidirectional operators, discussions of properties for these operators will follow in later sections: Section 13.2.2☞470 discusses, that the set of these operators is *complete*, but not minimal. Section 13.2.3☞471 discusses the *theoretic complexity* of operators in O-Notation. Section 13.2.4☞471 summarizes, that the operators are *reusable*, since they are used several times within the same application.

Before discussing these properties, the implementation of operators is sketched in Section 8.3☞267 to prepare their application in several domains in Part V☞467.

Chapter 8

Implementation

MoConseMI Framework realizes the MoConseMI Design

This section sketches the technical realization of the MoConseMI approach of Chapter 6☞185 as framework, which is reusable for several application examples in Part IV☞283 in order to evaluate the design of MoConseMI. The implementation of this MoConseMI framework can be seen as first, small validation of the design regarding its technical feasibility. Additionally, the framework is implemented in order to fulfill the technical requirements in Section 4.2☞157.

Outline

The described parts of the implementation are selected in order to support mainly *platform specialists* when extending the MoConseMI framework, but also *methodologists* are supported with technical details. Section 8.1 provides an overview of the MoConseMI framework with its main architectural concepts. Section 8.2☞264 sketches the realization of models, metamodels and differences, which are important for methodologists and adapter providers. Section 8.3☞267 provides technical details to implement the concretely designed operators of Chapter 7☞241. Section 8.4☞271 supports *adapter providers* with foundations how to implement adapters and presents predefined adapters. Section 8.5☞279 sketches visualizations which are provided by the MoConseMI framework in order to support methodologists during their application of MoConseMI. This section is summarized in Section 8.6☞280.

8.1 Overview

implemented in Java

The MoConseMI framework is implemented with the general-purpose programming language Java, since Java applications are executable on the main desktop operating systems Windows, Linux and MacOS. Additionally, EMF as technical space for Ecore metamodels and Edapt as technical space for EMF models are written in Java, which allows a seamless reuse of these technical spaces for the framework.

developed as Maven Project

In order to provide MoConseMI as reusable library according to Requirement R 5 (Reusable Library)☞158, the framework is developed as Maven project, which can reuse other Maven projects like for EMF and Edapt easily. Additionally, developing the framework as Maven project makes it independent from the used IDE, e. g. the framework can be developed not only with Eclipse, but also with other IDEs like IntelliJ.

Dependencies between Maven Projects

The source code for the implementation of the MoConseMI framework is split into multiple Maven projects, as depicted by the dark gray boxes in Figure 8.1☞264. The light gray boxes are external dependencies and the directed edges indicate, that the dependency at the edge end with an arrow tip is used by the project at the other end. The dependencies in Figure 8.1☞264 are strongly simplified, i. e. some direct dependencies are hidden, if they are already transitively included, some dependencies are renamed for readability and some dependencies which are depicted as single boxes consist of multiple Maven projects like EMF.

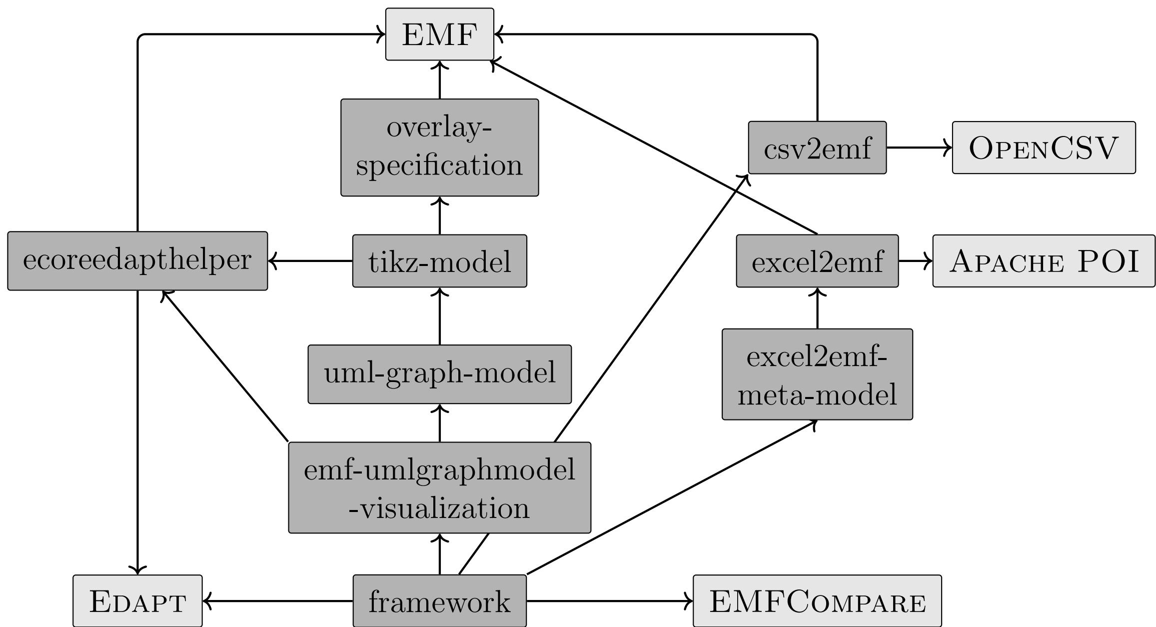


Figure 8.1: Maven Projects and Dependencies

own Maven Projects

The source code for MoConseMI is split into the Maven projects as shown in Figure 8.1, since they improve the architecture according to the principle of separation of concerns and they allow the reuse of functionality, e. g. `ecoreedapthelper` contains reusable supporting functionalities to deal with EMF and Edapt. The main functionalities of MoConseMI are realized in the `framework`, which (transitively) reuses implementations for EMF and Edapt to represent metamodels and models, as deepened in Section 8.2. The projects `csv2emf`, `excel2emf` and `excel2emf-meta-model` are used by adapters for CSV and Excel (Section 8.4☞271). EMF Compare is used to calculate differences between EMF models in order to detect user changes. The rest of the own projects are used to realize various visualizations (Section 8.5☞279).

MoConseMI as reusable Library

Since the MoConseMI approach and its framework do not force methodologists to use any graphical user interface (GUI) for the orchestration, the framework can be implemented and deployed as library, which is done by Maven. This library can be included into other software applications for ensuring inter-model consistency, e. g. as Maven dependency into other Maven projects. This design fulfills Requirement R 5 (Reusable Library)☞158.

8.2 Modeling Infrastructure

This section provides some technical insights into the technical realizations of models, metamodels and differences for models and metamodels, complementing their general design in Section 6.6☞221 and Section 6.7☞227. Additionally, these technical insights are required to implement operators, which change models and metamodels (Section 8.3☞267). The current model and its metamodel are coupled by an instance of `MigrationInformation` (Figure 6.20☞225) called `infos` by convention, which is given to unidirectional operators and their model decisions for in-place transformations. Additionally, `MigrationInformation` provides several supporting methods to ease the work to implement and configure unidirectional operators. Therefore, this section provides technical support for modeling models and metamodels for methodologists, platform specialists and adapter providers.

Metamodels

As designed in Section 6.6.2☞222, `metamodels` are realized with dynamic EMF. The possible concepts for metamodels are depicted in Figure 2.21☞88. The current metamodel as collection of one or more `EPackages` and their classifiers can be retrieved from `Migra-`

tionInformation infos by methods, for which some examples are given:

- List<EPackage> metamodel = infos.getRootPackages(); provides all root namespaces of the current metamodel.
- EAttribute attrib = (EAttribute) infos.getMetamodelElementByName("package.Class.attribute"); provides arbitrary elements in the metamodel by specifying its fully-qualified name.
- Alternatively, EEnumLiteral literal = infos.getLiteralByName("package.Enum .literal"); provides elements without type casting.

The values of metamodel elements, which are identified in this way, can be queried and changed by the usual means of dynamic EMF (Steinberg, Budinsky et al., 2009). Creating and deleting instances of the meta-classes in Figure 2.21☞88 must not be done with the usual means of dynamic EMF, but with special methods provided explicitly for these purposes by MigrationInformation infos. These methods must be used, since MigrationInformation internally checks and enforces uniqueness of UUIDs (Section 6.6.4☞225).

- As an example for creation, a new class can be created with EClass newClass = infos.createClass("uuid", "name", abstractFlag, containerPackage); by specifying its UUID (or null resulting in a random UUID), the desired name for the new class, a boolean indicating whether the new class is abstract or not, and the EPackage of the new class.
- As an example for deletion, infos.deleteEnum(enum); deletes the specified EEnum and all its EEnumLiterals.

Models

As designed in Section 6.6.3☞223, *models* are realized with EDAPT: The concepts for models are depicted in Figure 6.19☞224. The objects in the current model can be retrieved from MigrationInformation infos by methods, for which some examples are given:

- List<Instance> instances = infos.getInstances("package.Class"); provides all objects which have the specified class (but not one of its super classes) as type.
- List<Instance> instances = infos.getAllInstances("package.Class"); provides all objects which have the specified class (or one of its super classes) as type.
- Instance instance = infos.getInstanceByUuid("UUID of the object"); provides a single object by its UUID.

The objects, which are identified in this way, can be queried and changed with the following means, which internally maintain the Slots for internal representation in EDAPT:

- In order to *get* the value for a feature (EAttribute xor EReference) with the name "featureName", use the following methods provided by Instance:
  - For an EAttribute, use instance.get("featureName") to get the concrete value in case of single-value attributes or to get a List of values in case of multi-value attributes.
  - For a single-value EReference, use instance.getLink("featureName") to get the Instance (or null).
  - For a multi-value EReference, use instance.getLinks("featureName") to get a List of Instances.

In some cases, these easy methods do not find an existing Slot, since the operator already did some changes in the metamodel (currently, this is true at least for →MergeClasses). In that case, use the methods getSlotValueSingle(...) or getSlotValueMulti(...) of EcoreHelper.

- In order to *change* the value for a feature (EAttribute xor EReference) with the name "featureName", use the following methods provided by Instance:
  - For single-value features, use instance.set("featureName", newValue).
  - For adding a new value to a multi-value feature, use instance.add("featureName", additionalValue).
  - For removing an existing value from a multi-value feature, use instance.remove ("featureName", existingValueToRemove).

  If the feature is a bidirectional EReference, it is sufficient to change one direction (the other one will be automatically fixed by Edapt).

Creating and deleting objects must not be done with the usual means of Edapt, but with special methods provided explicitly for these purposes by MigrationInformation infos. These methods must be used, since MigrationInformation internally checks and enforces uniqueness of UUIDs (Section 6.6.4[☞225]). Additionally, the Types of Edapt are automatically and internally maintained.

- In order to create a new object, use Instance newInstance = infos.createInstance("new UUID", "package.Class");.
- In order to delete an existing object, use infos.deleteInstance("UUID", "true, if contained instances should be deleted as well, false otherwise");.

Differences

Differences for models and metamodels are designed in Section 6.7[☞227]. Although this difference representation is designed to be model-based, the difference representation is directly and manually realized with Java inside the Maven project framework (Figure 8.1[☞264]) and not explicitly model-based with EMF, since the representation contains lots of additional functionalities including parts of the model difference co-evolution (Section 6.7.3[☞235]), which is easier to realize in pure Java as in EMF. Differences can be represented in textual form for documentation. Differences can be serialized with EMF, i. e. the serialization of differences is explicitly model-based.

All differences are collected in a ChangeContainer (Figure 6.24[☞233]): Each difference targets one particular element in either the model or the metamodel. Differences which target the same element are collected by the ChangeContainer in a container which represents this changed element. In order to check, whether an element is changed, the following methods of ChangeContainer changes can be used, demonstrated here for features:

- If the UUID of the element is known, FeatureContainer feature = changes.getFeature("featureUuid"); provides the feature by UUID.
- If the current name of the element is known, FeatureContainer feature = (FeatureContainer) changes.getContainedElementByFullyQualifiedName("package.Class.feature"); provides the feature by its fully-qualified name.
- In both cases, List<Change> list = feature.getChangesOrdered(); provides the differences, which directly changed this feature. Alternatively, boolean changed = feature.isNotEmpty(); determines, whether the feature is directly changed.

8.3 Operator Implementation

This section supports platform specialists with technical details, how to implement additional operators. Section 8.3.1 implements unidirectional operators completely in Java and discusses, why no traditional model transformations are used. Section 8.3.2☞270 implements bidirectional operators by coupling two unidirectional operators which are inverse to each other. Section 8.3.3☞270 implements a Java API, which is used by methodologists to select and combine bidirectional operators for the orchestration.

8.3.1 Unidirectional Operators

When executing a unidirectional operator, its generic behavior together with concrete configurations for its metamodel decisions and its model decisions are executed. All these three parts determine the behavior of unidirectional operators and are realized with Java and not with traditional model transformations, as discussed in the following paragraphs.

While there are lots of *formal approaches for consistency* (Lucas, Molina and Toval, 2009) like Reder and Egyed (2012) with mathematical expressions for formally describing consistency, Java is easier to learn and to apply than formal specifications for methodologists inexperienced with formalisms. In general, formal methods seem to be not very popular (Lucas, Molina and Toval, 2009, p. 1637). Le Noir, Delande et al. (2011) compare direct Java checks for consistency with PROLOG-based consistency checking finding that Java is faster, while the comparison is limited due to prototypical implementations. The most important argument against formalizations is, that MoConseMI is model synchronization-based and therefore requires transformations but no formalizations for execution, as discussed and designed in Section 6.3.2☞200.

no Formalizations

The *generic behavior of unidirectional operators* is implemented with imperative statements written in Java, which directly transform the current model and its metamodels in-place. Imperative specifications with Java are preferred over traditional model transformation approaches for the following reasons:

implementing unidirectional Operators with Java

- When comparing transformations written in Java and in model transformations regarding (accidental) complexity, like for ATL, Götz, Tichy and Kehrer (2021) found, that "*even newer versions of Java still having to deal with the complexity overhead that ATL is able to hide. Specifically, while the traversal complexity could be greatly reduced through the use of newer language features, handling traces still entails a large overhead*" (Götz, Tichy and Kehrer, 2021, p. 132). Since the implementation of unidirectional operators enforces methodologists to configure model decisions along dedicated Java interfaces, this overhead is reduced, since "only some questions" in form of model decisions must be answered. Traversing models in Java is eased by the streaming API (Götz, Tichy and Kehrer, 2021), while explicit links are not required by MoConseMI. Instead, more flexible history maps can be used to remember arbitrary information, while such mechanisms are usually not supported by model transformation languages.

- Additionally, "*there is insufficient evidence for and [...] research about properties of model transformation languages*" (Götz, Tichy and Groner, 2021, p. 489), which makes a comparison hard in general. For model co-evolution as a specific application of model transformations, dedicated approaches might be preferred about model transformations and general-purpose languages (Rose, Kolovos et al., 2014; Götz, Tichy and Groner, 2021), but again this statement is weak, since it served as motivation for developing such approaches.

- Even if dedicated languages provide concepts tailored to specific problems resulting in better solutions, stakeholders might have more (self-)confidence in general-purpose

languages like Java, since they are more experienced with them, as found for specifying constraints (e. g. for consistency goals in rule-based approaches) with OCL in contrast to Java (Maraee and Sturm, 2021; Yue and Ali, 2016).

- Even the model transformation community itself is noticing or expecting, that dedicated model transformation approaches are becoming less popular (Burgueño, Cabot and Gérard, 2019). In particular, the development of huge and complex transformation scenarios seems to be easier to manage with general purpose languages.

- Henshin as in-place model transformation approach, for which Arendt, Biermann et al. (2010) sketched a prototype for model co-evolution with transformations for metamodels and transformations for models, organized as operators, is not used, since there are two Henshin transformations, one transformation for the metamodel and one transformation for the model, which are independently executed. With that design, transformations for models and metamodels cannot be mixed, which is necessary e. g. for →ReplaceAttributeByReference, which first creates the new `EReference` in the metamodel, then converts attribute values into links in the model, and finally deletes the `EAttribute` in the metamodel.

- Transformations for multi-level modeling, which could easily transform elements in different meta-levels, are hard to use here, since models and metamodels are not represented as multi-level models by MoConseMI.

Summarizing, Java is chosen as imperative general-purpose programming language for implementing unidirectional operators, since Java is chosen for the implementation of the whole MoConseMI framework (Section 8.1☞263), transformations for models and metamodels can be mixed with each other depending on the particular operators, and additional information in form of history maps and branch differences can be exploited in a more generic way.

After selecting Java for the implementation of unidirectional operators, there are some guidelines for platform specialists, when they implement a new unidirectional operator to realize a generic metamodel evolution scenario:

Guideline for implementing unidirectional Operators

- Classes implementing unidirectional operators inherit the class `ExtendedMigrationOperator`, since it specifies the API for unidirectional operators. In particular, the method `execute(MigrationInformation infos, DecisionInformation decisionInfos, OperatorExecutor subOperatorExecutor)` contains the transformation of models and metamodels as provided by `MigrationInformation infos`. `DecisionInformation decisionInfos` provides access to the history maps and the current branch differences. `OperatorExecutor subOperatorExecutor` allows to execute other unidirectional operators as sub-operators.

- Since the framework must support UUIDs, operators must manage UUIDs for all model and metamodel elements, in particular, the stability of UUIDs must be ensured. The UUIDs for newly created elements can be made configurable by methodologists with decisions. UUIDs of elements which are deleted by the unidirectional operator and are restored by its inverse unidirectional operator could be remembered in history maps.

- Platform specialists must use only their own history map (`decisionInfos.getHistoryOperator()`) for implementing the unidirectional operator, since the other history map is reserved for methodologists for configuring model decisions. This detail ensures, that platform specialists and methodologists do not need to agree about their keys used in the history maps and accidental key conflicts cannot occur.

- If the metamodel evolution scenario consists of multiple metamodel differences, parts of it could be realized by already existing unidirectional operators as sub-operators. This strategy is recommended, since it allows the reuse of (sometimes very complex) logic for metamodel evolution and model co-evolution scenarios. Such sub-operators must be executed with the method `operator.checkAndExecute(MigrationInformation infos, DecisionInformation decisionInfos, Op-eratorExecutor subOperatorExecutor)` and `infos` and `subOperatorExecutor` as values of the parent operator, only `decisionInfos` might be different to the given `decisionInfos` and should be calculated with `decisionInfos.forSubOperator("keySubOperator")` as easiest solution.
- Section 8.2☞264 provides technical hints, how to create, delete and change objects in EDAPT models and elements in EMF metamodels. Additionally, the new unidirectional operators must be registered in `IntegrationOperations` due to technical issues of the EDAPT infrastructure.
- Platform specialists should define *metamodel decisions* for new operators in order to map the generic metamodel evolution scenario to the current metamodel.
- *Model decisions* should be defined for degrees of freedom within the required model co-evolution, e. g. how to deal with model elements which are invalid after the metamodel evolution, or how to create new model elements for newly created elements in the metamodel. In model decisions, the parameters
  - `infos : MigrationInformation`
  - `decisionInfos : DecisionInformation`

  should be always provided by platform specialists, since they are required for flexible configurations by methodologists, e. g. to create new objects (via `infos`) or to remember important information (with history maps in `decisionInfos`). In the documentation of operators in Section 7.3☞243, these parameters are hidden, since they are default parameters for all model decisions.

configuring Metamodel Decisions with Java expressions

Metamodel decisions allow to control the changes of unidirectional operators in the current metamodel. Since these changes are oriented along small, single metamodel evolution scenarios, metamodel decisions request single, static values like class names or multiplicities for associations (Section 6.2.2☞196). To configure these values, no model transformations are necessary, but single values can be directly provided with simple Java expressions.

configuring Model Decisions with Java

Model decisions allow to adapt the model co-evolution to project-specific needs and allow to realize project-specific consistency rules. Java is chosen for configuring model decisions by methodologists, since they increase the flexibility to deal with history maps and branch differences, similar to the argumentation for implementing unidirectional operators with Java by platform specialists (see above). Imperative configurations for model decisions are similar to imperative fixes in EVL (Kolovos, Paige and Polack, 2009), used also for EVL+STRACE (Samimi-Dehkordi, Zamani and Kolahdouz-Rahimi, 2018) as BX approach (Section 3.3.1☞108). Additionally, using Java prevents the need for methodologists to learn a possibly new model transformation language and decreases their learning curves, as discussed in Section 14.3.1.3☞493.

Guideline for configuring Model Decisions

After selecting Java for the configuration of model decisions, there are some guidelines for methodologists, when they configure model decisions to fulfill project-specific needs:

- When they create, delete or change objects, methodologists have to ensure, that UUIDs remain stable. UUIDs of elements which are deleted by a model configuration for the unidirectional operator and are restored by the model configuration of its inverse unidirectional operator could be remembered in history maps.

- Methodologists must use only their own history map (`decisionInfos.getHistoryDecisions()`) for configuring model decisions, since the other history map is reserved for platform specialists for implementing the unidirectional operator. This detail ensures, that platform specialists and methodologists do not need to agree about their keys used in the history maps and accidental key conflicts cannot occur.

- Section 8.2[☞264] provides technical hints, how to create, delete and change objects in Edapt models.

8.3.2 Bidirectional Operators

forward Configurations for Decisions to unidirectional Operators

Bidirectional operators couple two unidirectional operators which are inverse to each other and whose implementation is done according to Section 8.3.1[☞267]. When executing a bidirectional operator, one of its configured unidirectional operators is executed, depending on the current direction of the transformation (Section 6.5.3[☞217]). Bidirectional operators are selected, configured and combined into the orchestration by methodologists. Therefore, the configurations for model decisions and metamodel decisions of both unidirectional operators are given by methodologists to the bidirectional operator. The bidirectional operator forwards these configurations to its unidirectional operators. In other words, the decisions of the unidirectional operators determine the decisions of the bidirectional operator. The implementation of bidirectional operators by platform specialists requires to take the configurations for decisions of unidirectional operators from methodologists and to forward them to the unidirectional operators.

concrete Implementation

Classes implementing bidirectional operators inherit the class `BaseOperatorBidirectional`, since it defines the API for bidirectional operators. `BaseOperatorBidirectional` has two sub-classes, `NoSettingsBidirectional` and `RecreateRevertBidirectional`. Usually, `NoSettingsBidirectional` is chosen as super class. As alternative, `RecreateRevertBidirectional` can be chosen as super class, if the bidirectional operator allows to restore removed information, since `RecreateRevertBidirectional` provides some default support for configuring and executing this case (Section 6.5.2[☞214]).

automatically derive Configurations for Metamodel Decisions of the backward unidirectional Operator

In order to the ease the configuration effort of methodologists and to prevent conflicting configurations by accident, configurations for *metamodel decisions* must be provided by methodologists *only for the forward unidirectional operator*, since the configurations for the backward unidirectional operator can be automatically derived by proper design. These backward configurations could be derived directly from the forward configuration or are taken from the metamodel before or after executing the forward unidirectional operator for the first time, which can be done by implementing the methods `initializeImportBeforeFirstForwardExecution(...)` or `initializeImportAfterFirstForwardExecution(...)`.

Note, that for most bidirectional operators it is possible to implement an “inverse” bidirectional operator by switching the two unidirectional operators (Section 6.1.3[☞189]). As an example, the bidirectional operator ⇄AddDeleteAssociation is implemented by using →AddAssociation for the forward direction and its inverse unidirectional ←DeleteAssociation for the backward direction. An “inverse” bidirectional operator ⇄DeleteAddAssociation can be easily implemented by using →DeleteAssociation for the forward direction and using ←AddAssociation for the backward direction.

8.3.3 Java-API for Orchestration

Java API for configuring bidirectional Operators

To ease the work of methodologists, when they select, configure and combine bidirectional operators into a tree, a Java API is developed to declare the whole orchestration including API calls for each bidirectional operator. The use of this API by methodologists is

deepened in Section 12.1☞455. This API for configuring and combining operators can be seen as an internal DSL, i. e. a DSL embedded into a host general-purpose language (here: Java). Internal DSLs are used also by other related approaches, e. g. by Hinkel and Burger (2019) for an internal incremental BX approach.

Examples for the API calls for bidirectional Operators

An example for this API is given in Listing 8.1, whose Java source code is developed by the methodologist in order to configure an orchestration: The method `collectOperators()` is called by the MoConseMI framework and creates the orchestration. Here, the bidirectional operator ⇄RenameClassifier is selected and configured twice, leading to a chain of two configured bidirectional operators. The parameters are the configurations of the methodologist for the metamodel decisions of →RenameClassifier. Executing the first operator, →RenameClassifier renames the classifier with the fully-qualified name "pack.ClassType" to "DataType" in the metamodel. Since this renaming does not influence conforming models, →RenameClassifier does not define model decisions and therefore no configuration for model decisions are provided here. Executing the second operator afterwards, →RenameClassifier renames the (same renamed) classifier with the fully-qualified name "pack.DataType" to "Type" in the metamodel.

```
1 @Override
2 protected void collectOperators() {
3     // ...
4     renameClassifier("pack.ClassType", "DataType");
5     renameClassifier("pack.DataType", "Type");
6     // ...
7 }
```

Listing 8.1: Examples for the Java API to configure bidirectional Operators

at least one API Call for each bidirectional Operator

According to this idea, platform specialists have to implement an API call for the new bidirectional operator, which accepts configurations for all decisions by methodologists, according to the implementation of the bidirectional operator in Section 8.3.2☞270. Additionally, more than one API call for each bidirectional operator could be implemented in order to provide "shortcuts" for some operators by providing some default configurations for rarely used decisions.

8.4 Adapters

This section supports adapter providers with technical details, how to develop new adapters in order to bridge additional technical spaces with EMF as used by MoConseMI and to fulfill Requirement R 4 (Technical Spaces)☞158. Additionally, the following subsections introduce some predefined adapters, which can be reused by methodologists for all views in various projects.

Implementation Ideas from Related Work

There is some related research, which proposes some realization techniques for adapters: Dimitrieski (2017) proposes an approach for supporting more technical spaces, in particular, for Industry 4.0, which could be used to develop further adapters. Adapters could use model-to-text transformations (Rose, Matragkas et al., 2012) to represent the model of the view in textual representations. Excel and CSV (see below) show the need for having two-level to multi-level transformations (Atkinson, Gerbig and Tunjic, 2013b), since a model in classical modeling approaches is transferred into a model with two ontological levels in multi-level-modeling.

Guideline for implementing Adapters

Depending on the concrete implementation of the MoConseMI framework, there are some technical details for adapter providers in order to develop new adapters:

- All implementations for adapters must be sub-classes of `DataAdapter`, which defines the API of adapters as it is used in the orchestration and during the execution. Additionally, `DataAdapter` provides lots of internal support for managing data spread over multiple files and for ensuring unique UUIDs.

- The most important methods in the signature of `DataAdapter` are the following ones and must be adapted by new adapters according to the characteristics of the new technical space:

  - When the adapter is used for a data source, it must provide the initial model and metamodel during the initialization of the SU(M)M (Section 6.5.4 ☞219) via the signature `MigrationInformation initialize(...)`.
  - When the adapter is used for a new view(point), it gets the current model and metamodel for the first time during the initialization of the SU(M)M (Section 6.5.4 ☞219) via the signature `void initialize(MigrationInformation infos, ...)`.
  - During the change propagation, a user changes one view within its technical space. These user changes are provided by adapters via the signature `ChangeContainer loadUserChanges(...)` to MoConseMI.
  - The user changes are automatically propagated by MoConseMI to all views, resulting in execution differences for views (Section 6.5.3 ☞217). If the execution differences are not empty for a view, they are provided to its adapter via the signature `void applyChanges(ChangeContainer modelChangesToApply)`.

- Since all elements in models and metamodels must have stable UUIDs, adapters have to ensure a proper handling of UUIDs for all model and metamodel elements as well. If internal mechanisms for stable UUIDs are not sufficient, desired UUIDs could be made configurable for methodologists.

predefined Adapters

While this information helps adapter providers to develop new adapters for additional technical spaces, some adapters are already provided by MoConseMI and shortly presented here. Models and metamodels in EMF can be supported with predefined adapters for dynamic EMF (Section 8.4.1) and static EMF (Section 8.4.2 ☞273), according to the two modes of EMF (Section 2.5.3 ☞87). Data managed by users in spreadsheets can be used in MoConseMI by adapters for CSV (Section 8.4.4 ☞275) and Excel (Section 8.4.3 ☞273). In general, it is common to use data managed in spreadsheets also for model-based engineering, as the approaches of Cunha, Fernandes et al. (2012) and Francis, Kolovos et al. (2013) demonstrate. Section 8.4.5 ☞278 shortly presents the reuse of text which is structured with a grammar and conforms to a textual DSL in the technical space EMF.

8.4.1 Dynamically typed EMF

This adapter supports users who work on concrete renderings of views which are realized with dynamically typed EMF. This adapter is implemented in the `framework` Maven project in `de▸unioldenburg▸se▸mmi▸framework▸data▸source▸EmfDataAdapter`.

Configuration

The `EmfDataAdapter` gets paths for the metamodel in `*.ecore` files and for the model in `*.xmi` files. If the (meta)model is spread about multiple files, one instance of `PackageInfo` must be specified for each file, while their order might be important:

- Use the constructor `PackageInfo(String pathContent, String nsPrefix, String nsUri, String namespaceName, String namespaceUuid)` for *metamodel* files.

- Use the constructor `PackageInfo(String pathContent, String pathContentOutput, String namespaceName, String namespaceUuid)` for *model* files. For each model file its `namespaceName` and its `namespaceUuid` must be defined, since the UUIDs must be unique and the names are not contained in the model files!

If the adapter is used for new view(point)s, a `MetaModelFileProviderFilesSameFolder` could be used to store all root namespaces into single files into one folder instead of specifying lots of `PackageInfo`s. The parameter `String setMissingUuids` provides additional support for elements without UUIDs in data sources:

- set no missing UUIDs at all (value: `null`)
- generate default UUIDs for elements without UUIDs (value: `""` i. e. the empty String, which is the default option)
- use the value of a `EStructuralFeature` as UUID (value: the simple name of the `EStructuralFeature`)

8.4.2 Statically typed EMF

This adapter supports users who work on concrete renderings of views which are realized with statically typed EMF. This adapter is implemented in the `framework` Maven project in `de‣unioldenburg‣se‣mmi‣framework‣data‣source‣EmfStaticPackageAdapter`.

The difference between dynamically typed EMF and statically typed EMF is, that statically typed EMF uses generated Java classes as metamodel and represents objects in EMF models as Java objects with the generated Java classes as types, while dynamically typed EMF finally uses instances of `EObject` to represent objects in models and classes in metamodels (Section 2.5.3 ☞87). Using statically typed EMF has the benefit, that the generated Java source code could be extended with manually added functionality, e. g. for derived features.

static vs dynamic EMF

`EmfStaticPackageAdapter` for static EMF extends `EmfDataAdapter` for dynamic EMF by replacing the generated, specific `EPackages` with dynamic, generic `EPackages` in the metamodel. In the model, the types of objects and slots are replaced from the static `EClasses` and `EStructuralFeatures` to the corresponding dynamic types. Additionally, static enum literals in slots conforming to `EAttributes` with an enumeration as data type are replaced with the corresponding dynamic enum literals. When the model of a view which is realized with static EMF is updated due to changes in other views, this transformation is reversed, from dynamic metamodel elements to static metamodel elements.

replace static and dynamic Metamodel Elements with each other

Since the adapter for static EMF is an extended version of the adapter for dynamic EMF, the configurations of Section 8.4.1 ☞272 for the dynamic case are valid for the static case here as well. Additionally, the generated, static and specific `EPackage` of the view must be given to the adapter for static EMF.

Configuration

8.4.3 Excel

This adapter supports users who work on concrete renderings of views which are realized as spreadsheets with Excel. This adapter is implemented in the `framework` Maven project in `de‣unioldenburg‣se‣mmi‣framework‣data‣source‣ExcelAdapter` and uses the `excel2emf-meta-model` Maven project, which depends on the `excel2emf` Maven project.

In general, this adapter converts one (or more) Excel files into one EMF metamodel and into one conforming Edapt model, while configurations control the details of this transformation. The adapter extracts one metamodel in the EMF format from one header row in one sheet inside one Excel file. The adapter extracts one model in the Edapt

general Idea

format from multiple rows with content in one sheet inside one (or more) Excel files, following the (same) generated metamodel. This idea is also depicted in Figure 2.23 ☞91 in Part 13 ☞90 of the ongoing example for CSV, while this idea counts for all spreadsheets including CSV and Excel. In order to adapt these transformations, the Excel adapter provides lots of configurations. The Excel adapter transforms only content in Excel cells, no formatting and also no formulas. The Excel adapter could be extended with support for Excel formulas (Aivaloglou, Hoepelman and Hermans, 2017) for derived features.

`excel2emf` and `excel2emf-meta-model`

For the technical transformation between Excel and EMF, the project `excel2emf` is reused, which is developed by Säfken (2020) using the Java API of Apache POI to access Excel files (Figure 8.1 ☞264). Since `excel2emf` transforms one whole Excel file into one EMF model (conforming to a static metamodel representing the main features of Excel) without loosing information, it does not provide specific models and metamodels depending on the content of the Excel file, which is done by `excel2emf-meta-model`. The transformation from Excel to EMF as provided by `excel2emf-meta-model` covers only the information which is specified by configurations. All excluded sheets, rows and columns (and all formatting stuff) are *not* transformed into EMF and are lost, if the Excel file is deleted after the transformation. The transformation from EMF back into Excel merges the information currently available in EMF back into the Excel sheet(s), so that the information excluded from the transformation in the direction Excel-to-EMF is kept.

Metamodel for Configurations

In order to configure the Excel files, the header row for the metamodel, and the rows with content for the model, a flexible configuration is provided, which is realized again with EMF, since EMF provides means to load and save them easily in the file system. The metamodel for this configuration is shown in Figure 8.2 and some general ideas are sketched, while other details are skipped here.

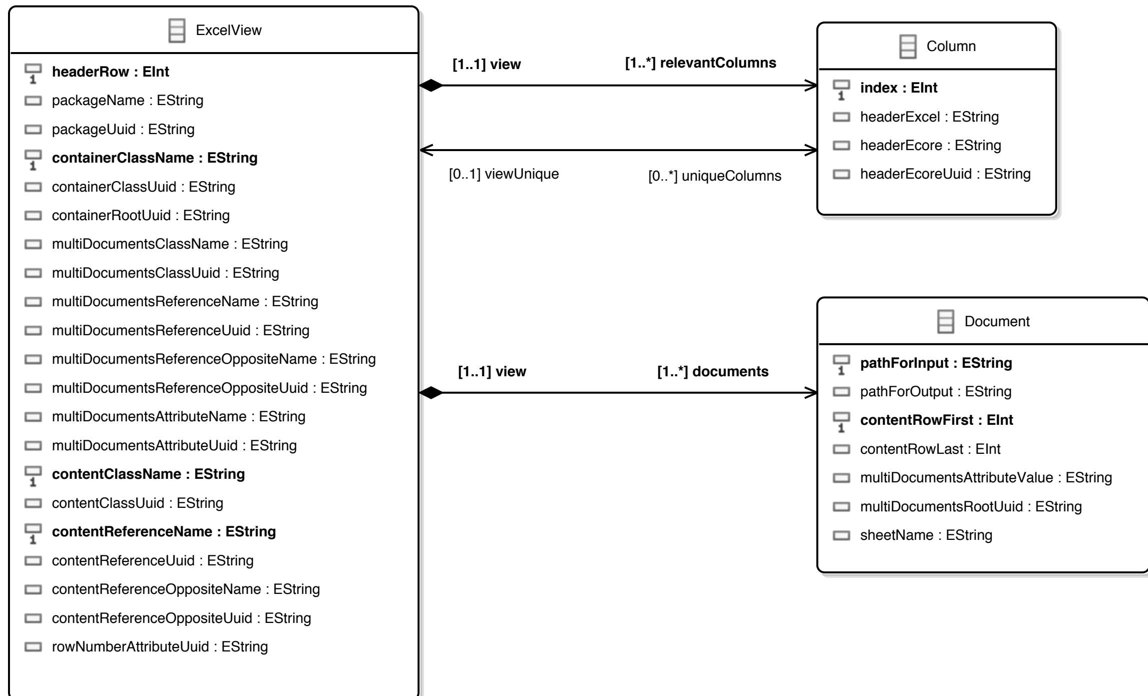


Figure 8.2: Metamodel for the possible Configurations of the Excel Adapter

Configuration

In order to apply the Excel adapter, an instance of `ExcelView` has to be created with one `Column` for each column in the Excel file to process (identified by the `index`) and one `Document` for each Excel file to extract rows from for the model. The other configurations allow methodologists to control some details of the transformations:

- The header row which determines the attributes in the metamodel must be the same in each used Excel file. Therefore, the configuration for the header row is located in `ExcelView.headerRow`.
- If more than one Excel document (with the same structure) should be transformed into the model, rows of different documents must be distinctable for the backward transformation. For that, a new attribute with the name `ExcelView.multiDocumentsAttributeName` is used, which gets the values of `Document.multiDocumentsAttributeValue`. There are two approaches, where to put this attribute and how to organize objects stemming from different documents:
  - This first (simple) approach is to add this additional attribute into the class which represents rows with content.
  - As (recommended) alternative, an "intermediate container" is used, which stores the rows of one Excel sheet. These intermediate containers are stored inside the root container. The intermediate container is configured with the other configurations starting with `multiDocuments*`.
- Inside an Excel file, only one of its sheets can be used. If there is only one sheet, `Document.sheetName` can be empty.
- Inside an Excel sheet, the lines with content are read starting with `Document.contentRowFirst` (including). If not all rows of the sheet should be read, the end can be specified with `Document.contentRowLast` (including), otherwise "-1" indicates to read all rows in the sheet. Note, that indices for rows and columns in Excel start with 1 (not with 0).
- Most of the other configurations allow to specify the desired names and UUIDs for the generated elements in the metamodel.

After doing the Excel-to-EMF-transformation, it is possible to change the resulting model, before doing the EMF-to-Excel-transformation in order to propagate changes in models back into Excel: For this backward transformation, it is important to identify and to map each object in the model with its corresponding Excel row. To identify corresponding rows and objects, keys are used for this identification. The chosen keys for identification must be stable during transformations and during changes in the model. Methodologists are supported with two possible strategies for stable identification keys:

UUIDs and transforming changed Models back into Excel

- One strategy is to store the row number in objects via an additional attribute with the restriction, that the row number must not be changed in the model.
- The other, more flexible and recommended strategy is to calculate the UUID of the object by composing some of its values (which are selected via `ExcelView.uniqueColumns`) with the restriction, that the composition of these values is unique for all objects. Since the UUID is stable, the used values (including the row number) can be changed in EMF, since the backward transformation takes the unchanged values in the Excel file for matching (unchanged) rows with the (unchanged) UUID of objects.

8.4.4 CSV

This adapter supports users who work on concrete renderings of views which are realized as spreadsheets with CSV. This adapter is implemented in the `framework` Maven project in `de▸unioldenburg▸se▸mmi▸framework▸data▸source▸CsvAdapter` and uses the `csv2emf`

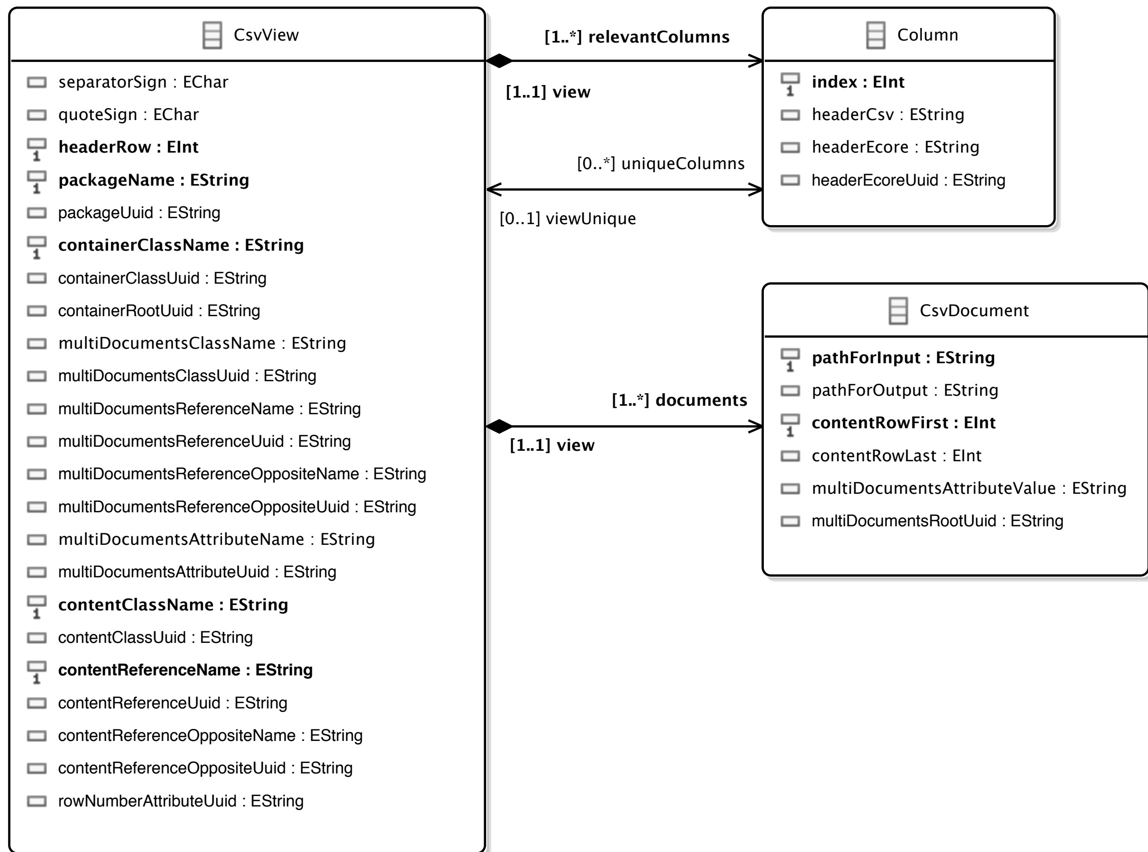


Figure 8.3: Metamodel for the possible Configurations of the CSV Adapter

Adapters for CSV and Excel are very similar

Maven project. Since CSV is also a spreadsheet format like Excel, most of the information for the Excel adapter counts for the Excel adapter as well.

Configuration

The possible configurations for CSV adapters are depicted in Figure 8.3. Most configurations of the Excel adapter count also for the CSV adapter, with the following exceptions:

- Row and column numbers start with 0.
- Since CSV documents contain only one sheet, there is no sheet name in CSV (sheet names are required only for Excel).
- All CSV files must be encoded using UTF-8.
- The characters for encoding separators and quotes can be configured using `CsvView.seperatorSign` and `CsvView.quoteSign`.

To make these configurations concrete, the following box shows an application of the CSV adapter for the ongoing example.

Ongoing Example, Part 24: **CSV Adapter for Requirements** ← List →

The CSV format is used to collect requirements for the requirements specification, as depicted in Figure 1.1☞38 in Part 5☞37 of the ongoing example for Requirements. In order to keep these requirements in CSV format consistent to the models of the other views, the methodologist applies the CSV adapter for Requirements and configures it by providing an instance of Figure 8.3.

The resulting metamodel is shown in Figure 8.4☞277. The most interesting parts of the configuration are sketched here. The configuration of the instance of `CsvView` contains the

following values:

- The `headerRow` is set to 0, since the CSV row 0 contains the headline of the data and its headers for the columns are used for the attributes in the metamodel (see below).
- The `contentClassName` is set to “RequirementsSpecification” and represents the container, which contains all `Requirements` (specified by `contentClassName`). Since no intermediate container for handling multiple CSV files is required, no values are given for the `multiDocument*` configurations.
- The role names for the containment `EReference` are specified with “container” for `contentReferenceOppositeName` and with “content” for `contentReferenceName`.

In order to extract all three columns of the CSV file (the first column in Figure 1.1 ☞38 visualizes only the row numbers), three instances of `Column` are configured, each with the index and the header of the column in the CSV file. The first column with the “ID” of the requirements is used for calculating stable UUIDs for the requirements objects in the model. These three columns result in the attributes `id`, `author` and `text` in `Requirement` in the metamodel with `EString` as default data type. The attribute `rowNumber` is always created in the metamodel by the CSV adapter.

As last step, the CSV file is configured with an instance of `CsvDocument`:

- The `pathForInput` points to the CSV file in the file system.
- The rows with content i. e. concrete requirements start at row 1 (value 1 for `contentRowFirst`) until the end of the CSV file (value -1 for `contentRowLast`).

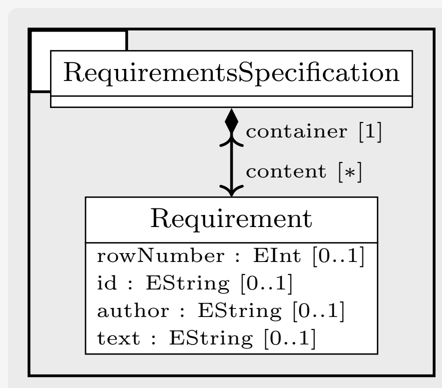


Figure 8.4: Metamodel for the data source Requirements

The resulting model is shown in Figure 8.5 ☞278. One object of type `RequirementsSpecification` is created, which contains one `Requirement` for each row with content in the CSV file. Accordingly, there are two `Requirements`, since the CSV file contains contains two rows starting at `contentRowFirst`. These row numbers are set as values for `rowNumber`. For the other attributes, the values from the CSV file are taken and put into the corresponding slots for each `Requirement`. The values for `id` are reused for the UUIDs of the `Requirements` objects.

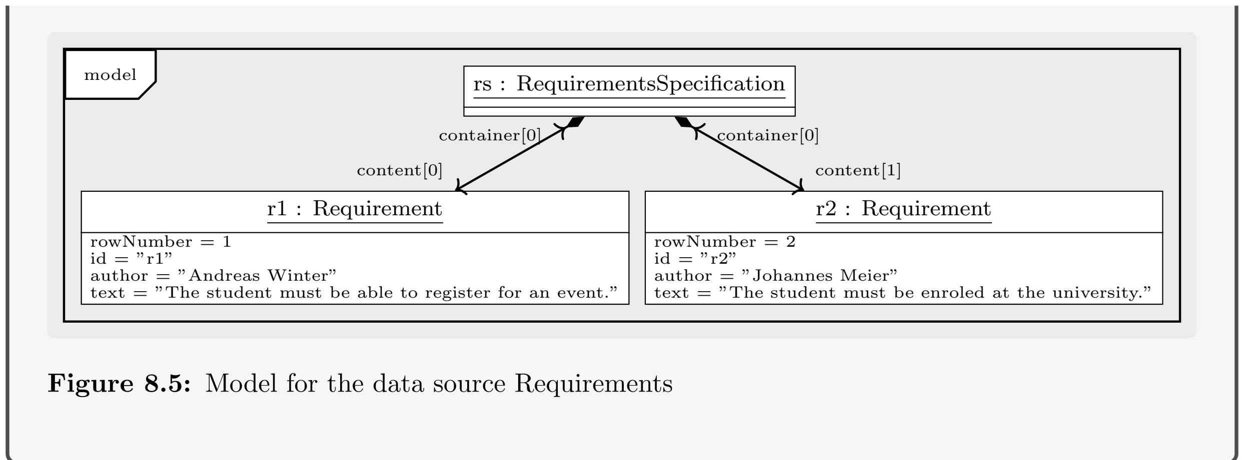


Figure 8.5: Model for the data source Requirements

8.4.5 Xtext

This adapter supports users who work on concrete renderings of views which are realized as structured text in form of a textual DSL with XTEXT. This adapter is implemented in the application of MoConseMI for managing access rights in Chapter 9 ☞283 and uses XTEXT libraries as dependencies. More details of XTEXT are provided in Chapter 9 ☞283.

XTEXT transforms structured text into EMF models and vice versa

Since XTEXT is realized for EMF as technical space, the adapter for XTEXT is a special adapter for dynamic EMF: When the EMF model is requested, the text conforming to the textual DSL is parsed by XTEXT and results in its desired representation as EMF model, which is transformed into an EDAPT model by the adapter for dynamic EMF. For the other direction, the current EDAPT model is transformed in an EMF model by the adapter for dynamic EMF and afterwards into the text conforming to the textual DSL by XTEXT. The required EMF metamodel is generated once by XTEXT.

Configuration

Therefore, the adapter for XTEXT is configured with paths and folders for the files containing structured text and containing the EMF (meta)models. Additionally, the DSL-specific transformers of XTEXT are given to the adapter.

Future Work: **More Adapters**

In order to manage data in further technical spaces with MoConseMI, additional adapters for additional technical spaces could be developed and existing adapters could be extended as future work:

- Relational *data bases* come with schema and instance data, which directly refer to metamodels and models, and could be supported by dedicated adapters.
- Adapters for additional formats for *spreadsheets* tables could be developed including for LibreOffice or Apple Numbers. Additionally, the adapters for spreadsheets could be extended with support for two dimensional tables.
- *Documentations* describe lots of information, which might need consistency support, therefore, adapters for documentation formats like LaTeX or Microsoft Word are useful. Developing adapters for documentation faces challenges to define project-specific metamodels for the documented information, since documents usually contain arbitrary, unstructured text without explicit schema.

8.5 Visualizations

This section supports methodologists to develop orchestrations and to supervise the executions of orchestrations. The MoConseMI framework applies the observer pattern to inform interested observers about certain situations and events during the configuration and execution of orchestrations. This infrastructure is used to collect runtime information about the orchestration and the current state of the execution, which are graphically visualized. Some visualizations are predefined by the MoConseMI framework in order to support methodologists. Additionally, the observer infrastructure allows to define custom visualizations. The MoConseMI framework supports the following kinds of observers:

supported Observers

OrchestrationObserver are informed about the configuration of the orchestration during the use case for initialization (Section 5.2.3 ☞ 176), in particular, observers are informed about new bidirectional operators and the end of the execution for the initialization of the SU(M)M. The main purpose of this observer is to be informed about the progress of the use case for initialization (Section 5.2.3 ☞ 176).

SynchronizationExecutionObserver are informed about the execution loops including their starts, ends, applied user changes and the remaining views to update. The main purpose of this observer is to understand the execution loop and the order in which views are updated.

OperatorExecutionObserver are informed about the steps to execute single unidirectional operators, including the calculation of branch differences and the execution of sub-operators. The main purpose of this observer is to get the current (meta)model and branch differences.

AdapterObserver are informed about certain situations during the work of adapters, including their initialization, user changes and applied execution differences. The main purpose of this observer is to get insights, when the adapters of views are used and when they transform information between technical spaces.

Some examples for visualizations, which are provided by the MoConseMI framework to support methodologists, are sketched here:

provided Visualizations

- Visualizations of the static structure of the orchestration with its configured bidirectional operators, like in Figure 6.11 ☞ 209, are implemented in `OrchestrationVisualization` and triggered by `InternalOrchestrationVisualizer`, notified as `OrchestrationObserver`.
- Visualizations of the execution loop and its order of updated models, like in Figure 6.18 ☞ 221, are implemented in `RunVisualization` and are triggered by `InternalRunVisualizer`, notified as `SynchronizationExecutionObserver`.
- Visualizations of the current (meta)models after each operator execution, like in Figure 5.2 ☞ 177,are implemented in `MetaModelVisualizationControllerTikzSingle`, notified as `OperatorExecutionObserver` and as `SynchronizationExecutionObserver`.
- Visualizations of the execution of a single unidirectional operator with the model before and the model after the operator execution, similar to visualizations in Walter and Ebert (2009), are implemented in `OperatorVisualizationSingleExecution` notified as `OperatorExecutionObserver`.

All visualizations are realized as LaTeX graphics with the Ti*k*Z package in order to include them into LaTeX documentations. Models and metamodels are rendered by the `emf-umlgraphmodel-visualization` project as "UmlGraphModel" (UGM), which is a generic

Infrastructure for realizing Visualizations

representation for class diagrams and object diagrams provided by the `uml-graph-model` project. This UGM model is transformed into a model-based representation for TikZ graphics, which is transformed into TikZ code as part of the `tikz-model` project (Figure 8.1 ☞ 264). This design allows to reuse transformation logic and eases creation and maintenance of visualizations.

8.6 Summary

This Chapter 8 ☞ 263 provides some insights into the implementation of the MoConseMI framework, which technically realizes the MoConseMI approach as designed in Chapter 6 ☞ 185. By realizing the MoConseMI framework as stand-alone Maven project without forcing methodologists to use a GUI, Requirement R 5 (Reusable Library) ☞ 158 is fulfilled (Section 8.1 ☞ 263). Section 8.2 ☞ 264 provides technical insights for methodologists and adapter providers, how to work with models and metamodels. Section 8.3 ☞ 267 provides guidelines, how to implement additional operators. Section 8.4 ☞ 271 supports adapter providers with technical hints for bridging additional technical spaces by adapters and sketches the predefined adapters of MoConseMI. Section 8.5 ☞ 279 supports methodologists with visualizations for the configuration and for the execution of orchestrations, supported by the observer infrastructure.

This implemented MoConseMI framework allows to apply MoConseMI in several application examples in Part IV ☞ 283 for evaluation. Limitations of this implementation are discussed in Section 14.3.2 ☞ 495. Possible extensions of this implementation are discussed as future work in Section 14.4.2 ☞ 497.

Part IV

Application

This part applies the MoConseMI framework in order to evaluate the applicability of MoConseMI in practice. The application examples show, how methodologists apply MoConseMI to configure orchestrations for ensuring project-specific consistency. These configurations are validated with acceptance tests, where users change one view and these changes are automatically propagated to all other views in order to keep them consistent to the changed view. In order to generalize these concrete applications, Chapter 12 ☞455 provides some guidelines for methodologists.

Chapter 9

Access Data

Distributed Access Management

This application applies MoConseMI to distributed access management. Objective of access management is to control the access of persons, users or systems to critical resources. This is done by managing users, resources and different rights like read-only or read-and-write. Later on, the access of users to parts of a subversion repository and folders on a web server is managed in simplified way.

In growing landscapes of information systems of companies, more and more such critical resources like applications or documents are upcoming. The same counts for users with potential access rights to those resources. If each critical resource comes with its own small procedure to manage the access to it, the access management is distributed without global view on all registered users and given access rights. Those local systems represent *data sources*, since they come with a fixed structure and existing access rights which must be kept. Systems like data bases managing registered users represent additional data sources.

distributed without global View

These data sources are overlapping, since same users might have access rights for different resources. This can lead to inconsistencies, e. g. when users are deleted incompletely and not all of their access rights are deleted in all systems, or when details of users are changed, like their user names, and these changes are not propagated to all systems managing access rights.

Since local access rights are already existing and it is not always possible to replace them, it makes sense to complement them with an access management which is global for e. g. the whole company. This requests for a synchronized access system in order to ensure, among others, that a user is deleted with all access rights. Therefore, a new view is required showing all given access rights for each user.

Because of the existence of different overlapping data sources which must be kept consistent to each other and the need for new views which allow to change the data sources, the sketched distributed access management is a senseful application for MoConseMI: *Users* are people who manage the different small systems for local access management. They add and remove single access rights. *Methodologists* are managers of the whole access management inside e. g. the company. They know all existing systems for access management and know, which groups of people with potential access rights exist.

outline

As representatives for existing systems for access management, Section 9.1 ☞ 284 introduces the used data sources in more detail and presents the developed SU(M)M and the new view(point). After that, the operators to form the SU(M)M are described in Section 9.2 ☞ 299. The definition of the new view(point) is shown in Section 9.3 ☞ 313. Along multiple scenarios with changes of the users, Section 9.4 ☞ 328 demonstrates, that the consistency between all data sources, the SUM and the new view is ensured by the configured operators. Section 9.5 ☞ 368 summarizes the findings and contributions of this application example.

This application bases on the works of Michel (2019), with improved and extended

consistency goals as well as with the new view(point). The orchestration is simplified on the one hand and extended with additional operators on the other hand, together with comprehensive documentations. Additionally, the role names in the UML diagrams for the models are hidden for an improved readability.

9.1 Application Domain

The domain consists of the data sources Htpasswd (Section 9.1.1), Authz (Section 9.1.2 ☞ 285) and Htaccess (Section 9.1.3 ☞ 289) as input. All relevant information for the domain are contained in the SUM (Section 9.1.4 ☞ 290). Parts of the SU(M)M are represented in the new view(point) Overview (Section 9.1.5 ☞ 292).

9.1.1 DataSource Htpasswd

Htpasswd: users + password hashes

The first data source is shown in Listing 9.1. Its purpose is to manage users with their passwords in form of hashes. The content is usually stored in a text file called `htpasswd` and is maintained by the command-line tool `htpasswd` which is part of the Apache HTTP Server project (The Apache Software Foundation, 2020b). Therefore, this first data source is called Htpasswd.

The initial concrete syntax before the initialization of Htpasswd is shown in the format Htpasswd in Listing 9.1.

```
1 #hp1
2 alice : theu9Naig7fophed #u1
3
4 bob : eengohbu4naisai7 #u2
5
6 frank : Ahch9iemai4Ui3si #u3
7 walter : eifeiho3iex3ahng #u4
```

Listing 9.1: The initial input of Htpasswd in Htpasswd format

Each non-empty row start with the user name of a user, followed by a colon, and ends with the hashed password of the user. `htpasswd` supports different hashing algorithms like MD5 or SHA1. Comments start with # and end at the end of the current line. The amount of white space does not matter.

Therefore, the example defines four users (alice, bob, frank and walter) with their password hashes. The comments are used as identifiers later.

Xtext

To bring the structured text into the technical space of EMF, an EBNF-based grammar is defined for Htpasswd files and realized with Xtext (Bettini, 2013). Xtext is chosen, because it generates a corresponding EMF metamodel and generates transformations from text to an EMF model and vice versa.

The metamodel of Htpasswd is shown in Figure 9.1 ☞ 285.

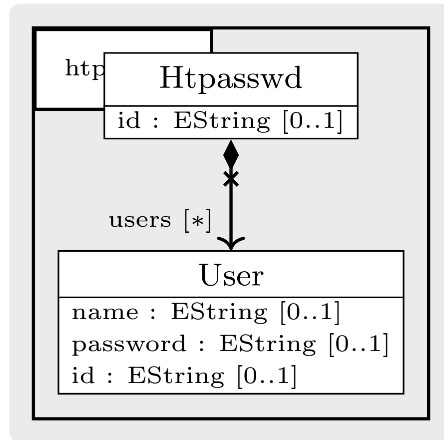


Figure 9.1: Metamodel of Htpasswd

Htpasswd represents a whole htpasswd-file containing an arbitrary number of users. Each User has a name and a password hash, which are both String values. The multiplicities 0..1 are generated by XTEXT, nevertheless, a name and a password are always mandatory.

The initial input model of Htpasswd is shown in Figure 9.2.

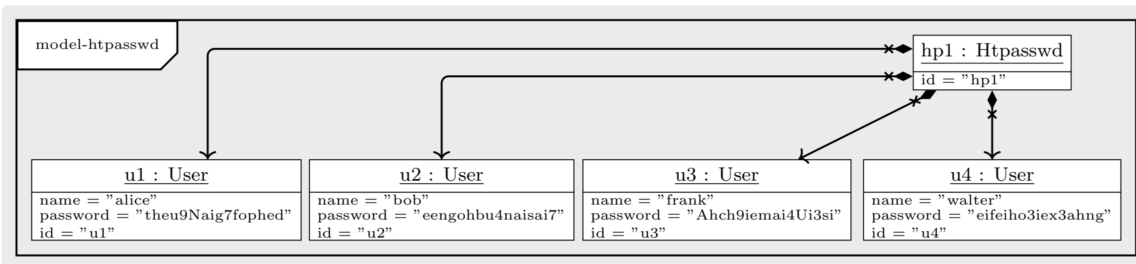


Figure 9.2: The initial input model of Htpasswd

The root object of type Htpasswd contains the four User instances with the names and password hashes from the htpasswd-file.

Since switching between technical spaces is not the main objective of this thesis, the details of the used grammar and its technical realization with XTEXT are skipped here, but are described by Michel (2019). More important is the handling of required UUIDs: Since each instance in the model must have a UUID, each type in the metamodel has an additional String-value attribute named id. Since the concrete UUIDs must be stable while switching between the technical spaces, the values for UUIDs are stored as comments in the htpasswd-file and used as values for the attribute, which is automated by XTEXT. These attribute values are used as object identifiers, which is realized by the developed adapter for XTEXT. As an example, the user “alice” has “u1” as value for its id-slot which is also used as object identifier in Figure 9.2. In Listing 9.1 ☞284, “u1” is stored as comment in the text line of “alice”.

UUIDs with XTEXT

All used data are artificial, created for the purpose of demonstration. In particular, the shown password hashes are not hashed from real passwords, but randomly generated strings.

9.1.2 DataSource Authz

The second data source is shown in Listing 9.2 ☞286. Its purpose is to manage access rights of users for Subversion (SVN) repositories (Collins-Sussman, Fitzpatrick and Pilato,

Authz: path-based access rights (Subversion)

2011). These access rights distinguish between read and write access and are given for single directories. The content is usually maintained by hand and stored in a text file called `authz`. Therefore, this second data source is called Authz.

The initial concrete syntax before the initialization of Authz is shown in the format Authz in Listing 9.2.

```
1  #a1
2  [/some/path/to/repo/r1] #r1
3  alice = rw #m1
4
5  [/some/path/to/repo/r2] #r2
6  alice = rw #m2
7  bob = rw #m3
8
9  [/some/path/to/repo/r3] #r3
10 charlie = r #m4
11 eric = rw #m5
```

Listing 9.2: The initial input of Authz in Authz format

Access rights are always bound to one directory (and its child directories, if not overridden), which is defined by the path surrounded by brackets in one line: The user with access to this directory are added afterwards, one user per line, with the user name, followed by the equal sign and the given access mode. Supported access modes are read-only (r) and read-and-write (rw).

The example is giving access rights for three directories (r1, r2, r3, all located in inside the directory /some/path/to/repo/): The user "alice" got read-and-write access to the directories r1 and r2, but no access to r3. The user "charlie" got read-only access to only the directory r3. The users "bob" and "eric" got also read-and-write access rights to r2, respectively r3. The comments are used as identifiers later.

Subversion supports much more features like groups of users, but they are not supported here (Collins-Sussman, Fitzpatrick and Pilato, 2011). Again, XTEXT is used to bring these data into the technical space EMF with the same strategy of handling stable UUIDs.

The metamodel of Authz is shown in Figure 9.3☞287.

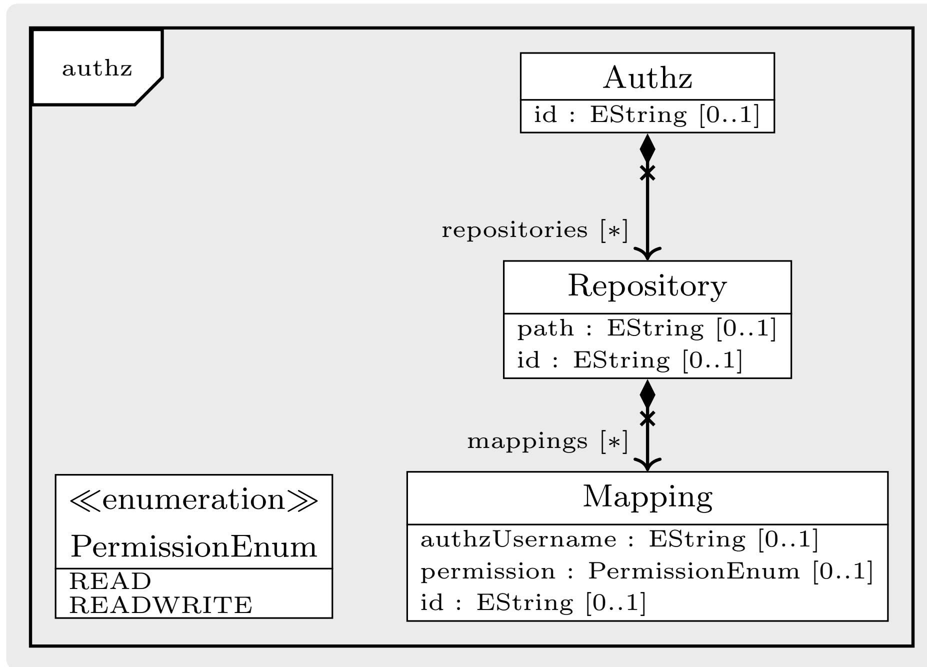


Figure 9.3: Metamodel of Authz

`Authz` represents as root the whole authz-document, in which multiple directories and their enabled users are managed: `Repository` represents one directory, whose path is stored in the `path`-attribute. Each repository has an arbitrary number of `Mapping`s which are giving one `permission` to one user, represented by its user name in the `authzUsername`-attribute. The possible permissions are modeled with the enum `PermissionEnum`.

The initial input model of Authz is shown in Figure 9.4 ☞288.

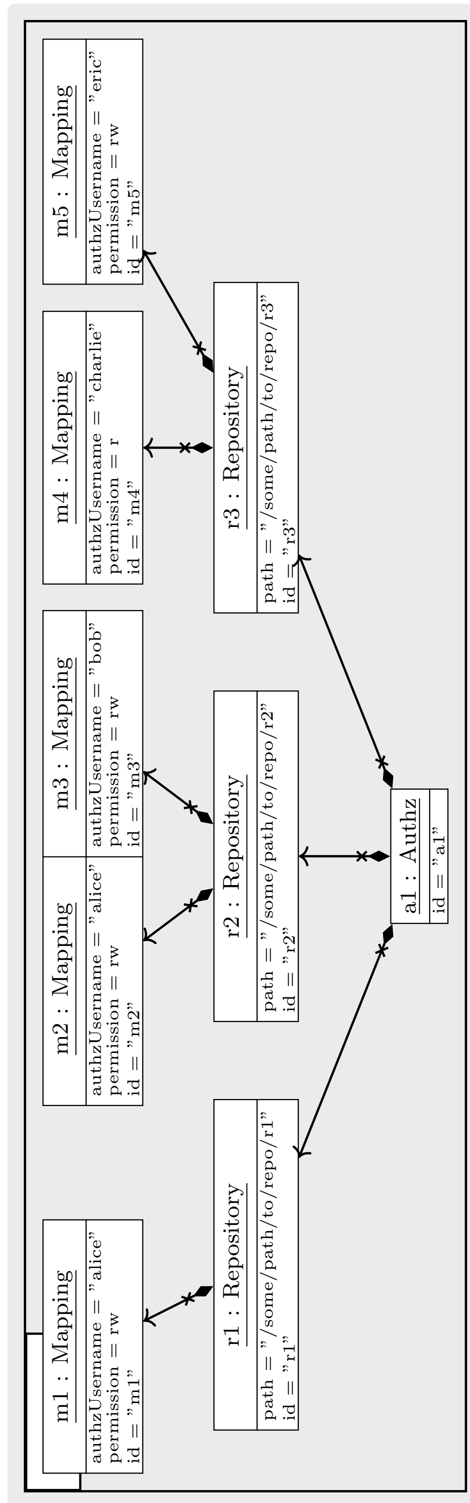


Figure 9.4: The initial input model of Authz

The root object of type `Authz` stores the three defined directories as instances of `Repository` with their paths. Each repository holds one `Mapping` instance for each user with access to it. Therefore, there are two mappings with the user name “alice”, connected to the two different directories.

All used data are artificial, created for the purpose of demonstration.

9.1.3 DataSource Htaccess

Htaccess: access to directory (Webserver)

The third data source is shown in Listing 9.3. Its purpose is to manage access rights of users for one directory inside a Apache HTTP Server (The Apache Software Foundation, 2020a). The content is usually maintained by hand and stored in a text file called `.htaccess` which is located inside the protected directory. Therefore, this third data source is called Htaccess.

The initial concrete syntax before the initialization of Htaccess is shown in the format Htaccess in Listing 9.3.

```
1 #ha1
2 AuthUserFile /some/path/to/userFile #s1
3 AuthGroupFile /some/path/to/groupFile #s2
4 AuthName "Some Name For This Auth -- Restricted Access" #s3
5 AuthType Basic #s4

7 require user alice #hu1
8 require user dave #hu2
```

Listing 9.3: The initial input of Htaccess in Htaccess format

A `.htaccess` files starts with some initial configurations (lines 1–4). Here, only AuthUserFile is interesting, since it points to the `htpasswd`-file of Htpasswd and accepts only users which are defined there. After the initial configurations, all users with access are listed (lines 6–7) within their own line, starting with "require user" and ending with the their user names. The shown `.htaccess` example file specifies access exactly for the users with the user names "alice" and "dave".

In general, some more features like groups are possible in `.htaccess`-files, but they are not supported here. Again, XTEXT is used to bring these data into the technical space EMF with the same strategy of handling stable UUIDs.

The metamodel of Htaccess is shown in Figure 9.5.

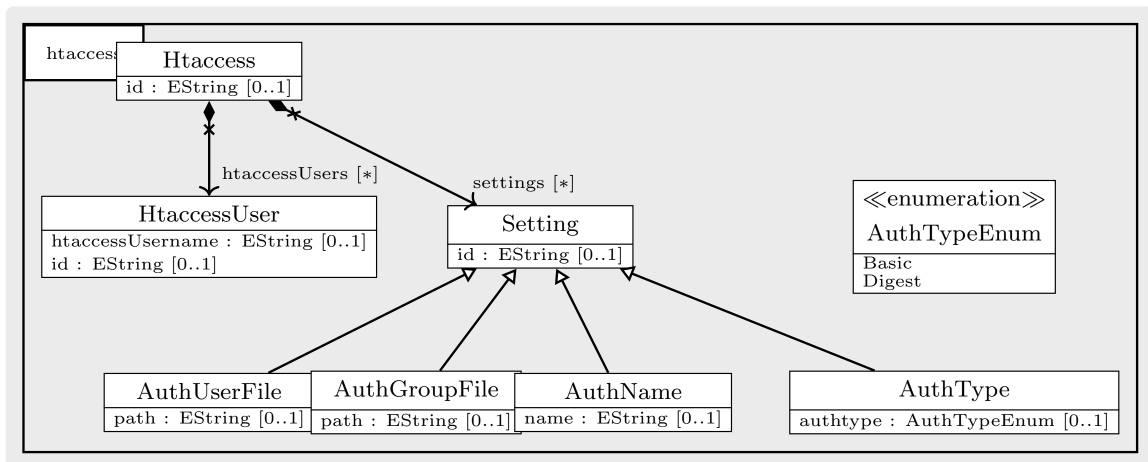


Figure 9.5: Metamodel of Htaccess

`Htaccess` is the root representing one `.htaccess`-file, which contains an arbitrary numbers of settings and allowed users. There are four different types of `Setting`s, namely `AuthzUserFile`, `AuthzGroupFile`, `AuthzName` and `AuthzType`, having paths or names or further properties like specified by the enumeration `AuthzTypeEnum`. More interesting are the users, represented by the type `HtaccessUser`, which stores the user name in the attribute `htaccessUsername`.

The initial input model of Htaccess is shown in Figure 9.6.

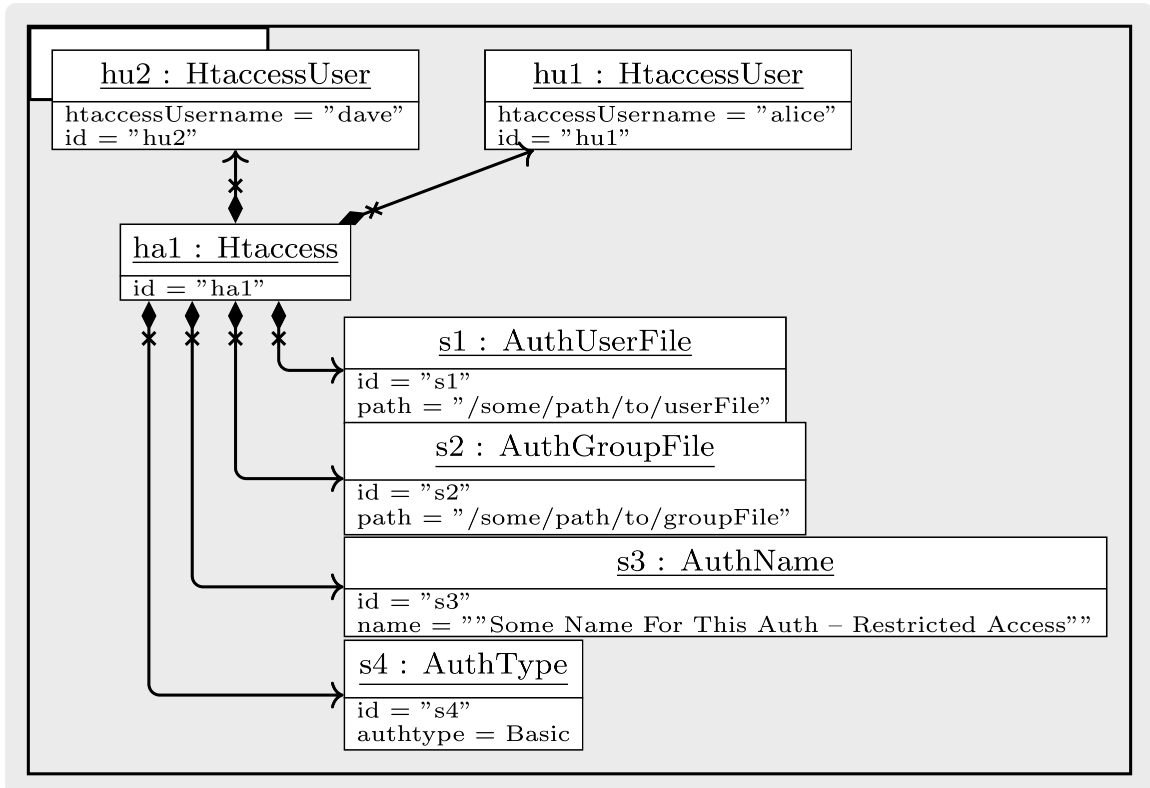


Figure 9.6: The initial input model of Htaccess

The root object of type `Htaccess` (ha1) stores the four settings (s1, s2, s3, s4) and the two users as instances of `HtaccessUser`. Their user names "alice" (hu1) and "dave" (hu2) are stored as `htaccessUsername`. All used data are artificial, created for the purpose of demonstration.

9.1.4 SU(M)M

Initially, there is no SU(M)M, but it is created for the first time during the initialization by executing the configured operators with the introduced data sources as starting point. The configurations of the operators control the final structure of the SU(M)M, which are documented in detail in Section 9.2 ☞ 299. This section serves as look-ahead and is useful for understanding the new view(point). The metamodel of SUMM is shown in Figure 9.7 ☞ 291.

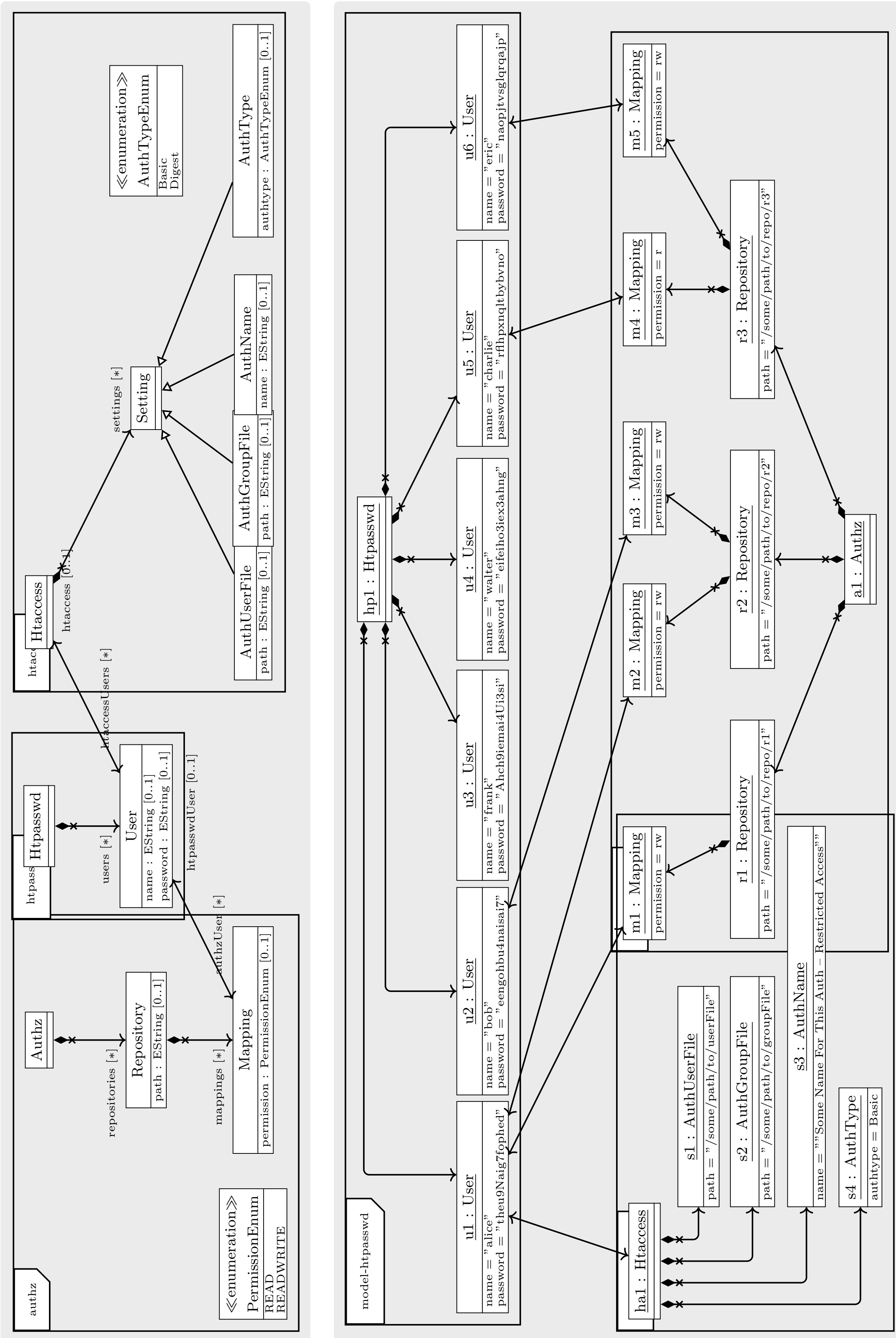


Figure 9.7: Metamodel (left/top) and the final model after the initialization (right/bottom) of SUMM

The three namespaces `htpasswd`, `authz` and `htaccess` of the three data sources are still existing in the SUMM and indicate the origins of the contained classes. Additionally, some integrations are visible in form of associations crossing the bounds of these namespaces, e. g. between the classes `Mapping`, `User` and `Htaccess`. Other content like the class `HtaccessUser` is missing in the SUMM, since it was removed as redundancies. The details of this integration are explained later.

The final model after the initialization of SUMM is shown in Figure 9.7 ☞291.

Corresponding to the SUMM, also the SUM still contains the namespaces of the three data sources on model level. There are lots of links connecting the namespaces with each other, since users in `model-htpasswd` are connected with their access rights in `model-authz` and `model-htaccess`. Names of users with multiple access rights like “alice” appear only once, according to the removal of redundancies.

9.1.5 New ViewPoint Overview

The new view called Overview addresses the problem, that there is no holistic summary about all given access rights for all users, since the data sources Authz and Htaccess target only one kind of access rights. For the needs of an access rights engineer, an EXCEL table is ideal, since it abstracts from the concrete technical realization.

The final concrete syntax after the initialization of Overview is shown in the format Excel in Figure 9.8.

A1 | fx name

	A	B	C	D	E	F	G	H
1	name	authzReadOr	authzWrite	htaccess	id			
2	alice	0	2	TRUE	u1			
3	bob	0	1	FALSE	u2			
4	charlie	1	0	FALSE	u5			
5	eric	0	1	FALSE	u6			
6	frank	0	0	FALSE	u3			
7	walter	0	0	FALSE	u4			
8								
9								

Figure 9.8: The final concrete syntax of Overview in Excel format

Each user gets one row in the table with name (column A) and ID (column E). Column “htaccess” (D) contains “TRUE”, if the user has the htaccess-right, and “FALSE” otherwise. If these values are changed by the user, the corresponding access right is added or removed. The columns “authzReadOnly” (B) and “authzWrite” (C) contain the numbers of given access rights for each user. Changes in these columns are ignored.

The metamodel of Overview is shown in Figure 9.9 ☞293.

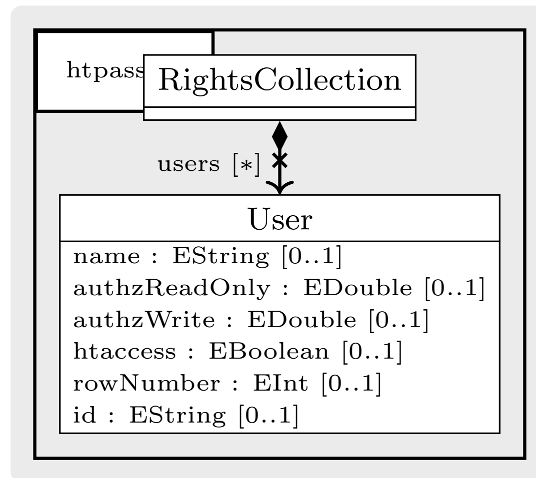


Figure 9.9: Metamodel of Overview

The general structure of this metamodel is determined by the adapter for EXCEL (Section 8.4.3☞273): `RightsCollection` represents the whole EXCEL file and contains one `User` for each row. The headers of the columns correspond to the attributes of `User` with same names. Additionally, the attribtue `rowNumber` stores the current number of the corresponding row (row numbers start with one). The `id` is used to identify users in rows uniquely. The attributes `authzReadOnly` and `authzWrite` store numbers of access rights and are integers, but are represented as doubles in the metamodel, because the transformations between EXCEL and EMF support only double for numerical values.

The final model after the initialization of Overview is shown in Figure 9.10☞294.

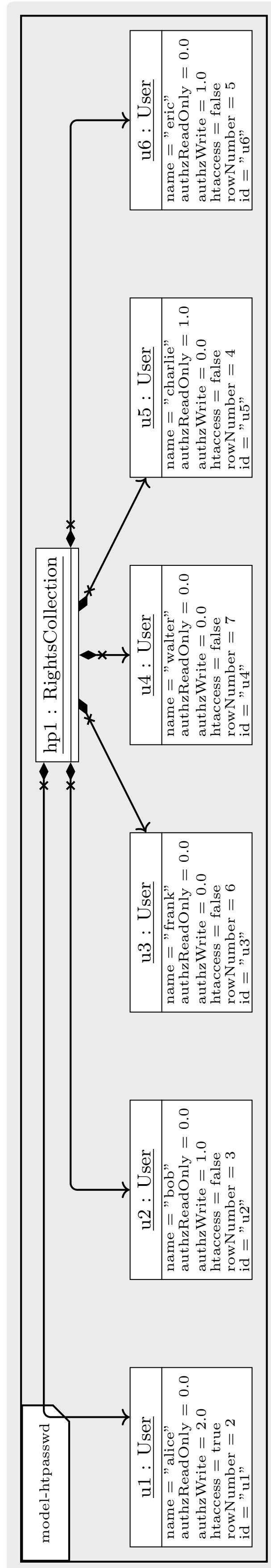


Figure 9.10: The final model after the initialization of Overview

The root object hp1 with type `RightsCollection` contains one `User` instance for each row representing a user in Excel (u1, u2, u3, u4, u5, u6). The values of the cells in Excel are transferred into the corresponding slots of the instances.

9.1.6 Realization Overview

Figure 9.11 presents an overview about all consistency goals, annotated along the edges. Hyperlinks at the consistency goals allow to jump to their introductions. The nodes in the graphic represent the data sources in this application. Hyperlinks at the nodes allow to jump to their introductions.

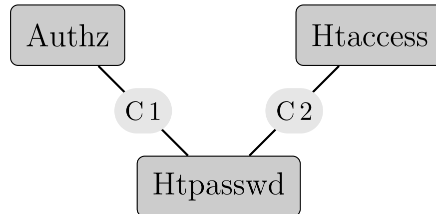


Figure 9.11: Overview about Consistency Goals in Access Data Management

The consistency is described by the following consistency goals and their concretizing consistency rules. Figure 9.11 visualizes the consistency goals and their involved data sources or new viewpoints.

Consistency Goal **C 1** — Htpasswd + Authz

User with access rights given by Authz must be registered in Htpasswd.

While Authz specifies the concrete access rights for users, their credentials are defined in Htpasswd. The credentials contain their hashed password, which is stored only once, but used for all rights in Authz. The mapping between Authz and Htpasswd is done by matching user names. Users in Htpasswd without any rights in Authz are allowed.

Consistency Rule **C 1 a** — for C 1

If the user of an access right in Authz is not defined in Htpasswd, a corresponding user is added in Htpasswd with a random password.

Since each access right must be related to a user as required by C 1, missing users are added in Htpasswd. This might lead to security issues, which are minimized, since the random password must be known to the person owning the access right. This consistency rule is explicitly tested by the test cases documented in Section 9.4.4☞ 340 and Section 9.4.5☞ 344.

Consistency Rule **C 1 b** — for C 1

If an access right is removed and its related user has no access rights anymore, that user is also removed.

Usually, only the access right is removed. If this access right was the only one for its user, the user is removed to keep the user management clean from unused users. This consistency rule is explicitly tested by the test cases documented in Section 9.4.3☞ 336 and Section 9.4.6☞ 348.

Consistency Rule C 1 c — for C 1

If a user is removed in Htpasswd, all its access rights in Authz are removed.

Removing a user indicates, that the whole user is to be removed, including all its access rights.

Consistency Rule C 1 d — for C 1

If the user name of an access right is changed, the corresponding user is renamed accordingly, if the user has not more than this access right. Otherwise, a new user with the new user name is created.

This ensures, that other access rights remain unchanged. The new user gets a random password, as in C 1 a. This behavior is analogous to C 1 b. The easier alternative is to rename the corresponding user in each case. This consistency rule is explicitly tested by the test case documented in Section 9.4.2 ☞ 332.

Consistency Rule C 1 e — for C 1

If a user is renamed, all its rights in Authz are renamed accordingly.

Since the user name is used and required for mapping to its access rights, renamings of the user must be applied also to all its access rights.

Consistency Goal C 2 — Htpasswd + Htaccess

User with access rights given by Htaccess must be registered in Htpasswd.

While Htaccess specifies the concrete access rights for users, their credentials are defined in Htpasswd. The credentials contain their hashed password, which is stored only once, but used for all rights in Htaccess. The mapping between Htaccess and Htpasswd is done by matching user names. Users in Htpasswd without any rights in Htaccess are allowed.

Consistency Rule C 2 a — for C 2

If the user of an access right in Htaccess is not defined in Htpasswd, this access right is removed in Htaccess.

Since each access right must be related to a user as required by C 1, missing users lead to invalid access rights. This is in contrast to C 1 a to show an alternative solution and can be rated as a more secure solution.

Consistency Rule C 2 b — for C 2

If an access right is removed, only this access right is removed, while the related user remains.

Only the access right is removed. Even users without any access rights anymore are kept, showing an alternative solution contrasting C 1 b.

Consistency Rule C 2 c — for C 2

If a user is removed in Htpasswd, all its access rights in Htaccess are removed.

Removing a user indicates, that the whole user is to be removed, including all its access rights.

Consistency Rule C 2 d for C 2

If the user name of an access right is changed, the corresponding user is renamed accordingly.

This ensures, that user and Htaccess rights still fit together. This behavior shows an alternative to C 1 d. This consistency rule is explicitly tested by the test case documented in Section 9.4.7 ☞ 352.

Consistency Rule C 2 e for C 2

If a user is renamed, all its rights in Htaccess are renamed accordingly.

Since the user name is used and required for mapping to its access rights, renamings of the user must be applied also to all its access rights.

All operators configured for the realization are visualized in Figure 9.12 ☞ 298 along the edges. Data sources are rendered as white rectangles. New viewpoints are rendered as gray rectangles. Intermediate (meta)models are rendered as black circles. The operators for the integration of the data sources into the SU(M)M are described in Section 9.2 ☞ 299. The operators to derive the new view(point) from the SU(M)M are described in Section 9.3 ☞ 313.

Figure 9.12: Configured Tree of Operators for Access Data Management

9.2 Integration of existing Data Sources

This section documents, how the existing data sources are integrated into the SU(M)M. For each used operator, its impact is highlighted and its configuration is sketched. The changes of the operator within the current metamodel are graphically visualized. The changes of the operator within the current model are graphically visualized. Only the combination of two (meta)models is not shown, since only two (meta)models are combined into one (meta)model on technical level without semantic changes.

In order to improve the readability, the documentation focuses on the most important information about the operators: For brevity, only the forward unidirectional operator is depicted in detail, while its inverse unidirectional operator is only mentioned. For some neighbored operators which realize similar objectives or work together for the same objective, the visualization of their impact is combined into a single graphic. Configurations for model decisions which are predefined by the operator and reused here, are not repeated again.

Htpasswd ⟷ 01: DeleteAddAllIDAttributes

This operator removes the IDs which are used only internally to prevent their change by users. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.13.

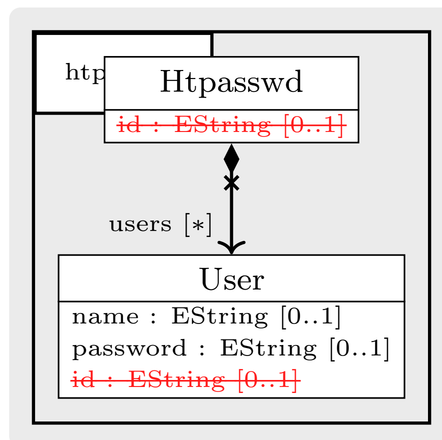


Figure 9.13: Metamodel Changes from Htpasswd to 01

Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.14.

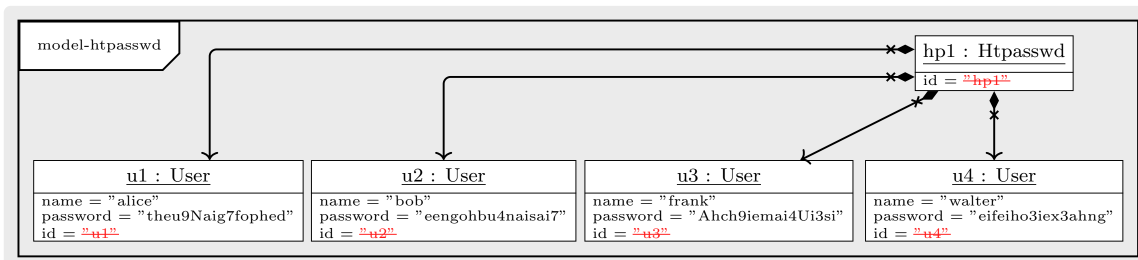


Figure 9.14: Model Changes from Htpasswd to 01

For the direction Htpasswd→01, the unidirectional operator is →DeleteAllIDAttributes (id).

Metamodel Decisions are not used by →CompositeOperator.

Model Decisions configured for →CompositeOperator: This operator has no configurations for model decisions.

For the inverse direction [Htpasswd]←01, the unidirectional operator is ←AddAllIDAttributes (id).

[Authz] ⟷ 02: DeleteAddAllIDAttributes

This operator removes the IDs which are used only internally to prevent their change by users. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.15.

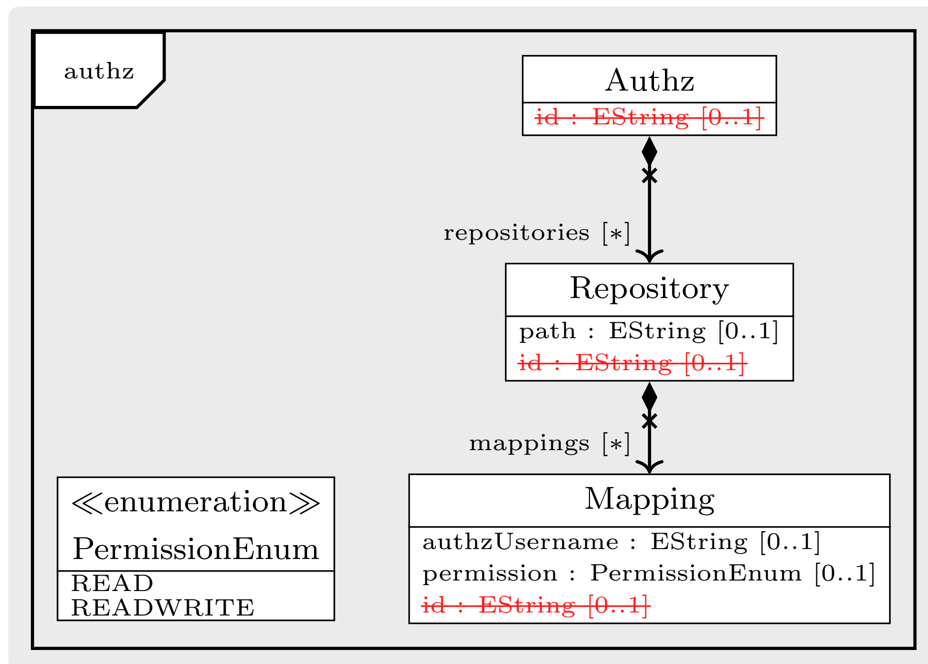


Figure 9.15: Metamodel Changes from [Authz] to 02

Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.16☞301.

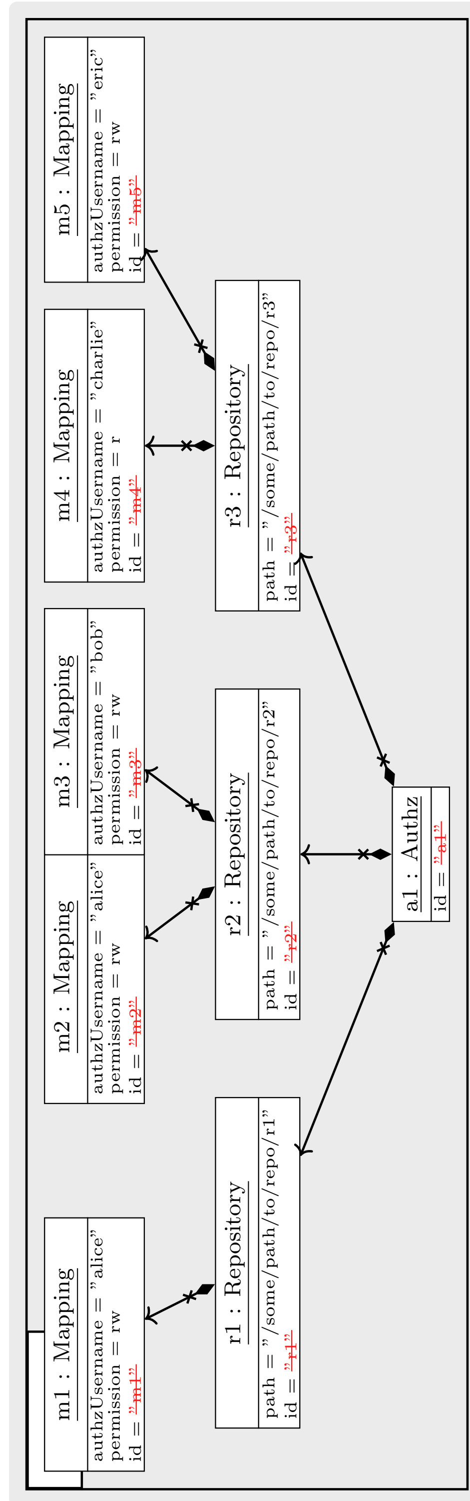


Figure 9.16: Model Changes from [Authz] to ⓿②

For the direction [Authz]→⓿②, the unidirectional operator is →DeleteAllIDAttributes (id).

Metamodel Decisions are not used by →CompositeOperator.

Model Decisions configured for →CompositeOperator: This operator has no configurations for model decisions.

For the inverse direction Authz←02, the unidirectional operator is ←AddAllIDAttributes (id).

9.2.1 Integrate Htpasswd and Authz

Integrates Htpasswd and Authz regarding overlapping user names.

03 ⟷ 04: AddDeleteAssociation

This operator relates Authz rights and users explicitly to each other for C 1. In case of a missing matching user, a new user is created in Htpasswd, according to C 1 a. By remembering all links, the corresponding user of a deleted access right can be identified and removed, if wanted by C 1 b. Removed users are detected in the same way, which leads to a deletion of its access rights, according to C 1 c. Renamings according to C 1 d and C 1 e are also realized by remembering links and checking the current changes. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.17 ☞ 303. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.17 ☞ 303.

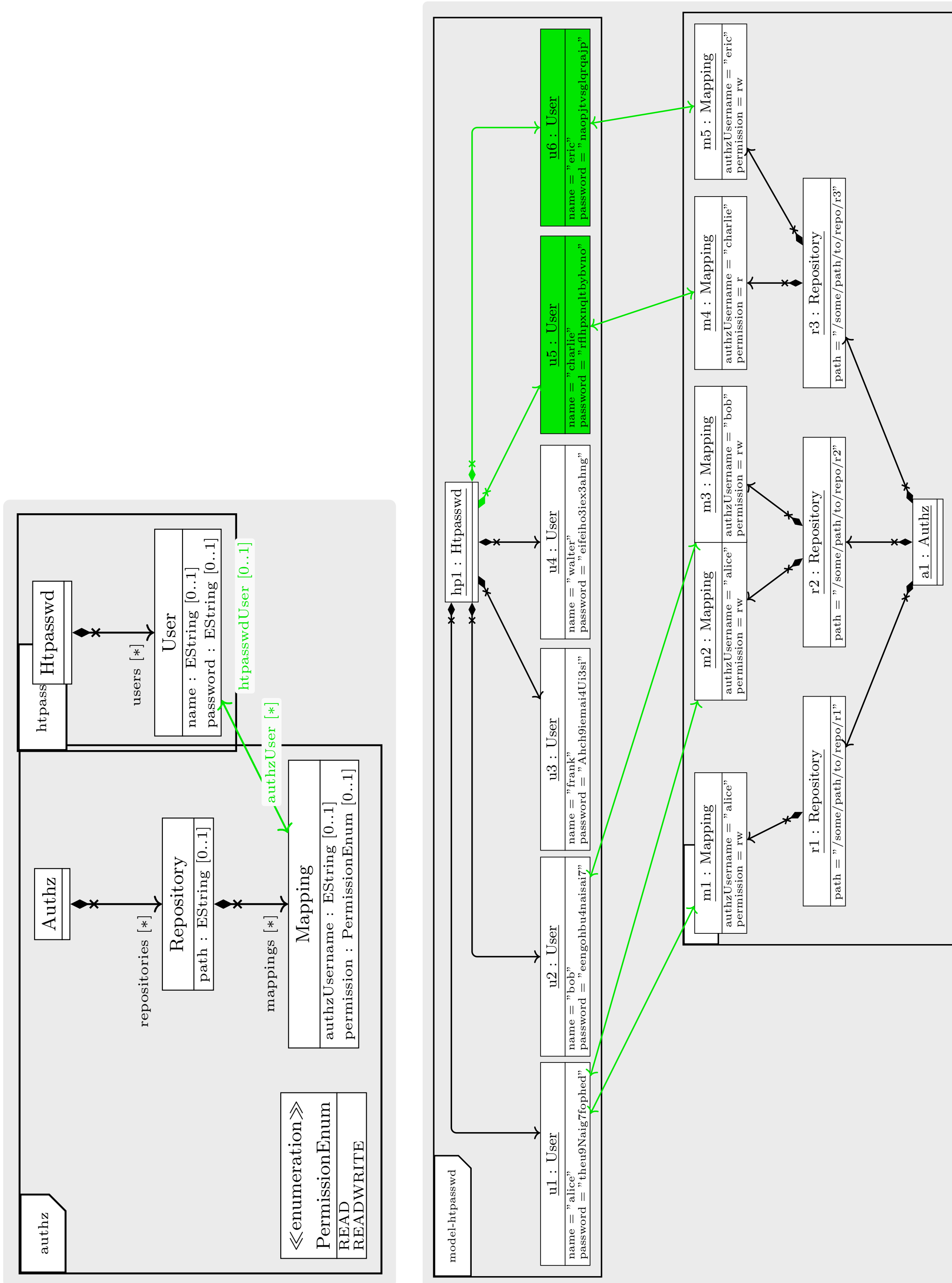


Figure 9.17: Changes in Metamodel (left/top) and Model (right/bottom) from ❸ to ❹

For the direction ❸→❹, the unidirectional operator is →AddAssociation (authz.Mapping.htpasswdUser).

Metamodel Decisions configured for →AddAssociation:

- firstClassNameFullyQualified = authz.Mapping
- secondClassNameFullyQualified = htpasswd.User

- firstEndName = authzUser
- firstEndLowerBound = 0
- firstEndUpperBound = -1
- firstEndComposition = false
- secondEndName = htpasswdUser
- secondEndLowerBound = 0
- secondEndUpperBound = 1
- secondEndComposition = false

Model Decisions configured for →AddAssociation: Configurations for model decisions are realized in `de▸unioldenburg▸se▸mmi▸rights▸integration▸decisions▸AddAuthzUserRelationDecision`. All their configurations for model decisions are listed here:

- createLinks (arg0 : `AddReference`, arg2 : `EReference`)
  Manages links between Authz rights and users by first exploiting the remembered historic links and second find new mappings based an matching names.

For the inverse direction 03←04, the unidirectional operator is ←DeleteAssociation (authz.Mapping.htpasswdUser).

04 ⟷ 05: DeleteAddAttribute

By removing the user name from Authz, redundancy is reduced. The user name is still stored in Htpasswd, which allows mappings to Htaccess later on. This produces a cleaner SU(M)M and prevents inconsistencies in the SUM. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.18 ☞305. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.18 ☞305.

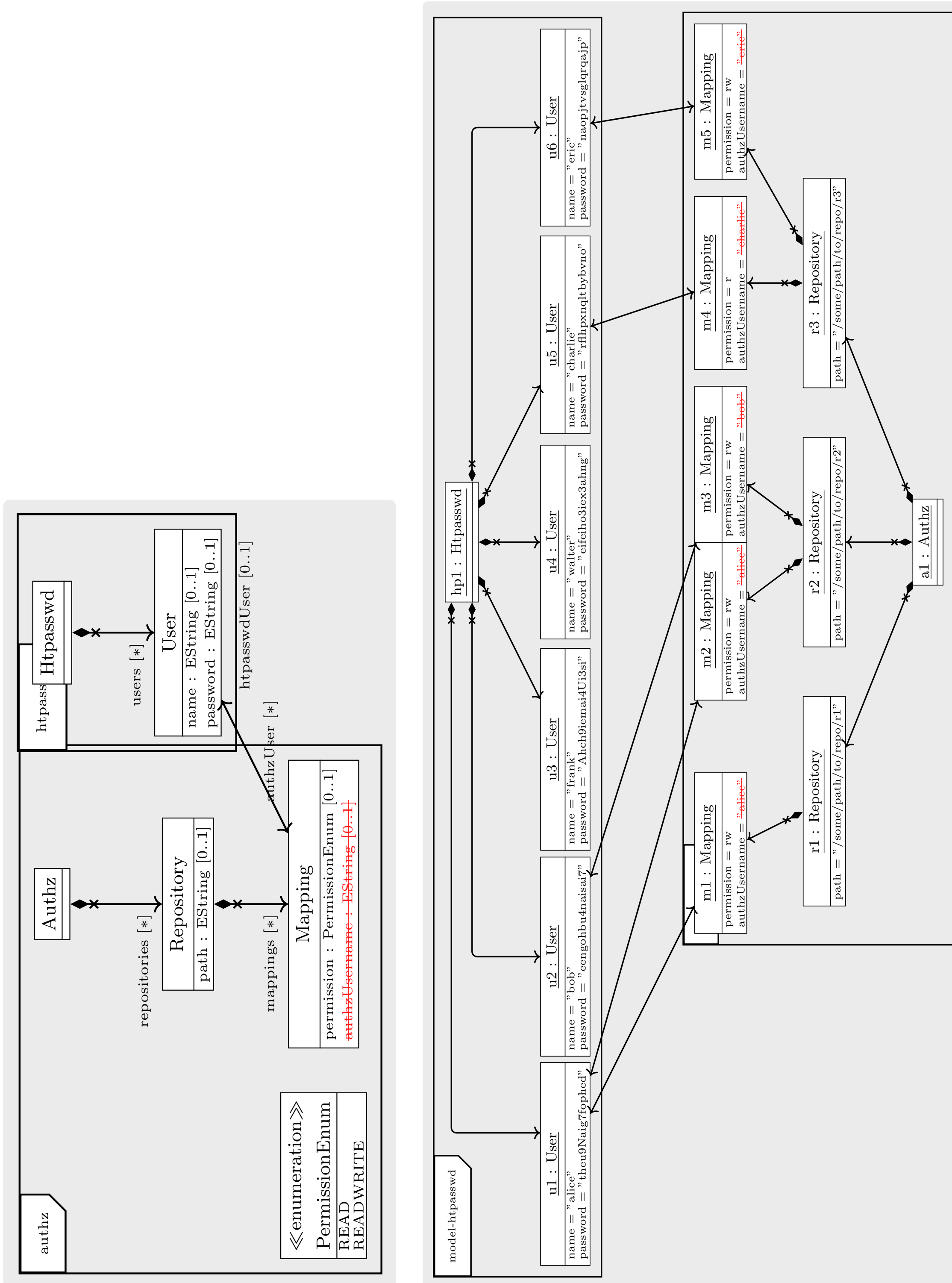


Figure 9.18: Changes in Metamodel (left/top) and Model (right/bottom) from ❹ to ❺

For the direction ❹→❺, the unidirectional operator is →DELETEATTRIBUTE (authz.Mapping.authzUsername).

Metamodel Decisions configured for →DELETEATTRIBUTE:

- fullyQualifiedAttributeName = authz.Mapping.authzUsername

Model Decisions configured for →DeleteAttribute: This operator has no configurations for model decisions.

For the inverse direction 04←05, the unidirectional operator is ←AddAttribute (authz.Mapping.authzUsername).

Htaccess ⟷ 06: DeleteAddAllIDAttributes

This operator removes the IDs which are used only internally to prevent their change by users. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.19.

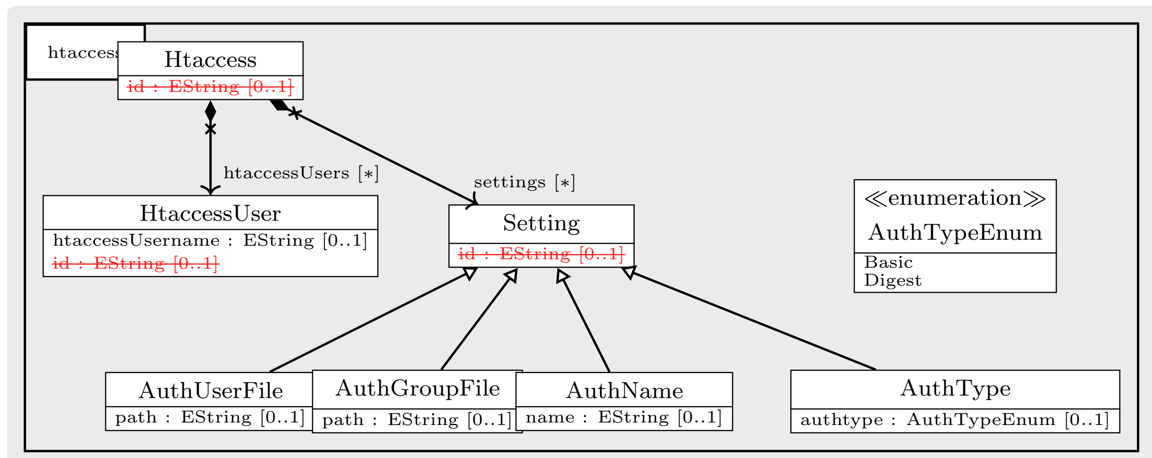


Figure 9.19: Metamodel Changes from Htaccess to 06

Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.20☞307.

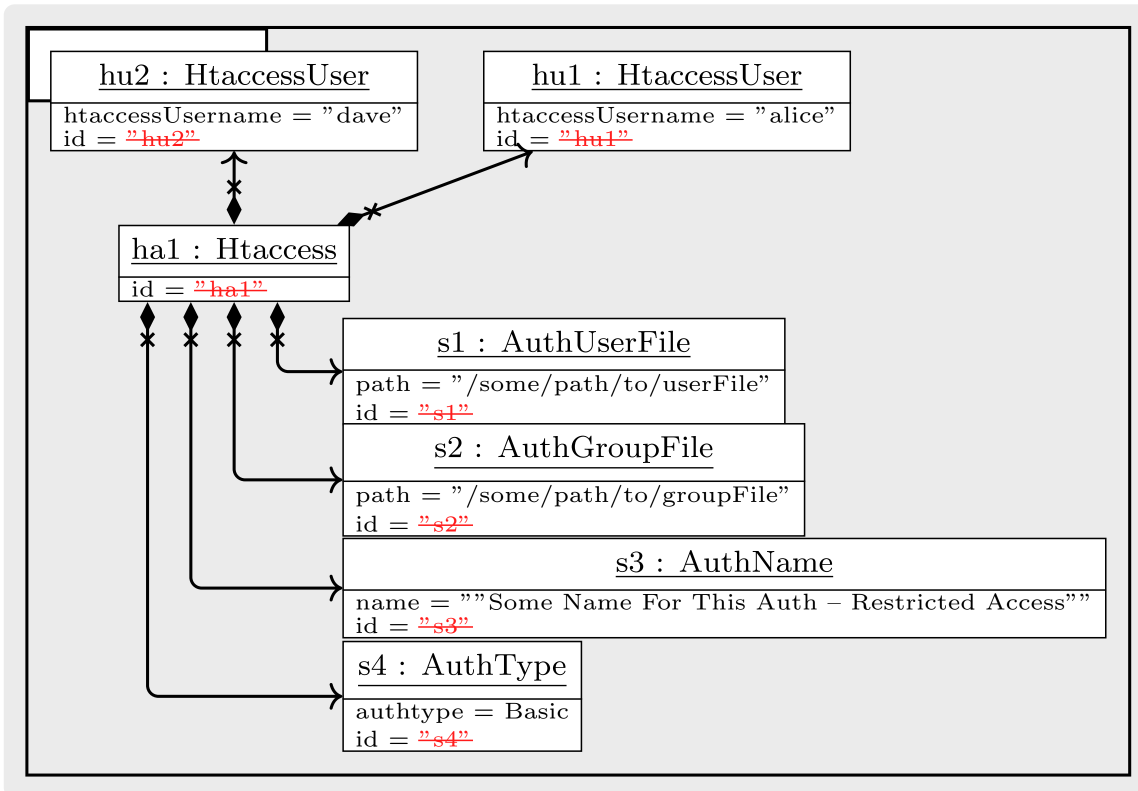


Figure 9.20: Model Changes from Htaccess to 06

For the direction Htaccess→06, the unidirectional operator is →DeleteAllIDAttributes (id).

Metamodel Decisions are not used by →CompositeOperator.

Model Decisions configured for →CompositeOperator: This operator has no configurations for model decisions.

For the inverse direction Htaccess←06, the unidirectional operator is ←AddAllIDAttributes (id).

9.2.2 Integrate Htaccess

Integrates Htaccess into the previous integration of Htpasswd and Authz.

07 ⟷ 08: AddDeleteOppositeRelation

This operator changes the existing unidirectional association into a bidirectional one. This is a preperation for the next operator and ensures, that a user can directly check, if it has an Htaccess right, by using this new direction for navigation, which is used in the model decision of the inverse operator. Since each HtaccessUser is always contained, 1 is chosen as new multiplicity. This association is created in unidirectional way by Xtext. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.21 ☞308. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.21 ☞308.

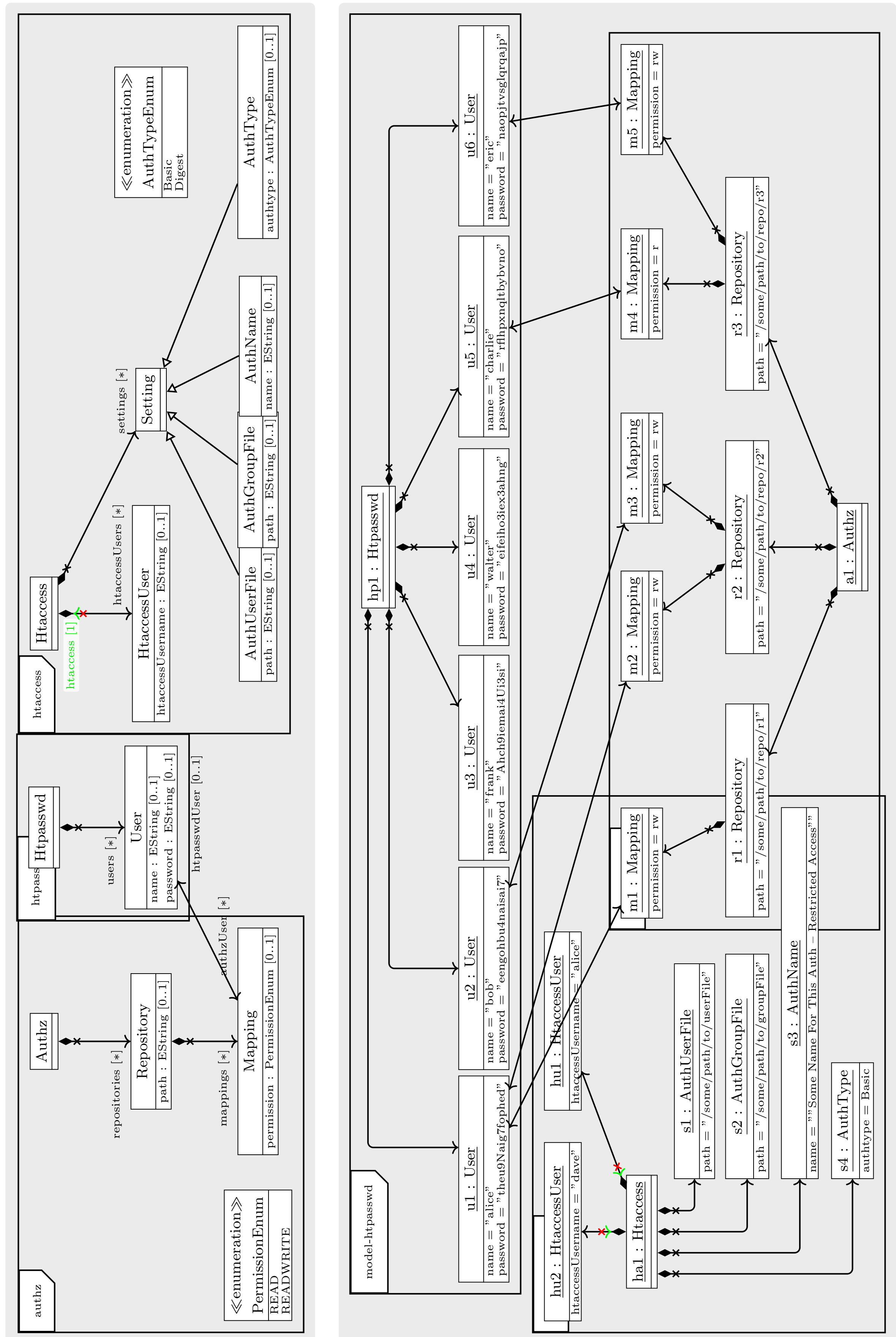


Figure 9.21: Changes in Metamodel (left/top) and Model (right/bottom) from 07 to 08

For the direction 07→08, the unidirectional operator is →AddOppositeRelation (add htaccess.HtaccessUser.htaccess as new opposite reference for htaccess.Htaccess.htaccess Users).

Metamodel Decisions configured for →AddOppositeRelation:

- existingReferenceFullyQualified = htaccess.Htaccess.htaccessUsers
- newFeatureName = htaccess
- newFeatureLowerBound = 1
- newFeatureUpperBound = 1

Model Decisions configured for →AddOppositeRelation: This operator has no configurations for model decisions.

For the inverse direction 07←08, the unidirectional operator is ←DeleteOppositeRelation (delete htaccess.HtaccessUser.htaccess, but keep its opposite htaccess.Htaccess.htaccessUsers).

08 ⟷ 09: MergeSplitClasses

This operator relates Htaccess rights and users explicitly to each other for C 2. This is done by merging the classes for users and rights into each other, as contrast to C 1. This is possible here, because each user has at maximum one Htaccess right. If there are more Htaccess rights possible, merging becomes impossible, but the Htaccess rights should be linked to ther user with AddDeleteAssociation again. In case of a missing matching user, the right is deleted, according to C 2 a. C 2 b does not required additional effort. Removed users are detected by remembering which rights and users are merged, which leads to a deletion of its access right, according to C 2 c. Renamings according to C 2 d and C 2 e are also realized by merge objects again, which were merged before. Because all users are contained in the Htpasswd-container, they must not be contained in the Htaccess-container (any more), the association is made non-containment. Since only some users have Htaccess-rights, the multiplicity is widened. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.22 ☞310. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.22 ☞310.

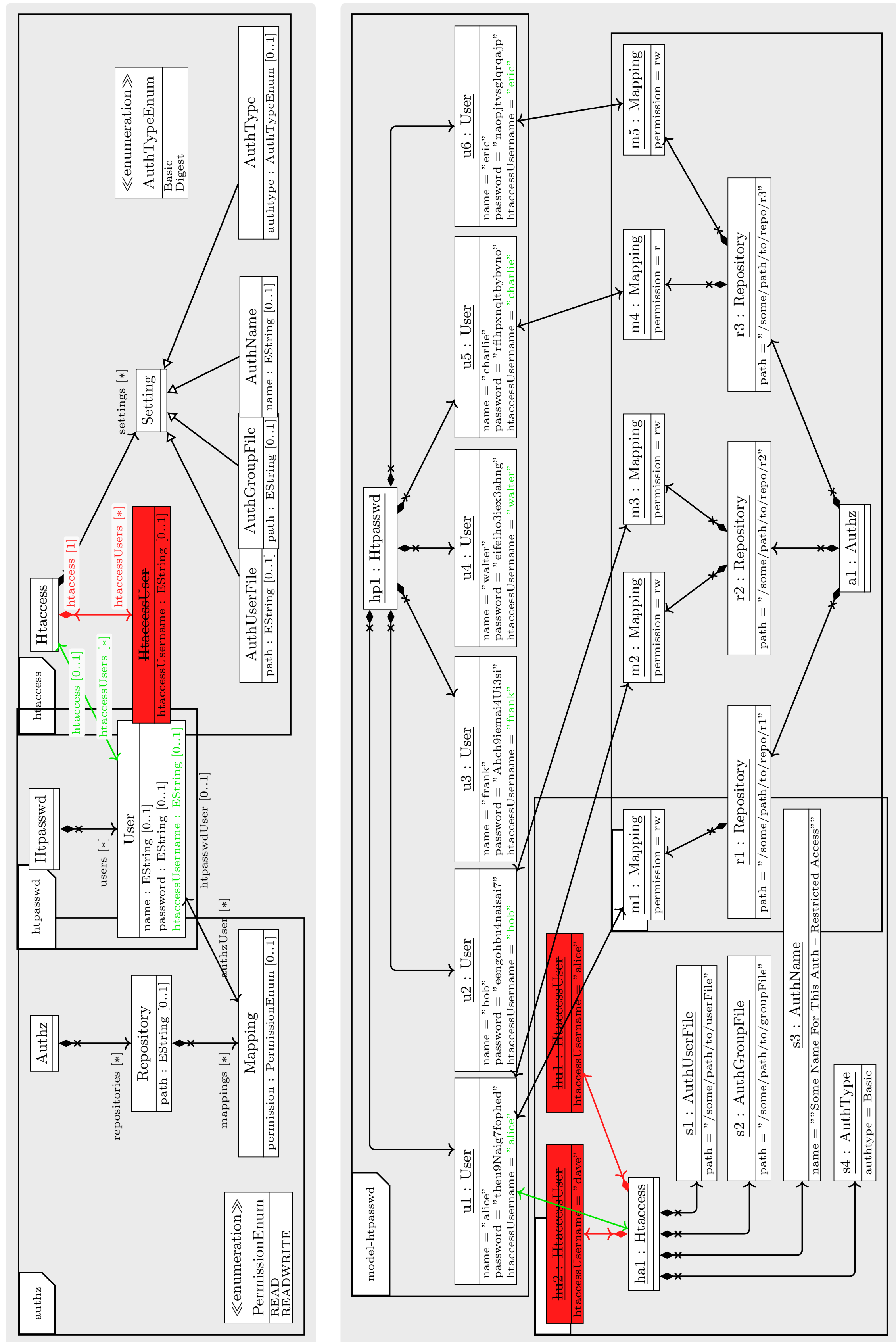


Figure 9.22: Changes in Metamodel (left/top) and Model (right/bottom) from 08 to 09

For the direction 08→09, the unidirectional operator is →MergeClasses (merge htaccess.HtaccessUser into htpasswd.User).

Metamodel Decisions configured for →MergeClasses:

- targetClassName = htpasswd.User
- sourceClassName = htaccess.HtaccessUser
- fullyQualifiedFeatureNamesToMakeNonContainment = { htaccess.Htaccess.htaccessUsers }
- fullyQualifiedFeatureNamesContainmentToChangeMultiplicity = { htaccess.HtaccessUser.htaccess }

Model Decisions configured for →MergeClasses: Configurations for model decisions are realized in `de ▸ unioldenburg ▸ se ▸ mmi ▸ rights ▸ integration ▸ decisions ▸ MergeHtaccessHtpasswdUsers`. All their configurations for model decisions are listed here:

- mergeAgain (arg0 : `Instance`, arg1 : `Instance`) : `MergeAgainDecision`
  Merges rights and users again, since they correspond to each other, which is known from the previous execution.
- mergedSourceHasNoTargetAnyMore (arg0 : `Instance`) : `MergedNowMissingDecision`
  Rights without user anymore are deleted, since their users were deleted before.
- mergedTargetHasNoSourceAnyMore (arg0 : `Instance`) : `MergedNowMissingDecision`
  Users without right are allowed and remain unchanged.
- areMatching (arg0 : `Instance`, arg1 : `Instance`) : `MatchDecision`
  Matches rights and users with same names.
- handleSourceWithoutMatch (arg0 : `Instance`) : `SearchedNoMatchDecision`
  Rights without user are deleted.
- initializeTargetFeatures (arg0 : `Instance`)
  Missing names are set, due to 1 as lower bound for the multiplicity of the corresponding attribute.
- initializeSourceFeatures (arg0 : `Instance`)
  Missing names are set, due to 1 as lower bound for the multiplicity of the corresponding attribute.

For the inverse direction 08←09, the unidirectional operator is ←SplitClass (split htpasswd.User, new class: htaccess.HtaccessUser).

09 ⟷ SUMM: MergeSplitAttributes

By removing the htaccessUsername, redundancy in terms of a duplicated attribute is reduced. The user name is still stored in name. This produces a cleaner SU(M)M and prevents inconsistencies in the SUM. Renamings are reflected in the merging process by using the changed name and ignoring the unchanged name, according to C 2 d and C 2 e. Therefore, the superfluous attribute is not deleted, but merged into the remaining attribute, which allows to take the newest value of both existing values. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.23☞312. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.23☞312.

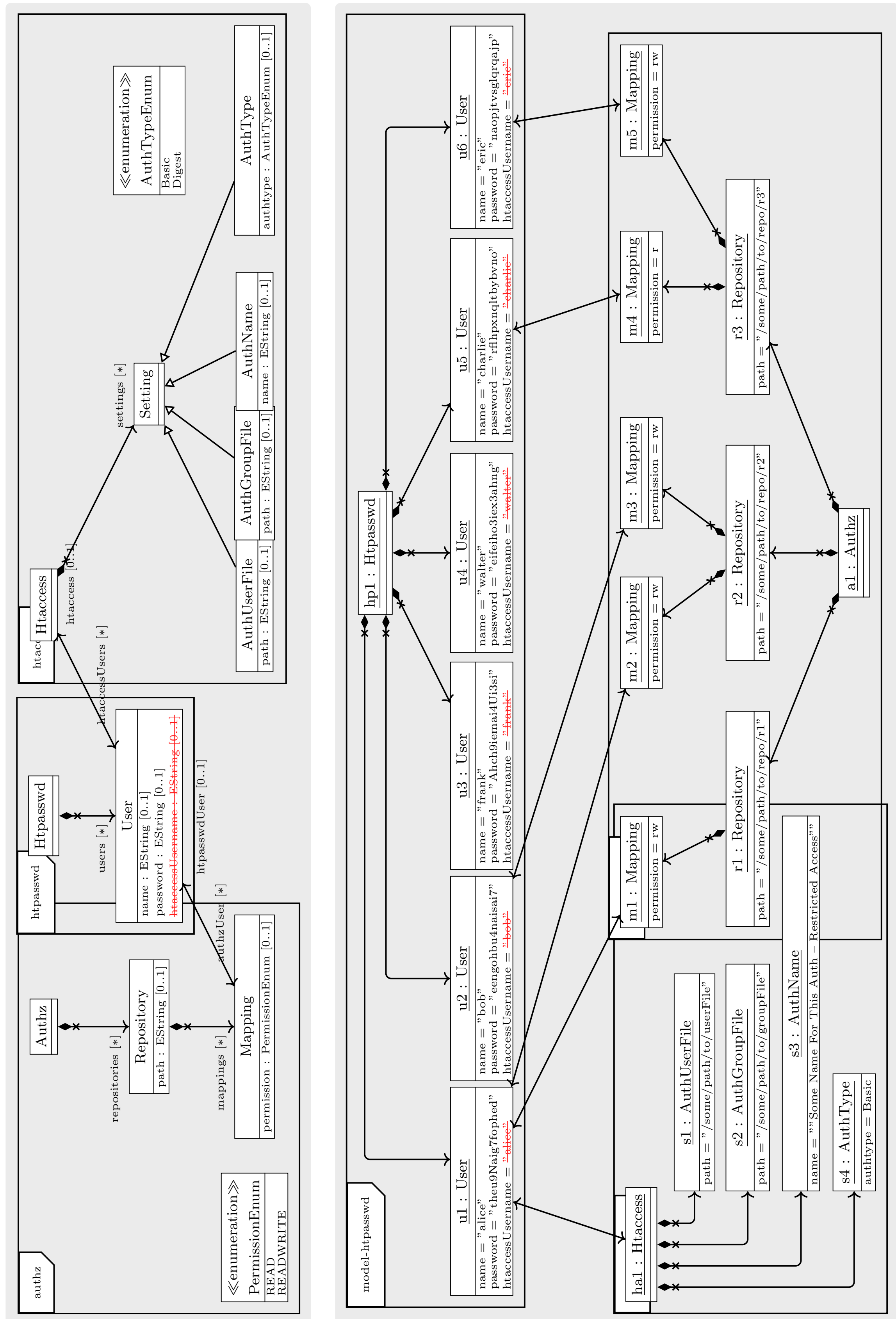


Figure 9.23: Changes in Metamodel (left/top) and Model (right/bottom) from 09 to SUMM

For the direction 09→SUMM, the unidirectional operator is →MergeAttributes (merge htpasswd.User.htaccessUsername into htpasswd.User.name).

Metamodel Decisions configured for →MergeAttributes:

- sourceAttributeNameFullyQualified = htpasswd.User.htaccessUsername
- targetAttributeNameFullyQualified = htpasswd.User.name

Model Decisions configured for →MergeAttributes: Configurations for model decisions are realized in `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸MergeAttributesWithoutConflict`. All their configurations for model decisions are listed here:

- merge (instance : `Instance`, sourceSlot : `AttributeSlot`, targetSlot : `AttributeSlot`)

  Merges two attributes with the assumption, that missing values can be copied and duplicate value are the same. If values are changed, the changed values are used as merged result. Precondition: The source slot and the target slot are valid, i.e. contain useful values. That is required to enable checking for changed values and to distinguish a missing value from an empty slot. (In case of conflicts (which should not occur in theory), the values of the target slot win!) This is a default configuration, reused from `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸MergeAttributesWithoutConflict`.

For the inverse direction 09←SUMM, the unidirectional operator is ←SplitAttribute (split htpasswd.User.name, new: htpasswd.User.htaccessUsername).

9.3 Definition of a new View(point)

This section documents, how the new view(point) is derived from the SU(M)M. For each used operator, its impact is highlighted and its configuration is sketched.

SUMM ⟷ 10: SubSet

Since the configuration stuff Htaccess is not needed in the new view, it is removed. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.24 ☞314. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.24 ☞314.

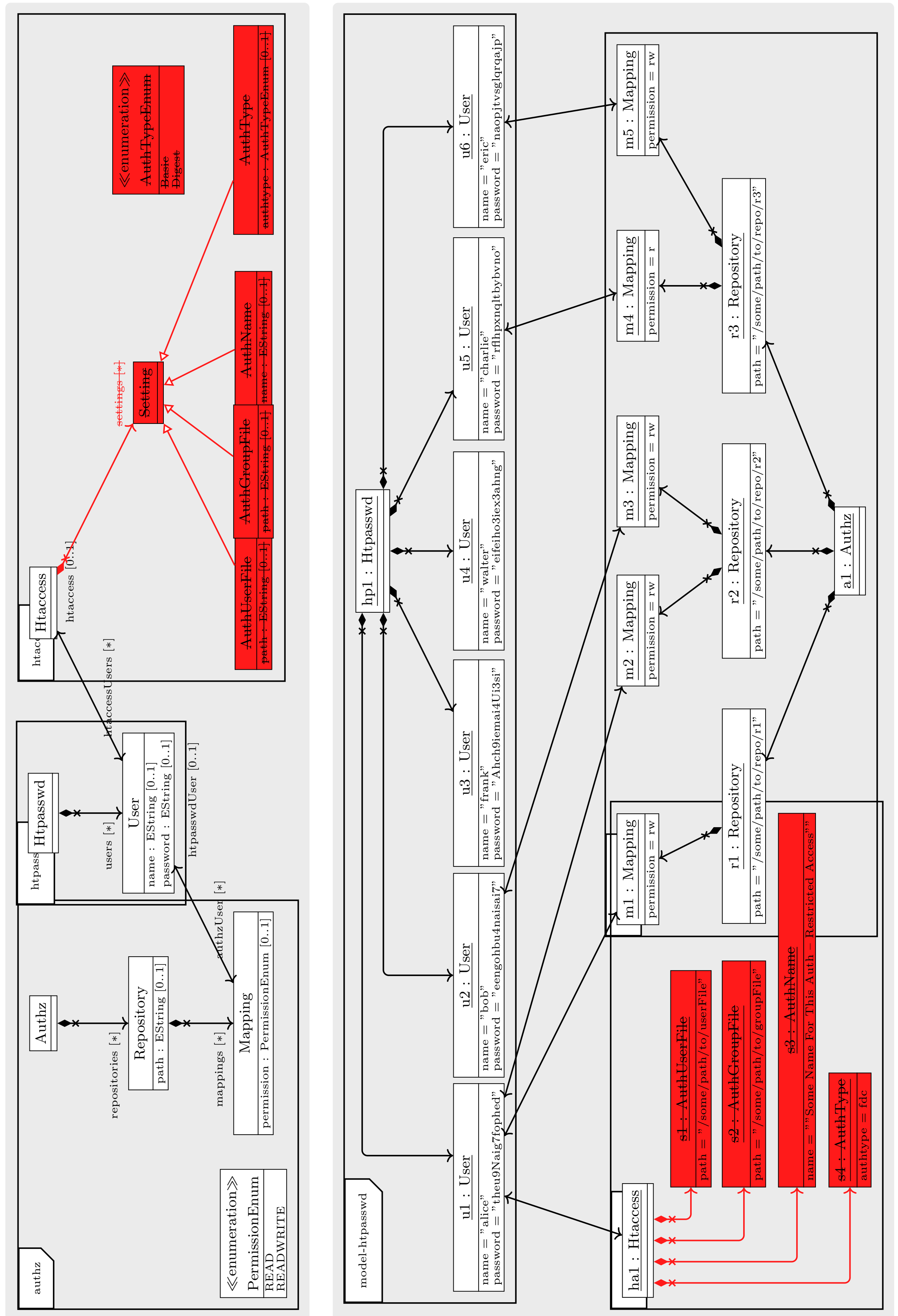


Figure 9.24: Changes in Metamodel (left/top) and Model (right/bottom) from SUMM to 10

For the direction SUMM→⓾, the unidirectional operator is →SubSetFilter (htaccess.Setting, htaccess.AuthType, htaccess.AuthName, htaccess.AuthGroupFile, htaccess.AuthUserFile, htaccess.AuthTypeEnum).

Metamodel Decisions are not used by →SubSetFilter.

Model Decisions configured for →SubSetFilter: This operator has no configurations for model decisions.

For the inverse direction SUMM←⓾, the unidirectional operator is ←SubSetRecreate (htaccess.Setting, htaccess.AuthType, htaccess.AuthName, htaccess.AuthGroupFile, htaccess.AuthUserFile, htaccess.AuthTypeEnum).

⓾ ⟷ ⓫: AddDeleteAttribute

To show, how many read-only Authz permission a user has, all users get such a new attribute. This new information in the view is read-only, i.e. changes of these cumulative values are reverted. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.25☞316. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.25☞316.

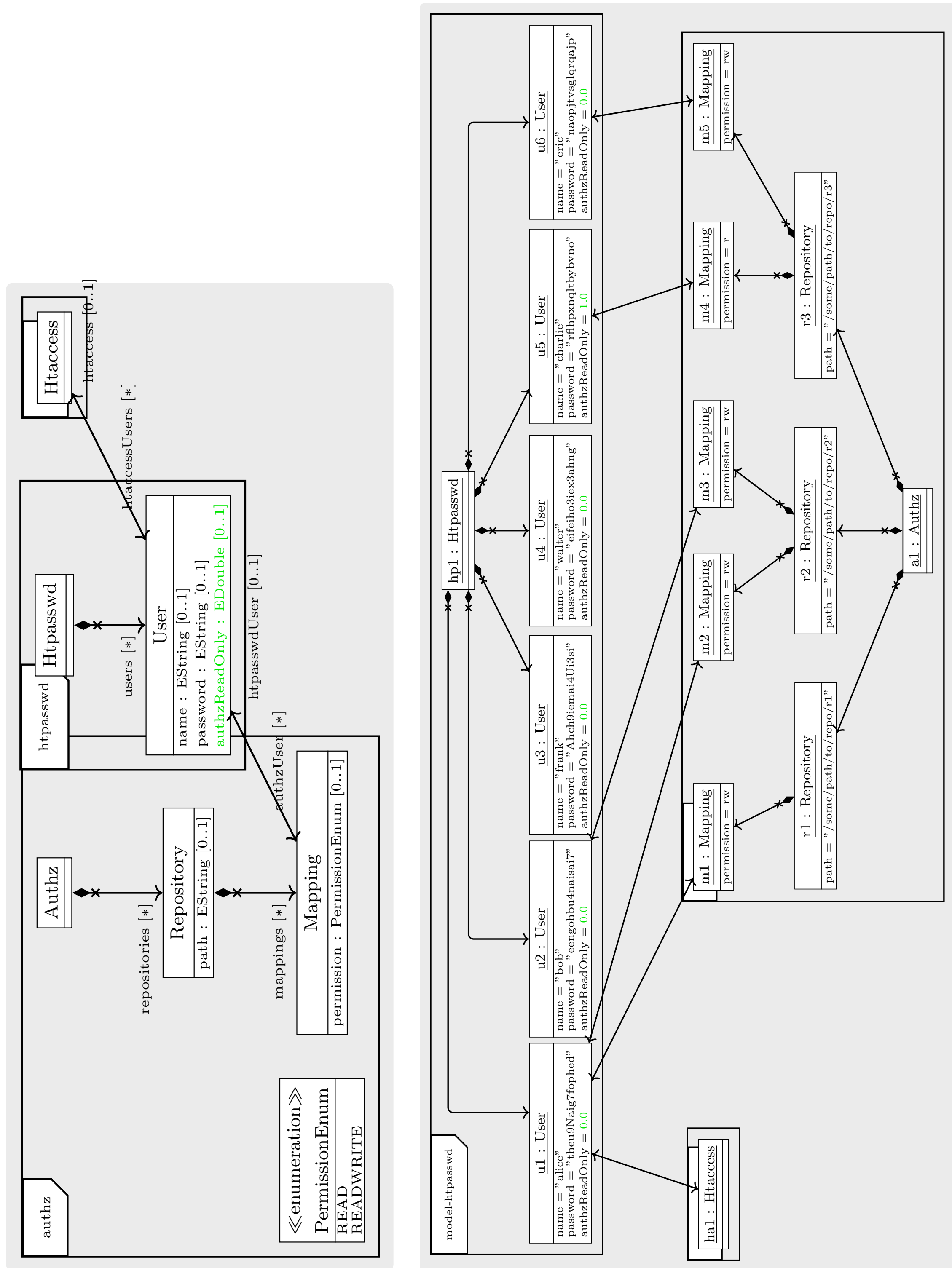


Figure 9.25: Changes in Metamodel (left/top) and Model (right/bottom) from ⓾ to ⓫

For the direction ⓾→⓫, the unidirectional operator is →AddAttribute (htpasswd. User.authzReadOnly).

Metamodel Decisions configured for →AddAttribute:

- classWithNewAttributeFullyQualified = htpasswd.User
- attributeName = authzReadOnly

- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EDouble

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de▸unioldenburg▸se▸mmi▸rights▸integration▸AuthIntegrationExample▸2`. All their configurations for model decisions are listed here:

- computeInitialValue (arg0 : `Instance`, arg1 : `EAttribute`) : `Object`
  The configuration counts, how many Authz mappings have a read-only permission.

For the inverse direction ⑩←⑪, the unidirectional operator is ←DeleteAttribute (htpasswd.User.authzReadOnly).

⑪ ⟷ ⑫: AddDeleteAttribute

To show, how many read-and-write Authz permission a user has, all users get such a new attribute. This new information in the view is read-only, i.e. changes of these cumulative values are reverted. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.26☞318. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.26☞318.

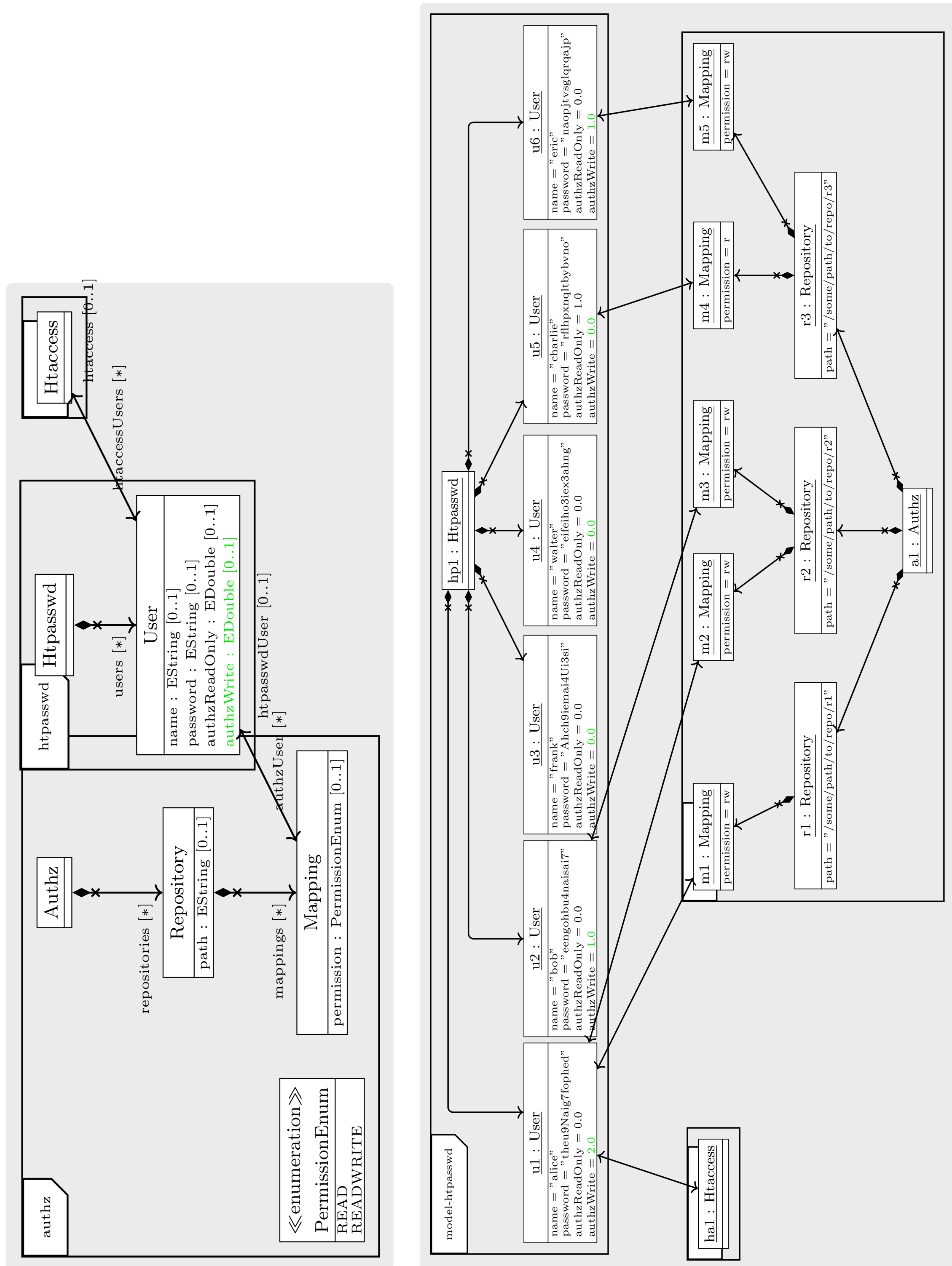


Figure 9.26: Changes in Metamodel (left/top) and Model (right/bottom) from ⓫ to ⓬

For the direction ⓫→⓬, the unidirectional operator is →AddAttribute (htpasswd.User.authzWrite).

Metamodel Decisions configured for →AddAttribute:

- classWithNewAttributeFullyQualified = htpasswd.User
- attributeName = authzWrite

- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EDouble

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de ▸ unioldenburg ▸ se ▸ mmi ▸ rights ▸ integration ▸ AuthIntegrationExample ▸ 3`. All their configurations for model decisions are listed here:

- computeInitialValue (arg0 : `Instance`, arg1 : `EAttribute`) : `Object`
  The configuration counts, how many Authz mappings have a read-and-write permission.

For the inverse direction ⓫←⓬, the unidirectional operator is ←DeleteAttribute (htpasswd.User.authzWrite).

⓬ ⟷ ⓭: ReplaceReferenceByAttribute

To inform, whether the user has the Htaccess right or not, a new boolean attribute is introduced. Its value can be changed in the view and the Htaccess right is added or removed in the SUM accordingly. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.27☞320. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.27☞320.

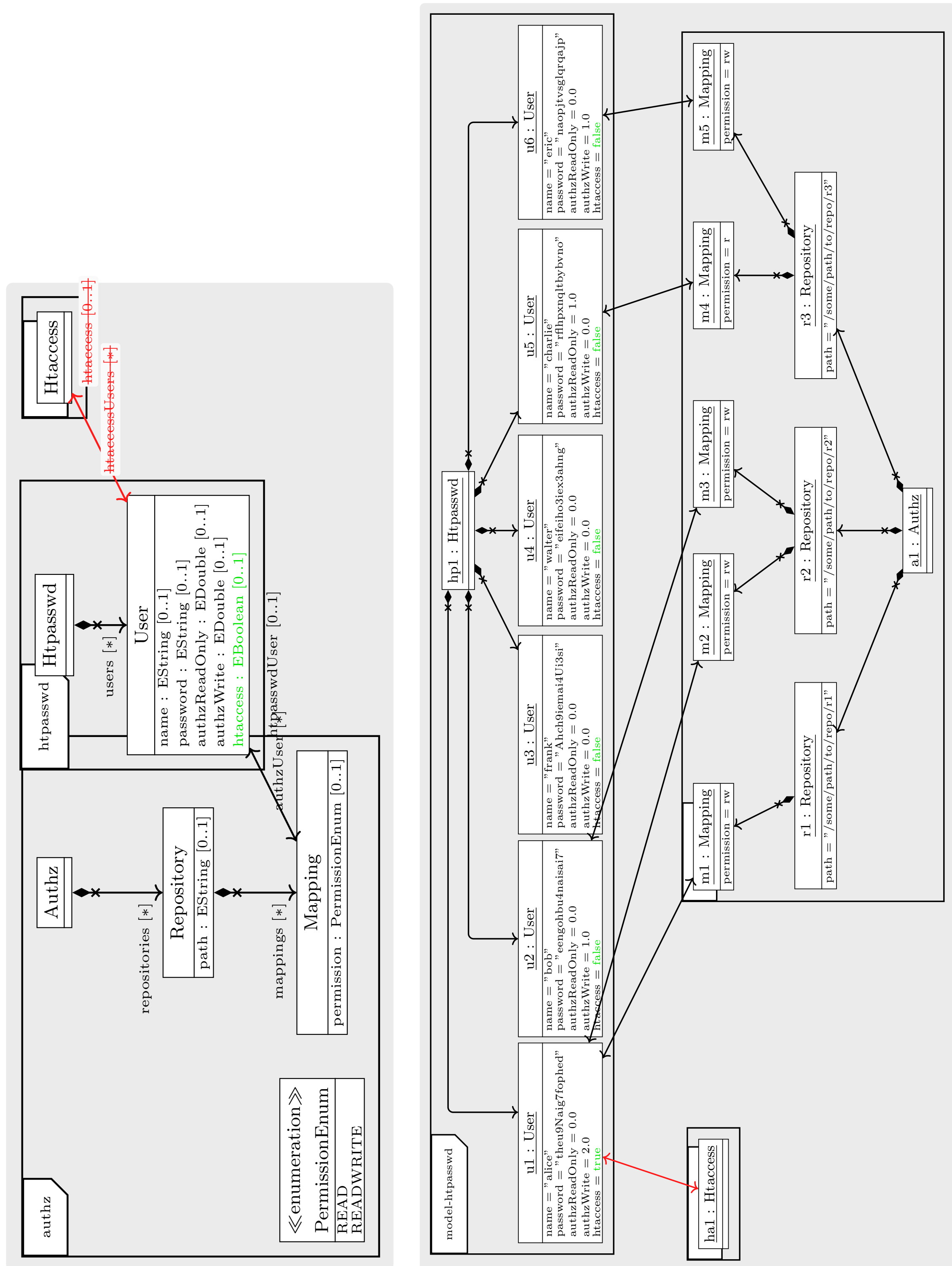


Figure 9.27: Changes in Metamodel (left/top) and Model (right/bottom) from ⓬ to ⓭

For the direction ⓬→⓭, the unidirectional operator is →ReplaceReferenceByAttribute (htpasswd.User.htaccess : htaccess.Htaccess).

Metamodel Decisions configured for →ReplaceReferenceByAttribute:

- sourceClassName = htpasswd.User
- referenceName = htaccess

- newAttributeType = ecore.EBoolean

Model Decisions configured for →ReplaceReferenceByAttribute: Configurations for model decisions are realized in `de▸unioldenburg▸se▸mmi▸rights▸integration▸AuthIntegrationExample▸4`. All their configurations for model decisions are listed here:

- replaceLink (arg0 : `ReferenceSlot`, arg1 : `Instance`, arg2 : `ReplaceReferenceByAttribute`) : `Object`

  If this method is called, the 'oldValue' exists and indicates, that the user has the Htaccess right, and returns TRUE. Otherwise, the default value FALSE is used implicitly.

For the inverse direction ⓬←⓭, the unidirectional operator is ←ReplaceAttributeByReference (htpasswd.User.htaccess ⟶ htaccess.Htaccess).

⓭ ⟷ ⓰

Since the details about the Htaccess and Authz rights are not necessary anymore, they are removed now. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.28☞322. Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.28☞322.

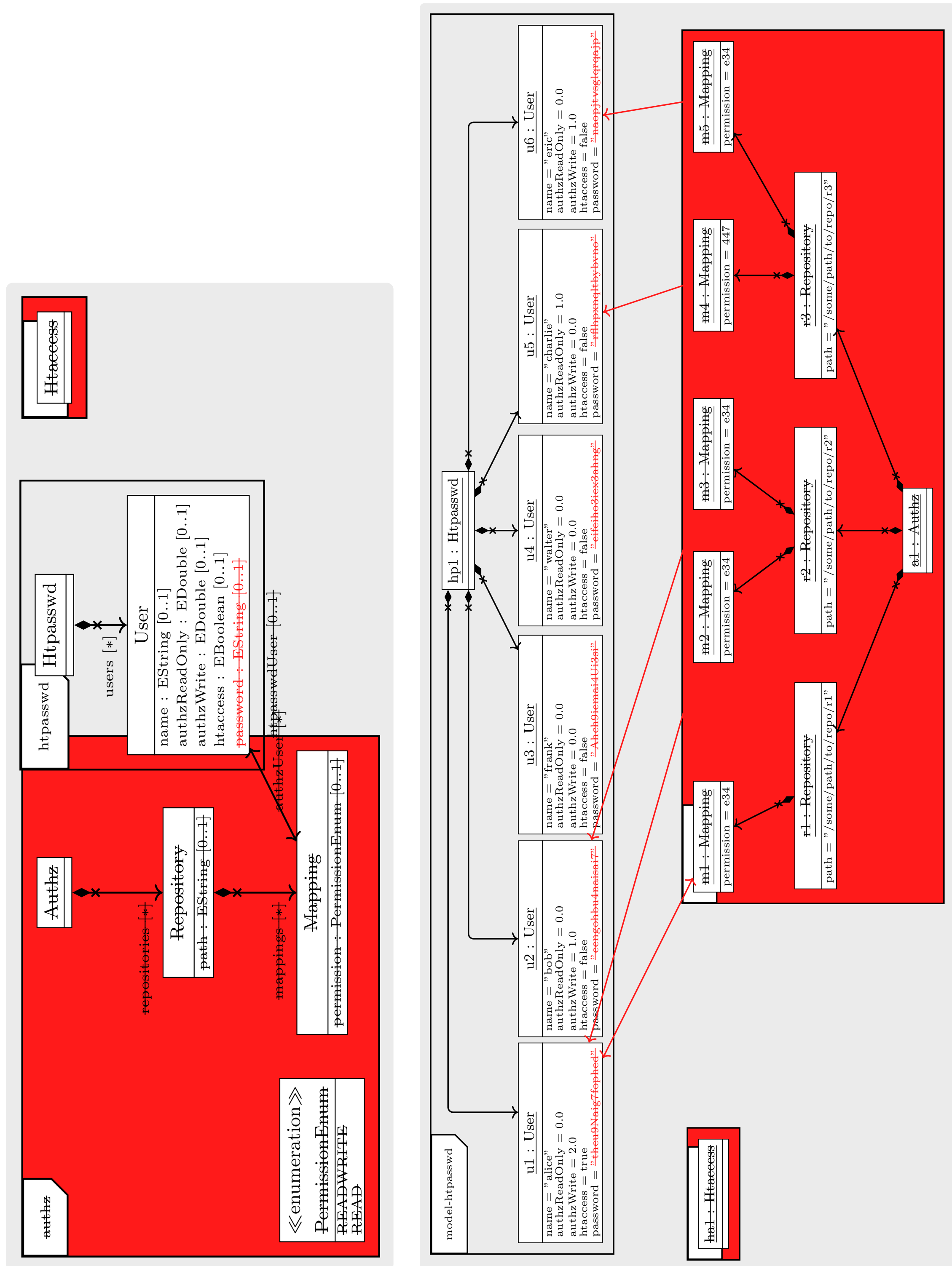


Figure 9.28: Metamodel Changes from ⓭ to ⓰

This is realized by the following three operators:

1. For the direction ⓭→⓮, the unidirectional operator is →SubSetFilter (authz, htaccess, htpasswd.User.password).

   Deleting the two packages with all their classes in the metamodel, leads also to the deletion of all instances.

Metamodel Decisions are not used by →SubSetFilter.

Model Decisions configured for →SubSetFilter: This operator has no configurations for model decisions.

For the inverse direction ⓭←⓮, the unidirectional operator is ←SubSetRecreate (authz, htaccess, htpasswd.User.password).

2. For the direction ⓮→⓯, the unidirectional operator is →DeleteNamespace (model-authz).

   The previous operator deleted all instances inside this model namespace, so it can be deleted now. It is not deleted together with the previous operator, since it allows only to specify elements of the metamodel explicitly, but not namespaces of the model.

Metamodel Decisions configured for →DeleteNamespace:

- namespaceFullName = model-authz

Model Decisions configured for →DeleteNamespace: This operator has no configurations for model decisions.

For the inverse direction ⓮←⓯, the unidirectional operator is ←AddNamespace (model-authz).

3. For the direction ⓯→⓰, the unidirectional operator is →DeleteNamespace (model-htaccess).

   Now the second empty model namespace is deleted.

Metamodel Decisions configured for →DeleteNamespace:

- namespaceFullName = model-htaccess

Model Decisions configured for →DeleteNamespace: This operator has no configurations for model decisions.

For the inverse direction ⓯←⓰, the unidirectional operator is ←AddNamespace (model-htaccess).

⓰ ⟷ ⓱: RenameClassifier

To make clear, that this is a new view(point) and not Htpasswd anymore, some elements are renamed now. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.29 ☞324.

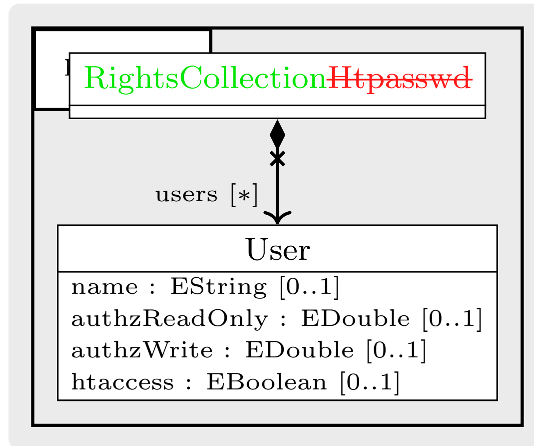


Figure 9.29: Metamodel Changes from ⓰ to ⓱

Accordingly, this part of the orchestration does not change the model. For the direction ⓰→⓱, the unidirectional operator is →RenameClassifier (htpasswd.Htpasswd → RightsCollection).

Metamodel Decisions configured for →RenameClassifier:

- elementFullyQualified = htpasswd.Htpasswd
- name = RightsCollection

Model Decisions configured for →RenameClassifier: This operator has no configurations for model decisions.

For the inverse direction ⓰←⓱, the unidirectional operator is ←RenameClassifier (htpasswd.RightsCollection → Htpasswd).

⓱ ⟷ ⓲: AddDeleteAttribute

Since the Excel format needs a row number for each entry, each user will get an Integer number, sorted by the user name. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.30.

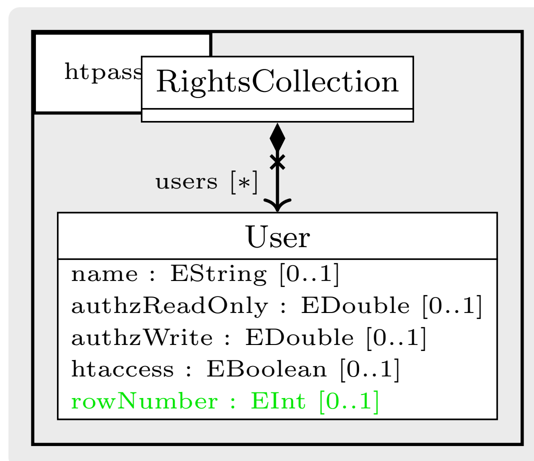


Figure 9.30: Metamodel Changes from ⓱ to ⓲

Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.31.

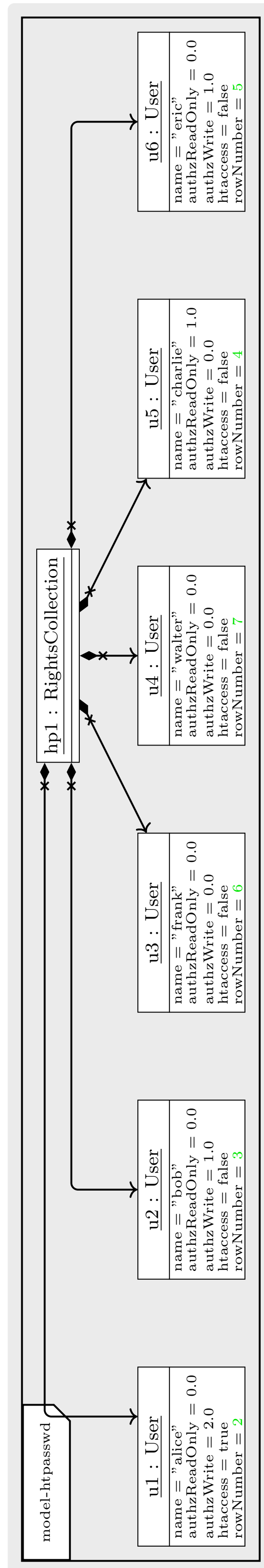


Figure 9.31: Model Changes from 17 to 18

For the direction ⓱→⓲, the unidirectional operator is →AddAttribute (htpasswd.User.rowNumber).

Metamodel Decisions configured for →AddAttribute:

- classWithNewAttributeFullyQualified = htpasswd.User
- attributeName = rowNumber
- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EInt

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de`▸`unioldenburg`▸`se`▸`mmi`▸`rights`▸`integration`▸`AuthIntegrationExample`▸`6`. All their configurations for model decisions are listed here:

- computeInitialValue (arg0 : `Instance`, arg1 : `EAttribute`) : `Object`
  To determine the order/position of the current user, this configuration counts, how many other users have a name which is earlier in the alphabet, than the name of the current user.

For the inverse direction ⓱←⓲, the unidirectional operator is ←DeleteAttribute (htpasswd.User.rowNumber).

⓲ ⟷ Overview: AddDeleteAttribute

Since the current Excel adapter requires to have a column usable as UUID, such column is created here. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 9.32.

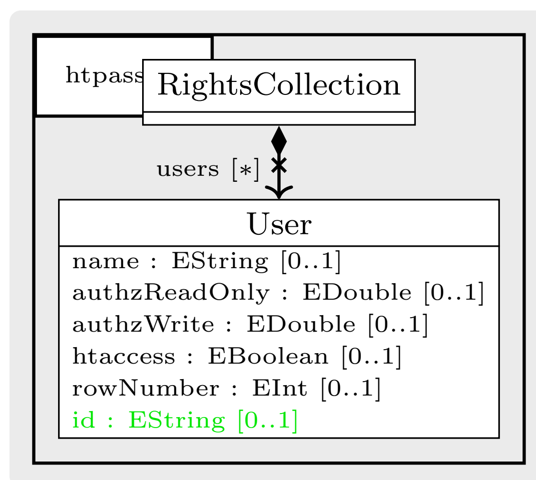


Figure 9.32: Metamodel Changes from ⓲ to Overview

Accordingly, this part of the orchestration changes the model, as depicted in Figure 9.33☞327.

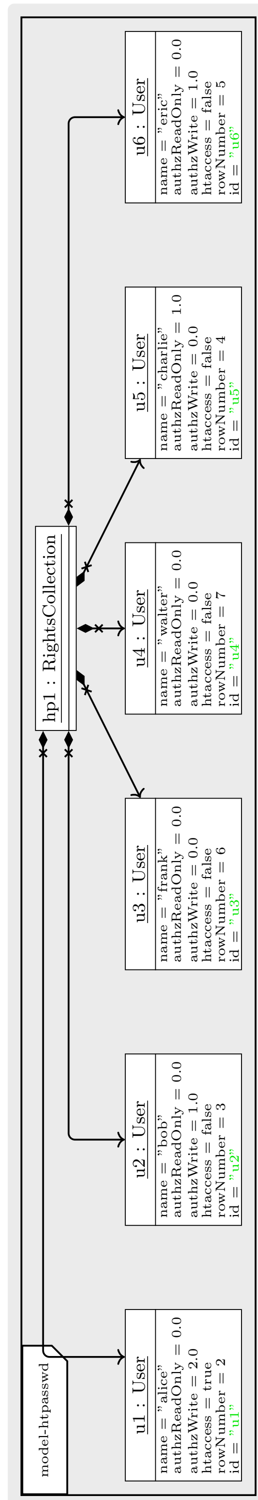


Figure 9.33: Model Changes from ⓲ to Overview

For the direction ⓲→Overview, the unidirectional operator is →ADDATTRIBUTE (htpasswd.User.id).

Metamodel Decisions configured for →ADDATTRIBUTE:

- classWithNewAttributeFullyQualified = htpasswd.User
- attributeName = id

- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EString

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣AddAttributeIdUuid`. All their configurations for model decisions are listed here:

- computeInitialValue (instanceToFix : `Instance`, newAttribute : `EAttribute`) : `Object`
  Uses the UUID of the instance as initial value for the new attribute. This is a default configuration, reused from `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣AddAttributeIdUuid`.

For the inverse direction 18←Overview, the unidirectional operator is ←DeleteAttribute (htpasswd.User.id).

9.4 Validation Scenarios

This section documents acceptance test cases for this application domain. Each test case is documented in its own section. The first section shows the initialization of the SU(M)M before running the test case. Additionally, the initialization shows the models for all views, before they might be changed in the particular test case. The initialization is the same for all following test cases and is documented only once.

9.4.1 Initialization by Execution

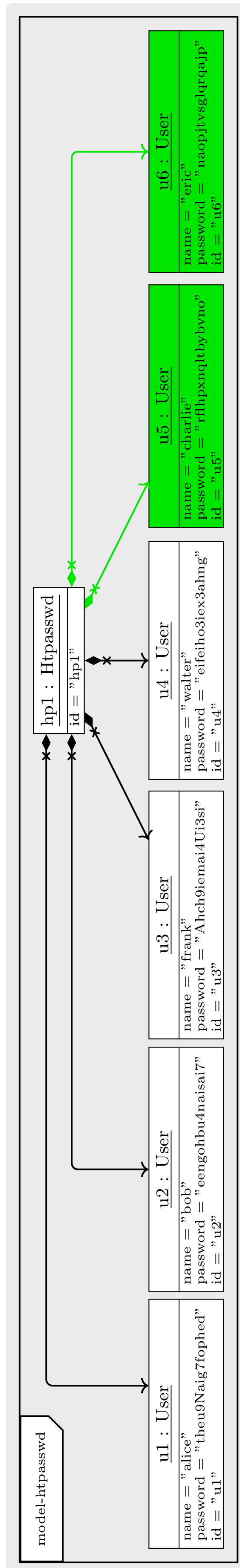
This is the description of the first execution run for the initialization: Starting with the initial data sources, the SU(M) and the new view(point)s are created, while possible inconsistencies in the data sources are fixed. The resulting models serve as starting point for the following test case scenarios.

As result after completing the synchronization, the following changes are expected:

- In Htpasswd, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #hp1
2  alice : theu9Naig7fophed #u1
3 bob : eengohbu4naisai7 #u2

5 frank : Ahch9iemai4Ui3si #u3
6 walter : eifeiho3iex3ahng #u4
7 charlie : rflhpxnqltbybvno #u5
8 eric : naopjtvsglqrqajp #u6
```

The model with highlighted changes is represented graphically:

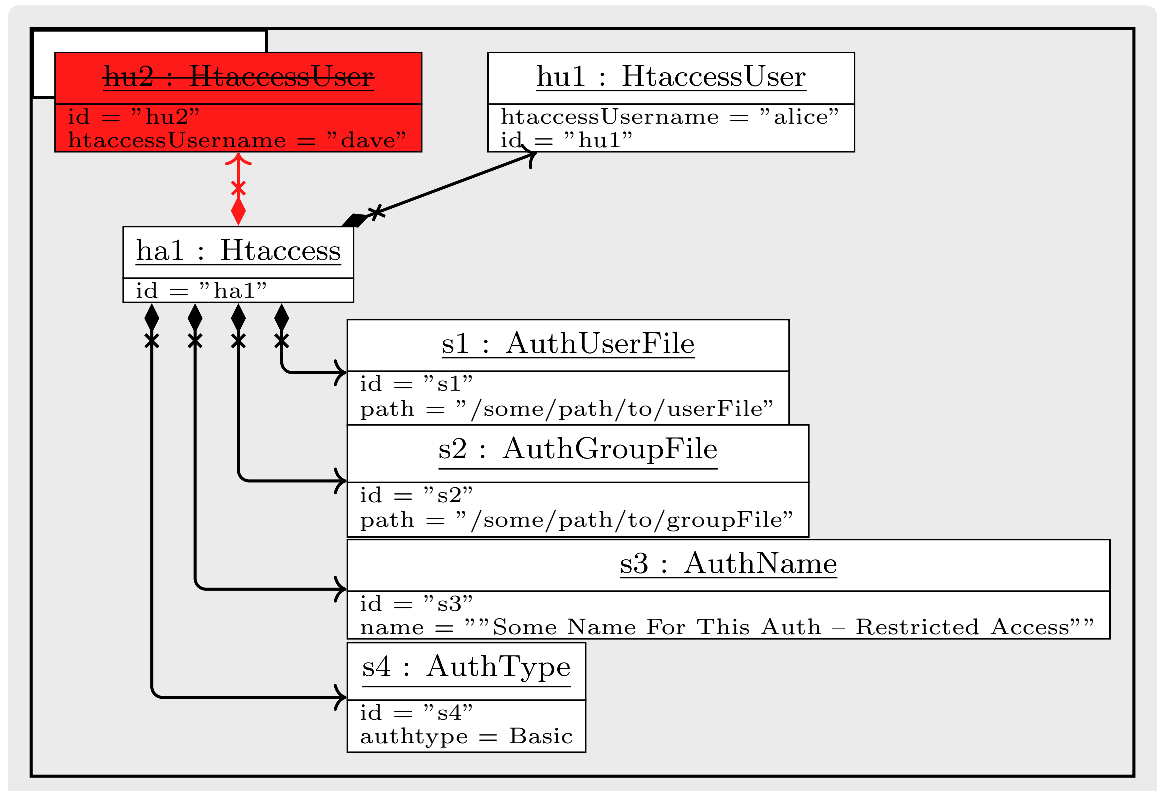

model-htpasswd
hp1 : Htpasswd
id = "hp1"
u1 : User
name = "alice"
password = "theu9Naig7fophed"
id = "u1"
u2 : User
name = "bob"
password = "eengohbu4naisai7"
id = "u2"
u3 : User
name = "frank"
password = "Ahch9iemai4Ui3si"
id = "u3"
u4 : User
name = "walter"
password = "eifeiho3iex3ahng"
id = "u4"
u5 : User
name = "charlie"
password = "rflhpxnqltbybvno"
id = "u5"
u6 : User
name = "eric"
password = "naopjtvsglqrqajp"
id = "u6"

- In Authz, no changes are expected in the model.

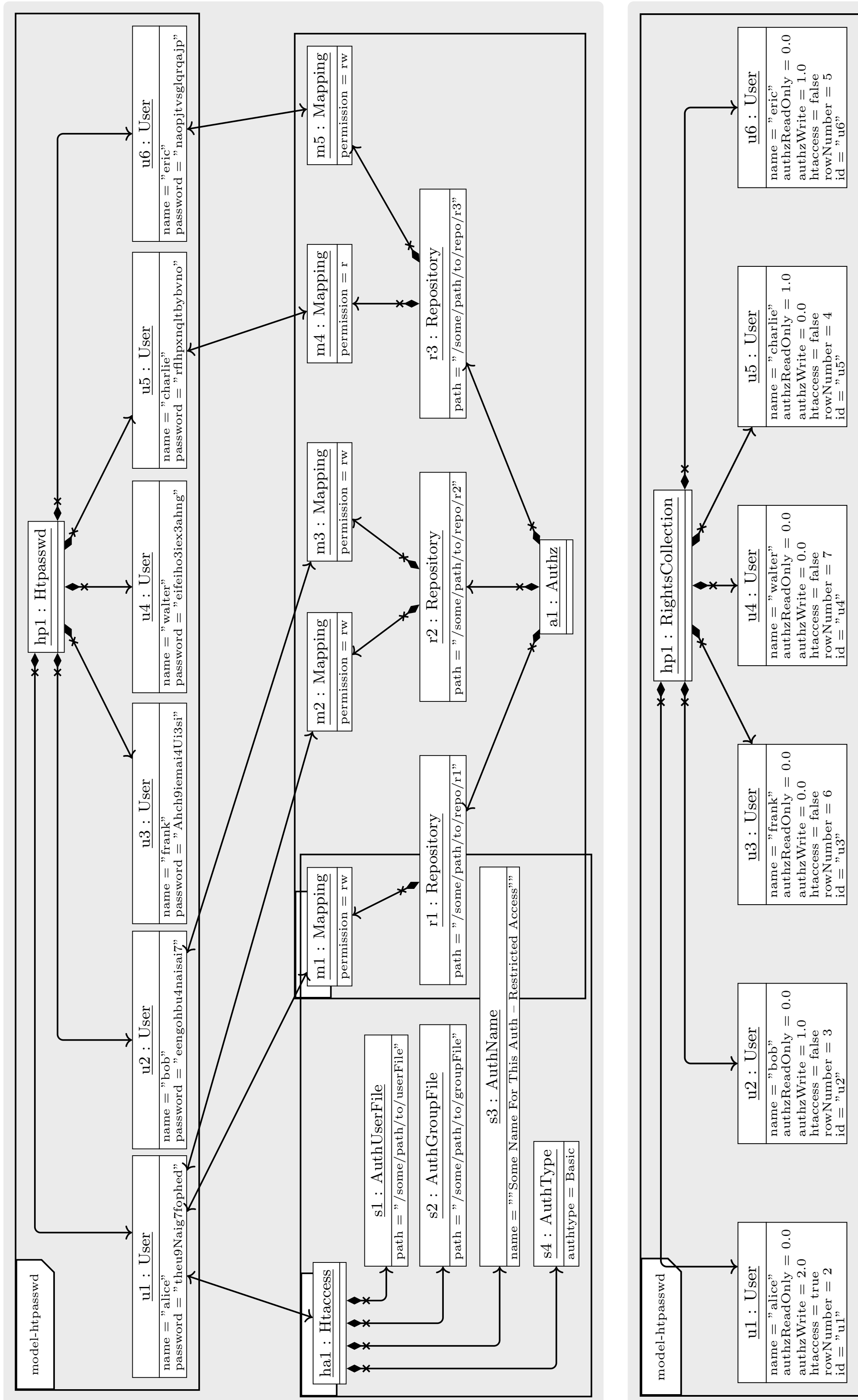
- In Htaccess, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #ha1
2  AuthUserFile /some/path/to/userFile #s1
3 AuthGroupFile /some/path/to/groupFile #s2
4 AuthName "Some Name For This Auth -- Restricted Access" #s3
5 AuthType Basic #s4

7 require user alice #hu1
8 require user dave #hu2
```

The model with highlighted changes is represented graphically:



- Since no model existed for SUM before this execution, it is conceptually not possible to specify expected changes. Only the model after the execution can be defined (left):



- Since no model existed for Overview before this execution, it is conceptually not possible to specify expected changes. Only the model after the execution can be defined in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected

changes in the models. Therefore, the scenario is successfully fulfilled by this test case.

9.4.2 Scenario: renamed Mapping, reload externally (Authz)

Changing the user name of an existing Authz access right of a user with multiple rights leads to the creation of a new user with the changed user name, since such user is missing. This test case tests mainly C 1 d.

The *User* applies the desired changes to Authz by changing its external representation. The model with highlighted changes is represented with its concrete rendering:

```
1  #a1
2  [/some/path/to/repo/r1] #r1
3  alice aliceChangedExternally = rw #m1
4  [/some/path/to/repo/r2] #r2
5  alice = rw #m2
6  bob = rw #m3

8  [/some/path/to/repo/r3] #r3
9  charlie = r #m4
10 eric = rw #m5
```

The model differences are represented in textual form:

```
1  m1.remove(authzUsername, 0, "alice")
2  m1.add(authzUsername, 0, "aliceChangedExternally")
```

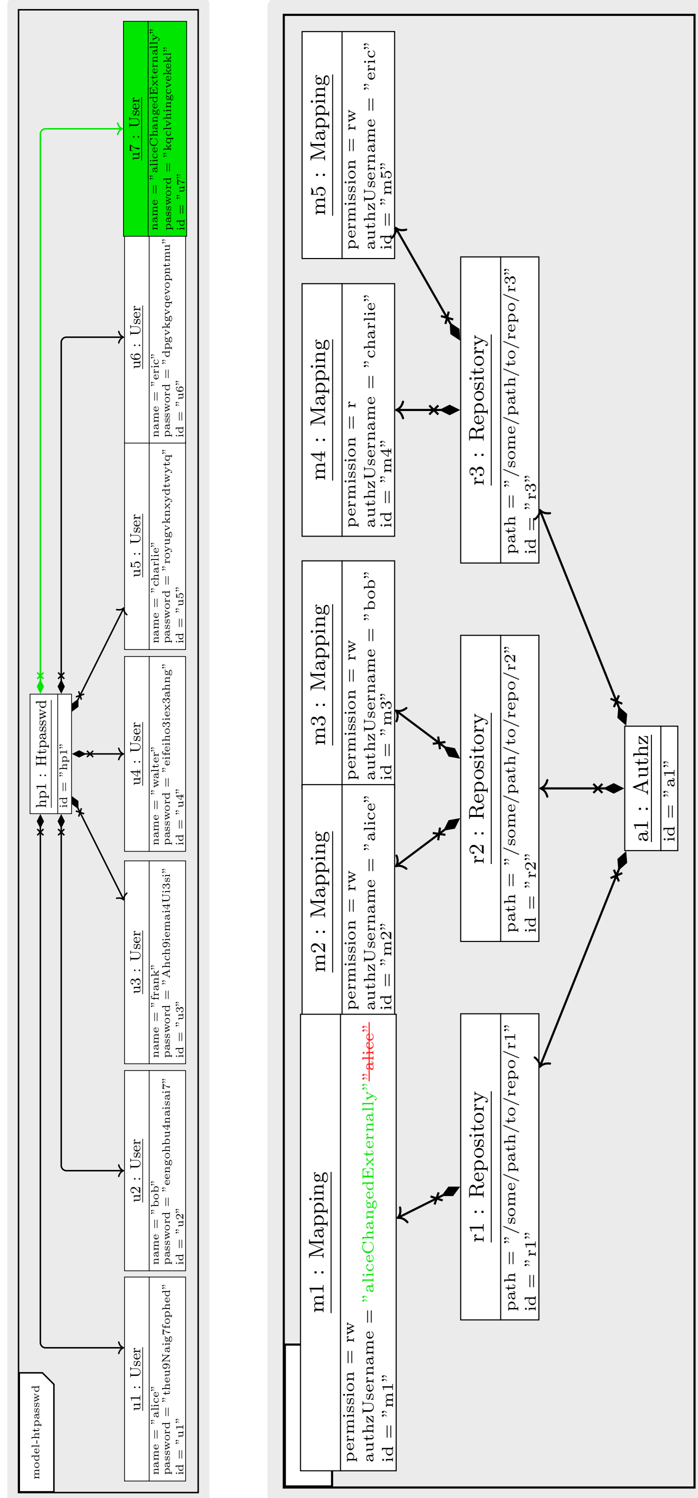
User $\Delta^{14}_{\text{Authz}}$

As result after completing the synchronization, the following changes are expected:

- In Htpasswd, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1  #hp1
2   alice : theu9Naig7fophed #u1
3  bob : eengohbu4naisai7 #u2
4  frank : Ahch9iemai4Ui3si #u3
5  walter : eifeiho3iex3ahng #u4
6  charlie : royugvknxydtwytq #u5
7  eric : dpgvkgvqevopntmu #u6
8  aliceChangedExternally : kqclvhingcvekekl #u7
```

The model with highlighted changes is represented graphically (left):



- In Authz, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Authz}}$). The model with highlighted changes is represented with its concrete rendering:

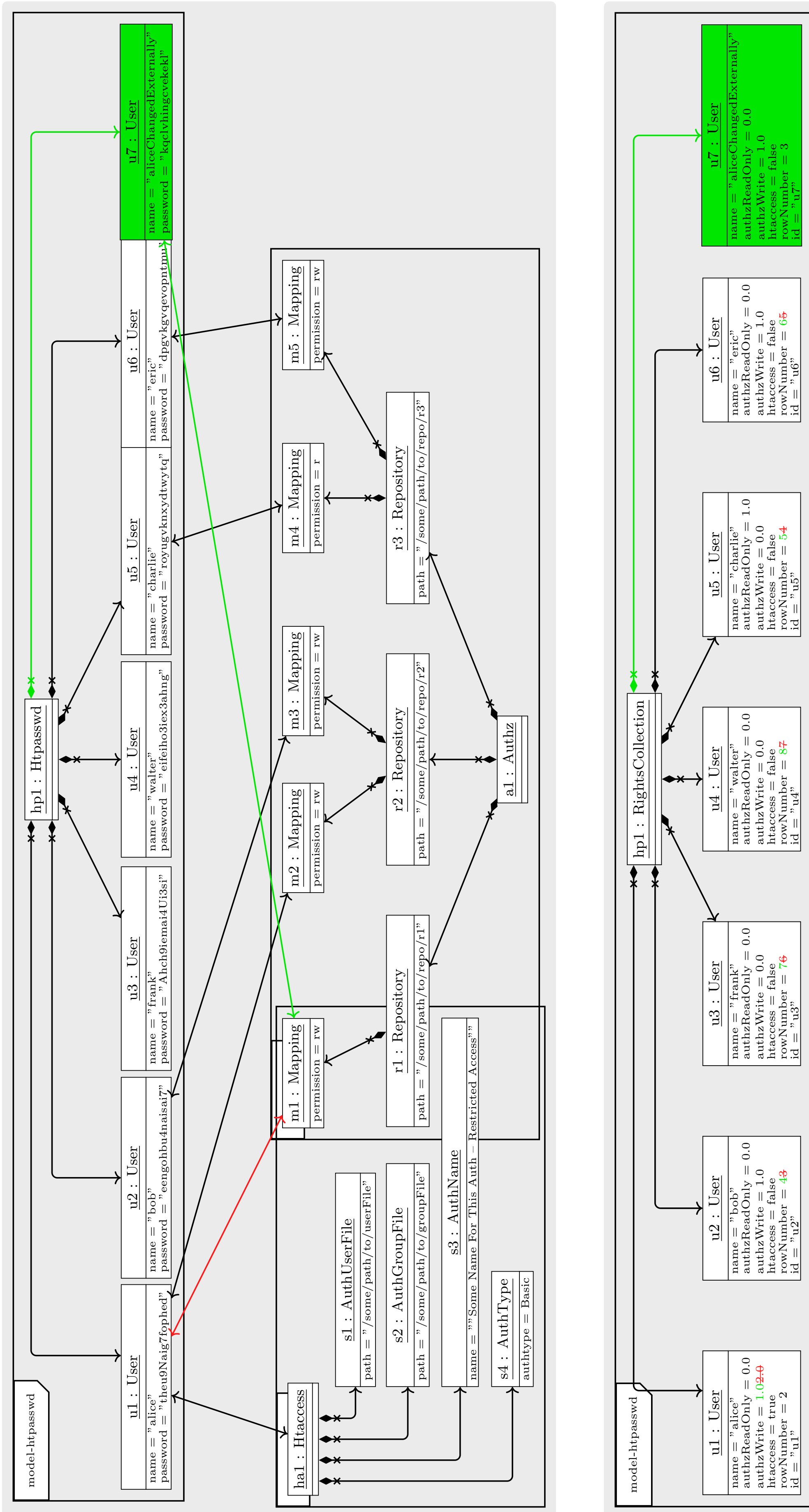
```
1 #a1
2 [/some/path/to/repo/r1] #r1
3 alice aliceChangedExternally = rw #m1
4 [/some/path/to/repo/r2] #r2
5 alice = rw #m2
6 bob = rw #m3
```

```
 8 [/some/path/to/repo/r3] #r3
 9 charlie = r #m4
10 eric = rw #m5
```

The model with highlighted changes is represented graphically in the figure before on the right.

- In Htaccess, no changes are expected in the model.

- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically (left):


model-htpasswd
hp1 : Htpasswd
u1 : User
name = "alice"
u2 : User
name = "bob"
u3 : User
name = "frank"
u4 : User
name = "walter"
u5 : User
name = "charlie"
u6 : User
name = "eric"
u7 : User
name = "aliceChangedExternally"
m1 : Mapping
permission = rw
m2 : Mapping
permission = rw
m3 : Mapping
permission = rw
m4 : Mapping
permission = r
m5 : Mapping
permission = rw
r1 : Repository
path = "/some/path/to/repo/r1"
r2 : Repository
path = "/some/path/to/repo/r2"
r3 : Repository
path = "/some/path/to/repo/r3"
a1 : Authz
ha1 : Htaccess
s1 : AuthUserFile
path = "/some/path/to/userFile"
s2 : AuthGroupFile
path = "/some/path/to/groupFile"
s3 : AuthName
name = ""Some Name For This Auth - Restricted Access""
s4 : AuthType
authtype = Basic
model-htpasswd
hp1 : RightsCollection
u1 : User
name = "alice"
authzReadOnly = 0.0
authzWrite = 1.0 2.0
htaccess = true
rowNumber = 2
id = "u1"
u2 : User
name = "bob"
authzReadOnly = 0.0
authzWrite = 1.0
htaccess = false
rowNumber = 4 3
id = "u2"
u3 : User
name = "frank"
authzReadOnly = 0.0
authzWrite = 0.0
htaccess = false
rowNumber = 7 6
id = "u3"
u4 : User
name = "walter"
authzReadOnly = 0.0
authzWrite = 0.0
htaccess = false
rowNumber = 8 7
id = "u4"
u5 : User
name = "charlie"
authzReadOnly = 1.0
authzWrite = 0.0
htaccess = false
rowNumber = 5 4
id = "u5"
u6 : User
name = "eric"
authzReadOnly = 0.0
authzWrite = 1.0
htaccess = false
rowNumber = 6 5
id = "u6"
u7 : User
name = "aliceChangedExternally"
authzReadOnly = 0.0
authzWrite = 1.0
htaccess = false
rowNumber = 3
id = "u7"

- In Overview, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

	A	B	C	D	E	F	G	H
1	name	authzReadOr	authzWrite	htaccess	id			
2	alice	0	1	TRUE	u1			
3	aliceChangec	0	1	FALSE	u7			
4	bob	0	1	FALSE	u2			
5	charlie	1	0	FALSE	u5			
6	eric	0	1	FALSE	u6			
7	frank	0	0	FALSE	u3			
8	walter	0	0	FALSE	u4			
9								
10								

  The model with highlighted changes is represented graphically in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Authz}}$ applied to Authz are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.4.3 Scenario: removed Mapping, reload externally (Authz)

Removing an Authz access right of a user with multiple rights leads only to the removal of this right. This test case tests mainly C 1 b.

The *User* applies the desired changes to Authz by changing its external representation. The model with highlighted changes is represented with its concrete rendering:

```
1  #a1
2  [/some/path/to/repo/r1] #r1
3  alice = rw #m1

5  [/some/path/to/repo/r2] #r2
6  alice = rw #m2
7  bob = rw #m3

9  [/some/path/to/repo/r3] #r3
10 charlie = r #m4
11 eric = rw #m5
```

The model differences are represented in textual form:

```
1 r1.remove(mappings, 0, m1)
2 m1.remove(authzUsername, 0, "alice")
3 m1.remove(permission, 0, e34)
4 m1.remove(id, 0, "m1")
5 m1.changeNamespace(model-authz ⇒ null)
6 m1.changeType(authz.Mapping ⇒ null)
7 m1.deleteInstance()
```

${}^{\text{User}}\Delta^{14}_{\text{Authz}}$

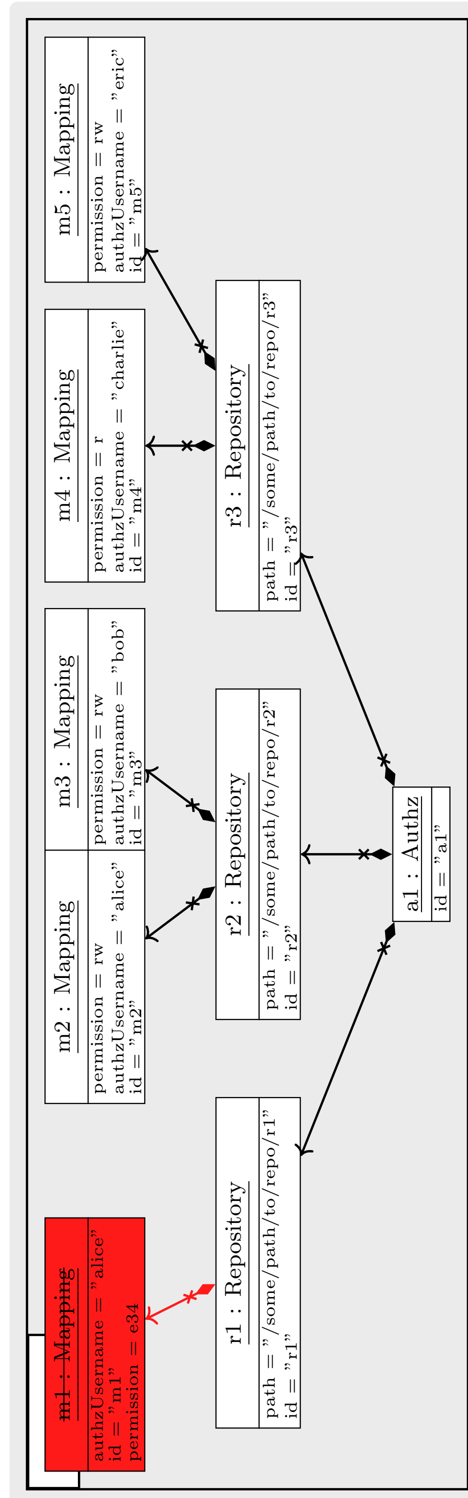
As result after completing the synchronization, the following changes are expected:

- In Htpasswd, no changes are expected in the model.

- In Authz, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Authz}}$). The model with highlighted changes is represented with its concrete rendering:

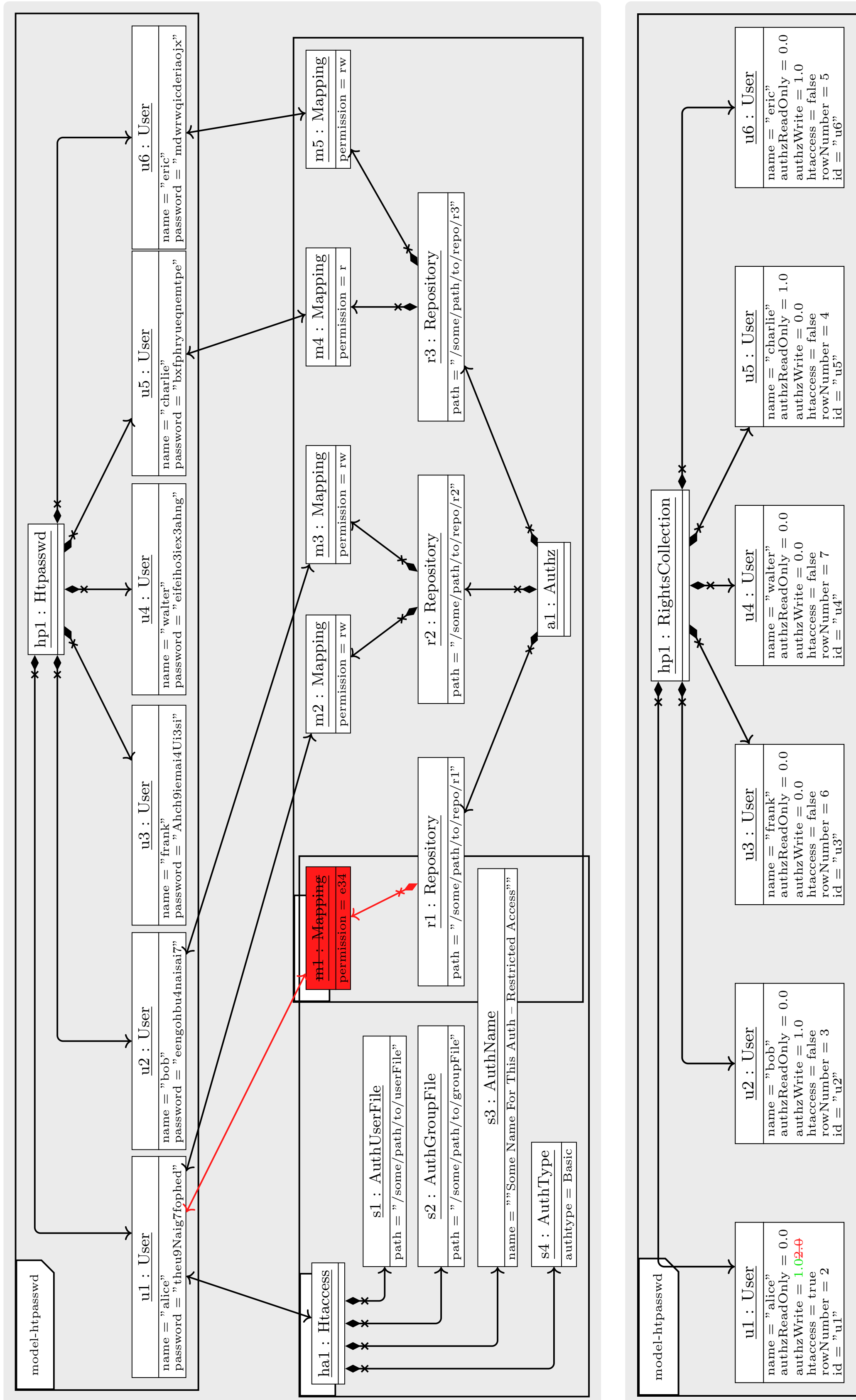
```
1  #a1
2  [/some/path/to/repo/r1] #r1
3  alice = rw #m1

5  [/some/path/to/repo/r2] #r2
6  alice = rw #m2
7  bob = rw #m3

9  [/some/path/to/repo/r3] #r3
10 charlie = r #m4
11 eric = rw #m5
```

The model with highlighted changes is represented graphically:



- In Htaccess, no changes are expected in the model.

- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically (left):



- In Overview, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

A1 | fx name

	A	B	C	D	E	F	G	H
1	name	authzReadOr	authzWrite	htaccess	id			
2	alice	0	1	TRUE	u1			
3	bob	0	1	FALSE	u2			
4	charlie	1	0	FALSE	u5			
5	eric	0	1	FALSE	u6			
6	frank	0	0	FALSE	u3			
7	walter	0	0	FALSE	u4			
8								
9								

The model with highlighted changes is represented graphically in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Authz}}$ applied to Authz are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.4.4 Scenario: added Mapping, reload externally (Authz)

Adding a new Authz access right without matching user leads to the creation of such a user. This test case tests mainly C 1 a.

The *User* applies the desired changes to Authz by changing its external representation. The model with highlighted changes is represented with its concrete rendering:

```
1  #a1
2  [/some/path/to/repo/r1] #r1
3  alice = rw #m1

5  [/some/path/to/repo/r2] #r2
6  alice = rw #m2
7  bob = rw #m3

9  [/some/path/to/repo/r3] #r3
10 charlie = r #m4
11 eric = rw #m5
12 dennis = r
```

The model differences are represented in textual form:

```
1 m6.createInstance()
2 m6.changeNamespace(null ⇒ model-authz)
3 m6.changeType(null ⇒ authz.Mapping)
4 m6.add(permission, 0, 447)
5 r3.add(mappings, 2, m6)
6 m6.add(authzUsername, 0, "dennis")
```

${}^{\text{User}}\Delta^{14}_{\text{Authz}}$

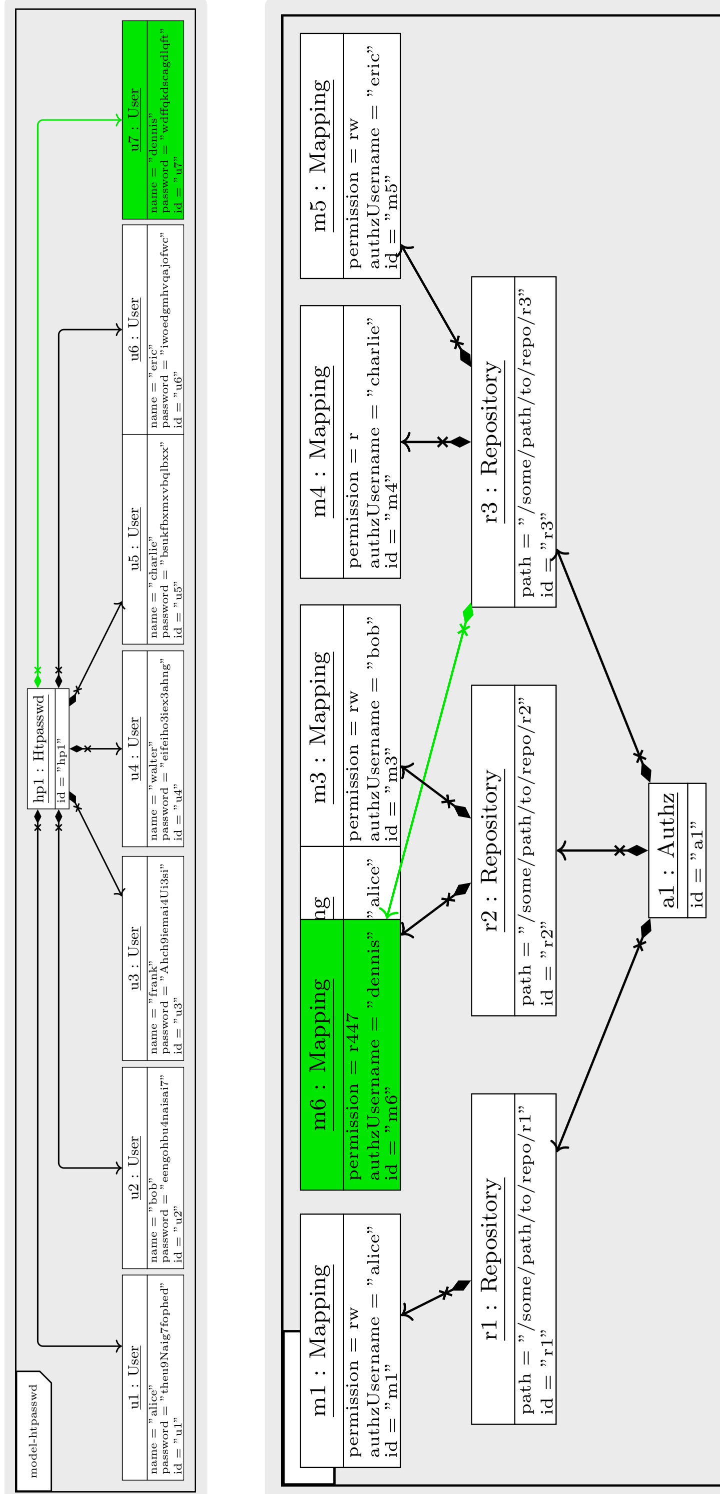
As result after completing the synchronization, the following changes are expected:

- In Htpasswd, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #hp1
```

```
2  alice : theu9Naig7fophed #u1
3 bob : eengohbu4naisai7 #u2
4 frank : Ahch9iemai4Ui3si #u3
5 walter : eifeiho3iex3ahng #u4
6 charlie : bsukfbxmxvbqlbxx #u5
7 eric : iwoedgmhvqajofwc #u6
8 dennis : wdffqkdscagdlqft #u7
```

The model with highlighted changes is represented graphically (left):



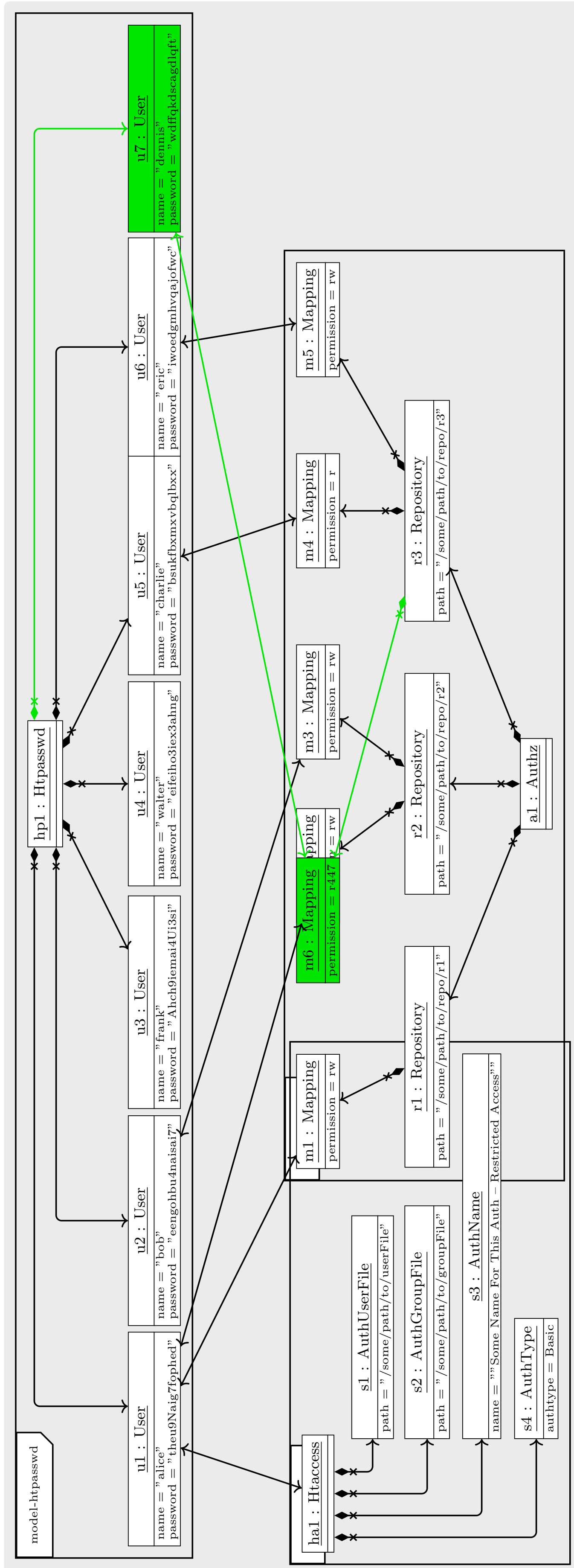
- In Authz, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Authz}}$). The model with highlighted changes is represented with its concrete rendering:

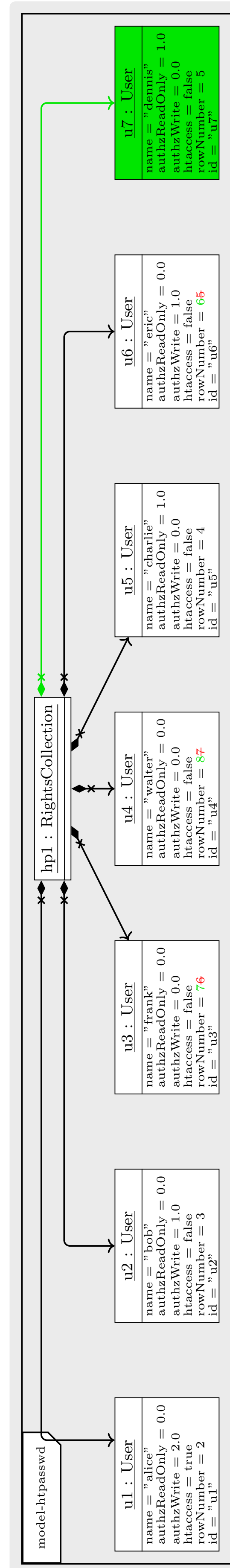
```
1  #a1
2  [/some/path/to/repo/r1] #r1
3  alice = rw #m1

5  [/some/path/to/repo/r2] #r2
6  alice = rw #m2
7  bob = rw #m3

9  [/some/path/to/repo/r3] #r3
10 charlie = r #m4
11 eric = rw #m5
12 dennis = r #m6
```

The model with highlighted changes is represented graphically in the figure before on the right.

- In Htaccess, no changes are expected in the model.

- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically (left):


model-htpasswd
hp1 : Htpasswd
u1 : User
name = "alice"
password = "theu9Naig7fophed"
u2 : User
name = "bob"
password = "eengohbu4naisai7"
u3 : User
name = "frank"
password = "Ahch9iemai4Ui3si"
u4 : User
name = "walter"
password = "eifeiho3iex3ahng"
u5 : User
name = "charlie"
password = "bsukfbxmxvbqlbxx"
u6 : User
name = "eric"
password = "iwoedgmhvqajofwc"
u7 : User
name = "dennis"
password = "wdffqkdscagdlqft"
ha1 : Htaccess
s1 : AuthUserFile
path = "/some/path/to/userFile"
s2 : AuthGroupFile
path = "/some/path/to/groupFile"
s3 : AuthName
name = ""Some Name For This Auth - Restricted Access""
s4 : AuthType
authtype = Basic
m1 : Mapping
permission = rw
r1 : Repository
path = "/some/path/to/repo/r1"
m6 : Mapping
permission = r447
m3 : Mapping
permission = rw
r2 : Repository
path = "/some/path/to/repo/r2"
m4 : Mapping
permission = r
m5 : Mapping
permission = rw
r3 : Repository
path = "/some/path/to/repo/r3"
a1 : Authz


model-htpasswd
hp1 : RightsCollection
u1 : User
name = "alice"
authzReadOnly = 0.0
authzWrite = 2.0
htaccess = true
rowNumber = 2
id = "u1"
u2 : User
name = "bob"
authzReadOnly = 0.0
authzWrite = 1.0
htaccess = false
rowNumber = 3
id = "u2"
u3 : User
name = "frank"
authzReadOnly = 0.0
authzWrite = 0.0
htaccess = false
rowNumber = 76
id = "u3"
u4 : User
name = "walter"
authzReadOnly = 0.0
authzWrite = 0.0
htaccess = false
rowNumber = 87
id = "u4"
u5 : User
name = "charlie"
authzReadOnly = 1.0
authzWrite = 0.0
htaccess = false
rowNumber = 4
id = "u5"
u6 : User
name = "eric"
authzReadOnly = 0.0
authzWrite = 1.0
htaccess = false
rowNumber = 65
id = "u6"
u7 : User
name = "dennis"
authzReadOnly = 1.0
authzWrite = 0.0
htaccess = false
rowNumber = 5
id = "u7"

- In Overview, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

	A	B	C	D	E	F	G	H
1	name	authzReadOr	authzWrite	htaccess	id			
2	alice	0	2	TRUE	u1			
3	bob	0	1	FALSE	u2			
4	charlie	1	0	FALSE	u5			
5	dennis	1	0	FALSE	u7			
6	eric	0	1	FALSE	u6			
7	frank	0	0	FALSE	u3			
8	walter	0	0	FALSE	u4			
9								
10								

  The model with highlighted changes is represented graphically in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Authz}}$ applied to Authz are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.4.5 Scenario: added Mapping, reload externally (Authz)

Adding a new Authz access right with matching user leads to its linkage to that user. This test case tests mainly C 1 a.

The *User* applies the desired changes to Authz by changing its external representation. The model with highlighted changes is represented with its concrete rendering:

```
1  #a1
2  [/some/path/to/repo/r1] #r1
3  alice = rw #m1
4  frank = r
5  [/some/path/to/repo/r2] #r2
6  alice = rw #m2
7  bob = rw #m3

9  [/some/path/to/repo/r3] #r3
10 charlie = r #m4
11 eric = rw #m5
```

The model differences are represented in textual form:

```
1  m6.createInstance()
2  m6.changeNamespace(null ⇒ model-authz)
3  m6.changeType(null ⇒ authz.Mapping)
4  m6.add(permission, 0, 447)
5  r1.add(mappings, 1, m6)
6  m6.add(authzUsername, 0, "frank")
```

${}^{\text{User}}\Delta^{14}_{\text{Authz}}$

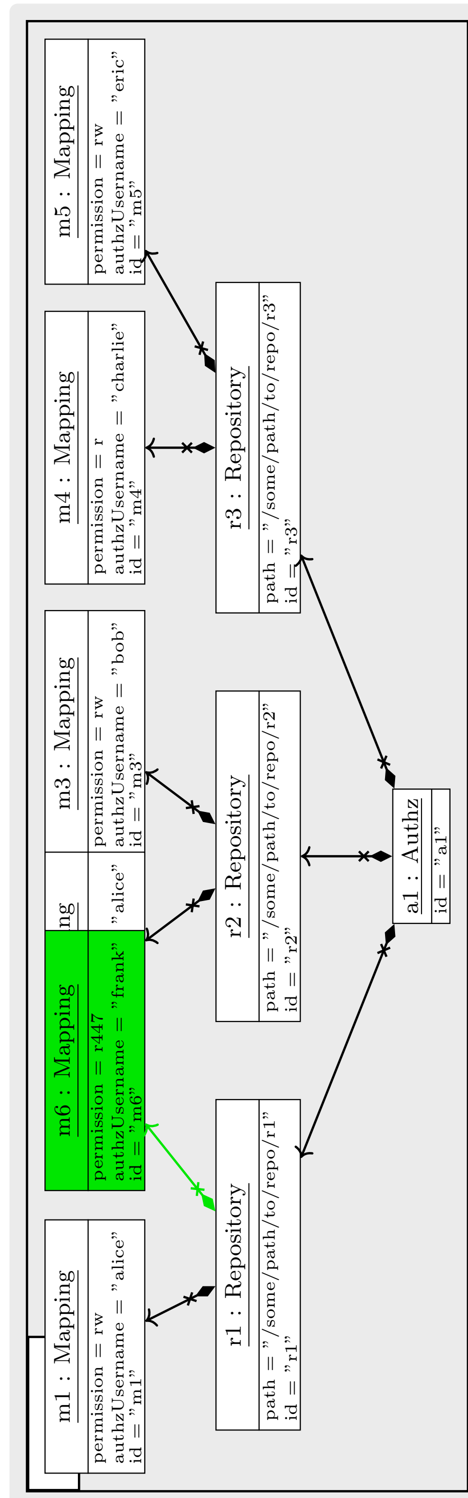
As result after completing the synchronization, the following changes are expected:

- In Htpasswd, no changes are expected in the model.

- In Authz, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Authz}}$). The model with highlighted changes is represented with its concrete rendering:

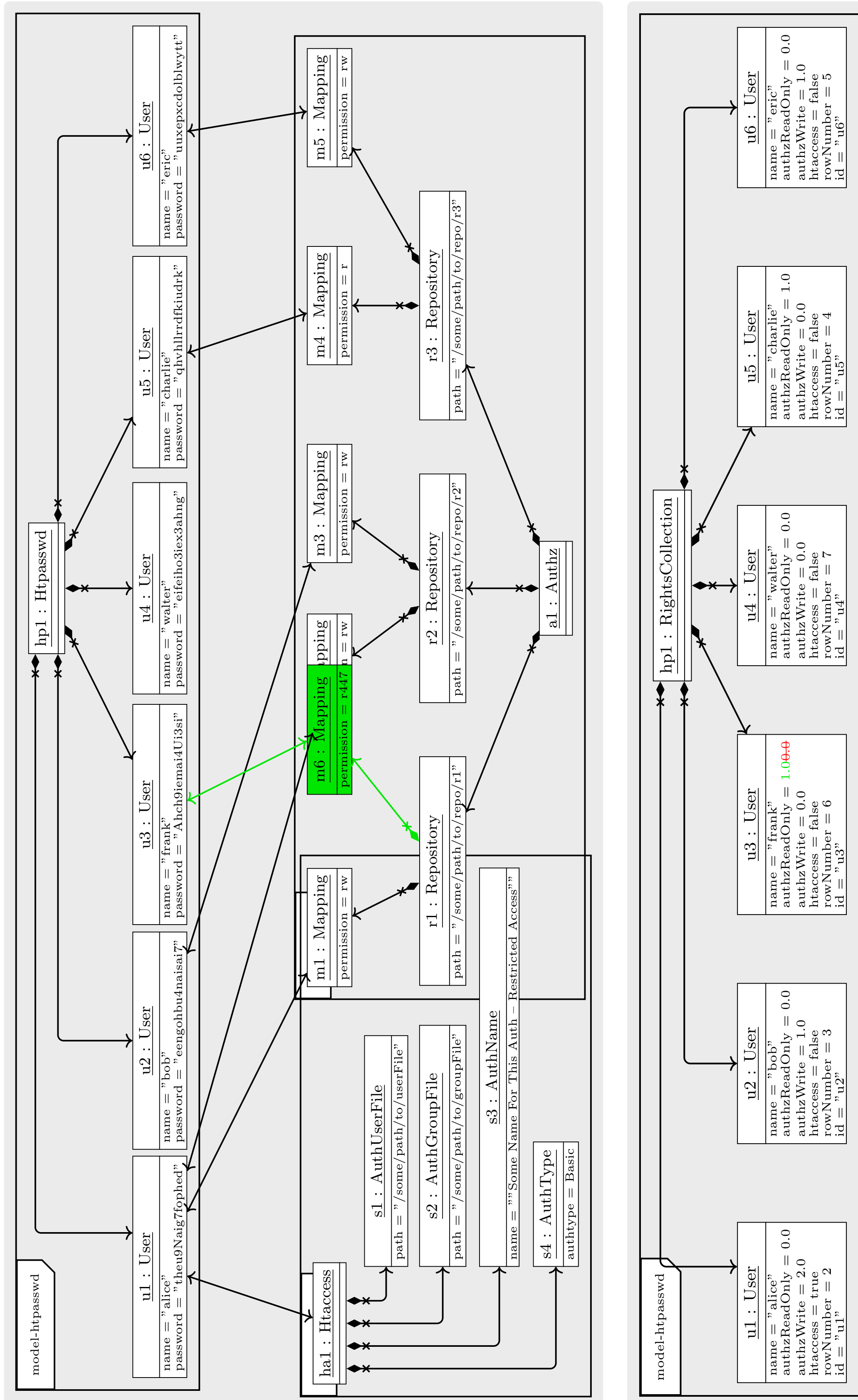
```
1  #a1
2  [/some/path/to/repo/r1] #r1
3  alice = rw #m1
4  frank = r #m6
5  [/some/path/to/repo/r2] #r2
6  alice = rw #m2
7  bob = rw #m3

9  [/some/path/to/repo/r3] #r3
10 charlie = r #m4
11 eric = rw #m5
```

The model with highlighted changes is represented graphically:



- In Htaccess, no changes are expected in the model.

- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically (left):



- In Overview, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

	A	B	C	D	E	F	G	H
1	name	authzReadOr	authzWrite	htaccess	id			
2	alice	0	2	TRUE	u1			
3	bob	0	1	FALSE	u2			
4	charlie	1	0	FALSE	u5			
5	eric	0	1	FALSE	u6			
6	frank	1	0	FALSE	u3			
7	walter	0	0	FALSE	u4			
8								
9								

The model with highlighted changes is represented graphically in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Authz}}$ applied to Authz are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.4.6 Scenario: removed Mapping, reload externally (Authz)

Removing an Authz access right of a user with only this right leads also to the removal of that user. This test case tests mainly C 1 b.

The *User* applies the desired changes to Authz by changing its external representation. The model with highlighted changes is represented with its concrete rendering:

```
1  #a1
2  [/some/path/to/repo/r1] #r1
3  alice = rw #m1
4
5  [/some/path/to/repo/r2] #r2
6  alice = rw #m2
7  bob = rw #m3
8
9  [/some/path/to/repo/r3] #r3
10 charlie = r #m4
11 eric = rw #m5
```

The model differences are represented in textual form:

```
1  r2.remove(mappings, 1, m3)
2  m3.remove(authzUsername, 0, "bob")
3  m3.remove(permission, 0, e34)
4  m3.remove(id, 0, "m3")
5  m3.changeNamespace(model-authz ⇒ null)
6  m3.changeType(authz.Mapping ⇒ null)
7  m3.deleteInstance()
```

${}^{\text{User}}\Delta^{14}_{\text{Authz}}$

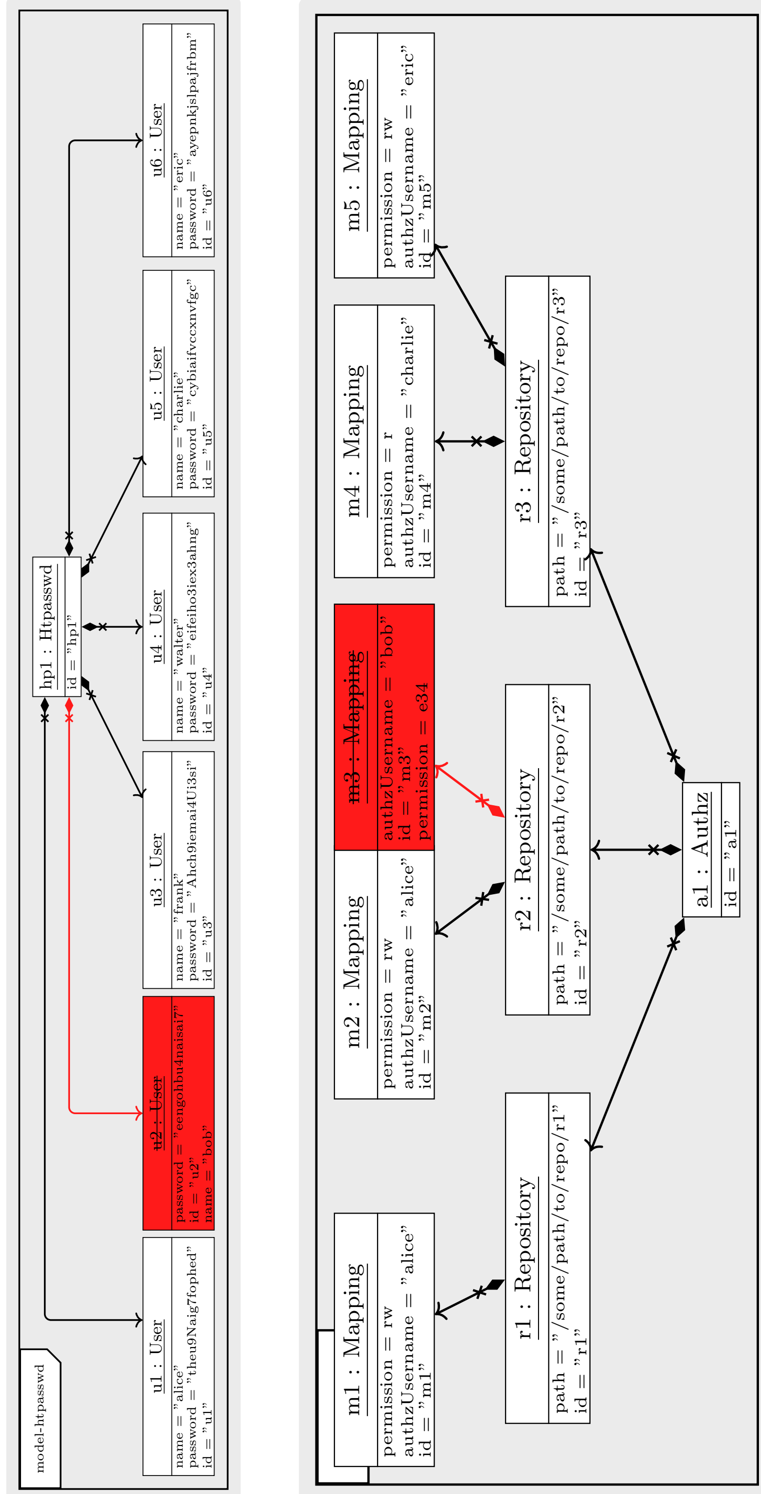
As result after completing the synchronization, the following changes are expected:

- In Htpasswd, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1  #hp1
```

```
2  alice : theu9Naig7fophed #u1
3 bob : eengohbu4naisai7 #u2
4 frank : Ahch9iemai4Ui3si #u3
5 walter : eifeiho3iex3ahng #u4
6 charlie : cybiaifvccxnvfgc #u5
7 eric : ayepnkjslpajfrbm #u6
```

The model with highlighted changes is represented graphically (left):



- In Authz, some changes are expected in the model (including the user changes

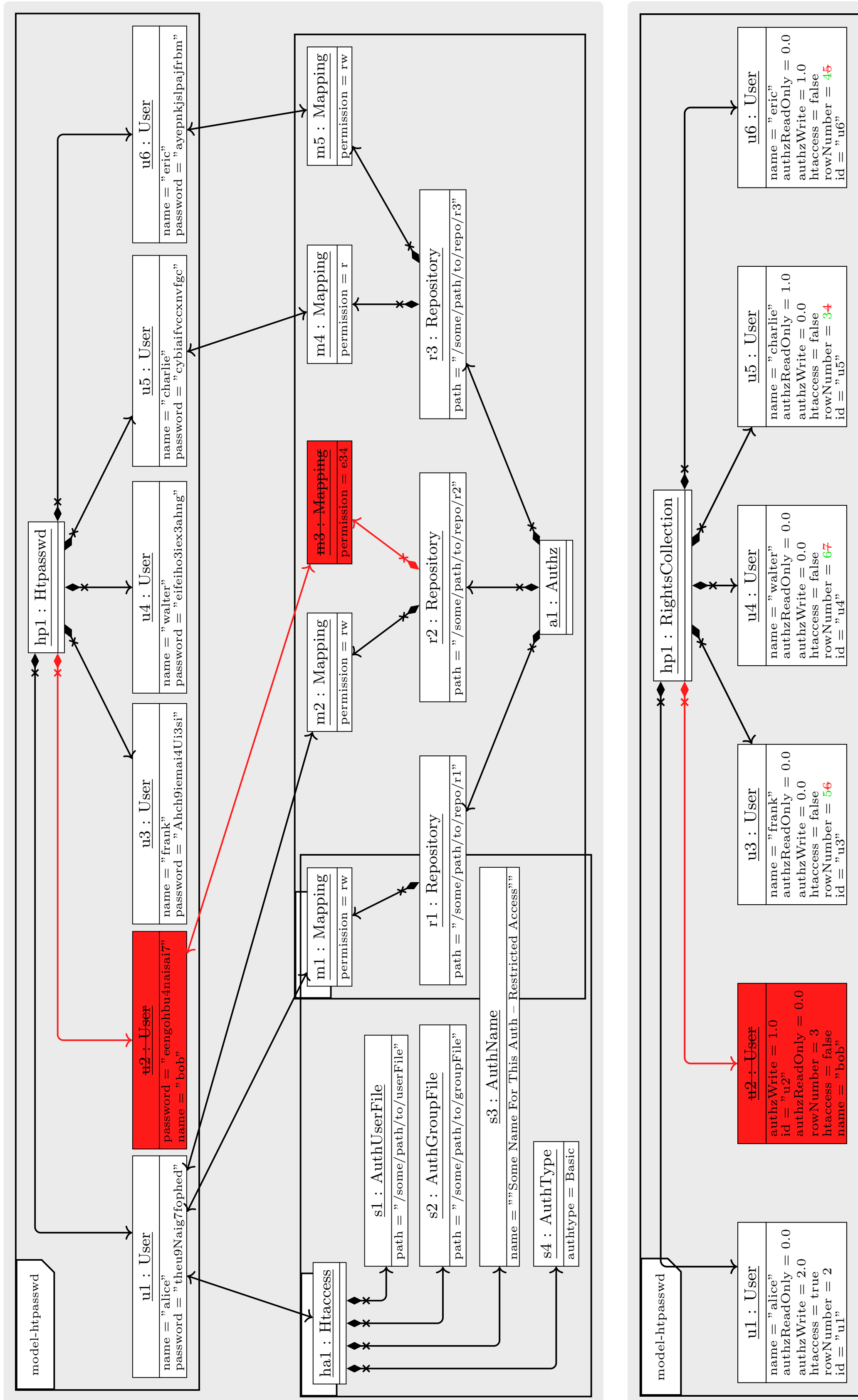
${}^{\text{User}}\Delta^{14}_{\text{Authz}}$). The model with highlighted changes is represented with its concrete rendering:

```
#a1
[/some/path/to/repo/r1] #r1
alice = rw #m1

[/some/path/to/repo/r2] #r2
alice = rw #m2
bob = rw #m3

[/some/path/to/repo/r3] #r3
charlie = r #m4
eric = rw #m5
```

The model with highlighted changes is represented graphically in the figure before on the right.

- In Htaccess, no changes are expected in the model.

- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically (left):



- In Overview, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

A1 | fx | name

	A	B	C	D	E	F	G	H
1	name	authzReadOr	authzWrite	htaccess	id			
2	alice	0	2	TRUE	u1			
3	charlie	1	0	FALSE	u5			
4	eric	0	1	FALSE	u6			
5	frank	0	0	FALSE	u3			
6	walter	0	0	FALSE	u4			
7								
8								
9								

The model with highlighted changes is represented graphically in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{User}\Delta^{14}_{Authz}$ applied to Authz are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.4.7 Scenario: renamed HtaccessUser, reload externally (Htaccess)

Renaming a Htaccess right leads to the renaming of the corresponding user. Because of C 1 e, the rights of this user in Authz are also renamed. This test case tests mainly C 2 d.

The *User* applies the desired changes to Htaccess by changing its external representation. The model with highlighted changes is represented with its concrete rendering:

```
1 #ha1
2  AuthUserFile /some/path/to/userFile #s1
3 AuthGroupFile /some/path/to/groupFile #s2
4 AuthName "Some Name For This Auth -- Restricted Access" #s3
5 AuthType Basic #s4
6 require user alice aliceChangedExternally #hu1
```

The model differences are represented in textual form:

```
1 hu1.remove(htaccessUsername, 0, "alice")
2 hu1.add(htaccessUsername, 0, "aliceChangedExternally")
```

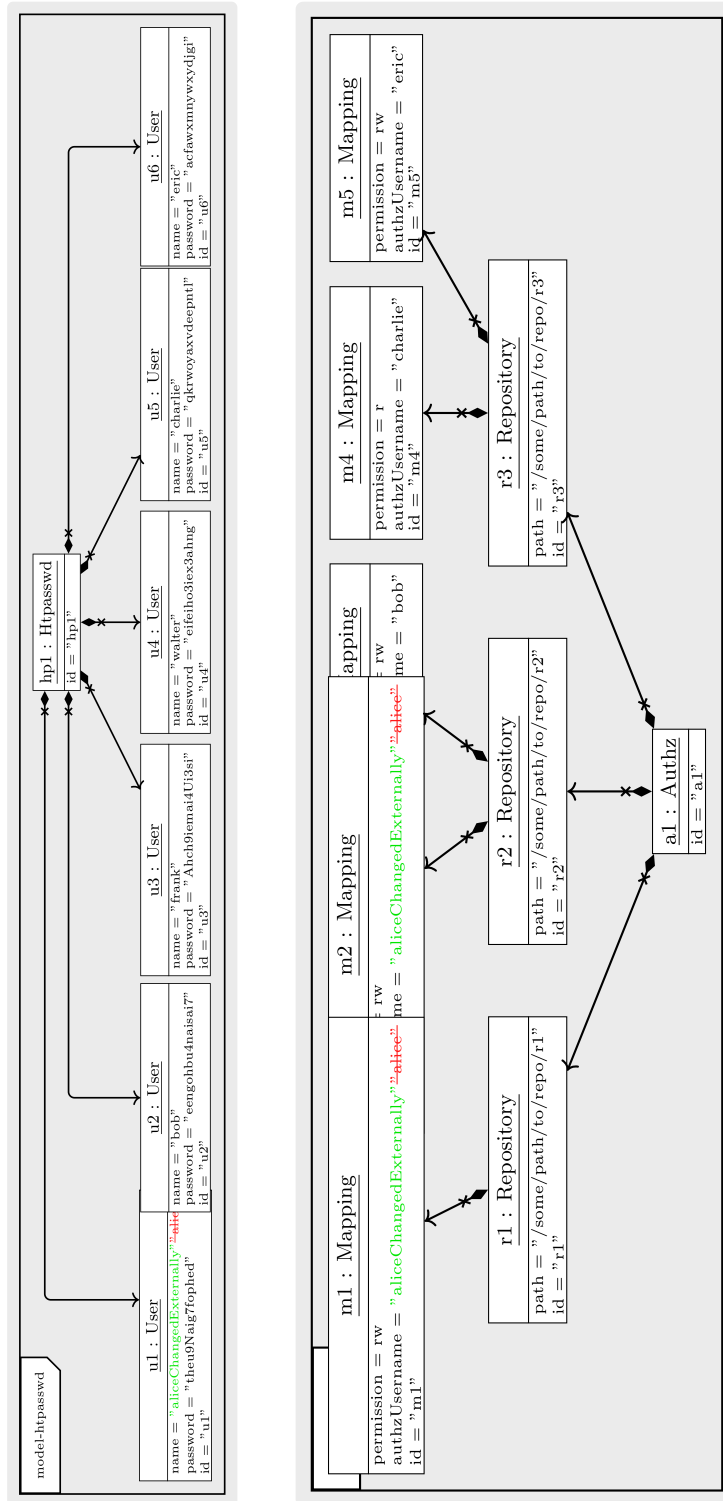
${}^{User}\Delta^{14}_{Htaccess}$

As result after completing the synchronization, the following changes are expected:

- In Htpasswd, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #hp1
2  alice aliceChangedExternally : theu9Naig7fophed #u1
3 bob : eengohbu4naisai7 #u2
4 frank : Ahch9iemai4Ui3si #u3
5 walter : eifeiho3iex3ahng #u4
6 charlie : qkrwoyaxvdeepntl #u5
7 eric : acfawxmnywxydjgi #u6
```

The model with highlighted changes is represented graphically (left):



- In Authz, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #a1
2 [/some/path/to/repo/r1] #r1
3 alice aliceChangedExternally = rw #m1
4 [/some/path/to/repo/r2] #r2
5 alice aliceChangedExternally = rw #m2
6 bob = rw #m3
```

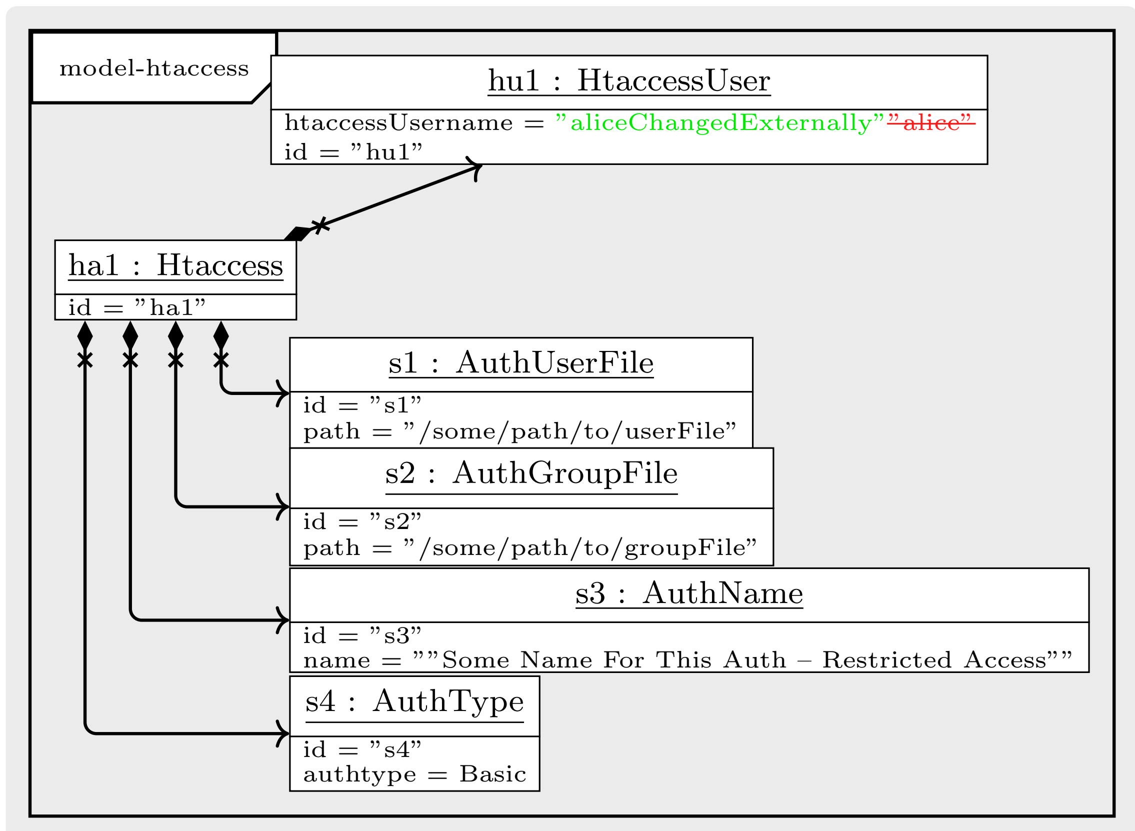
```
8  [/some/path/to/repo/r3] #r3
9  charlie = r #m4
10 eric = rw #m5
```

The model with highlighted changes is represented graphically in the figure before on the right.

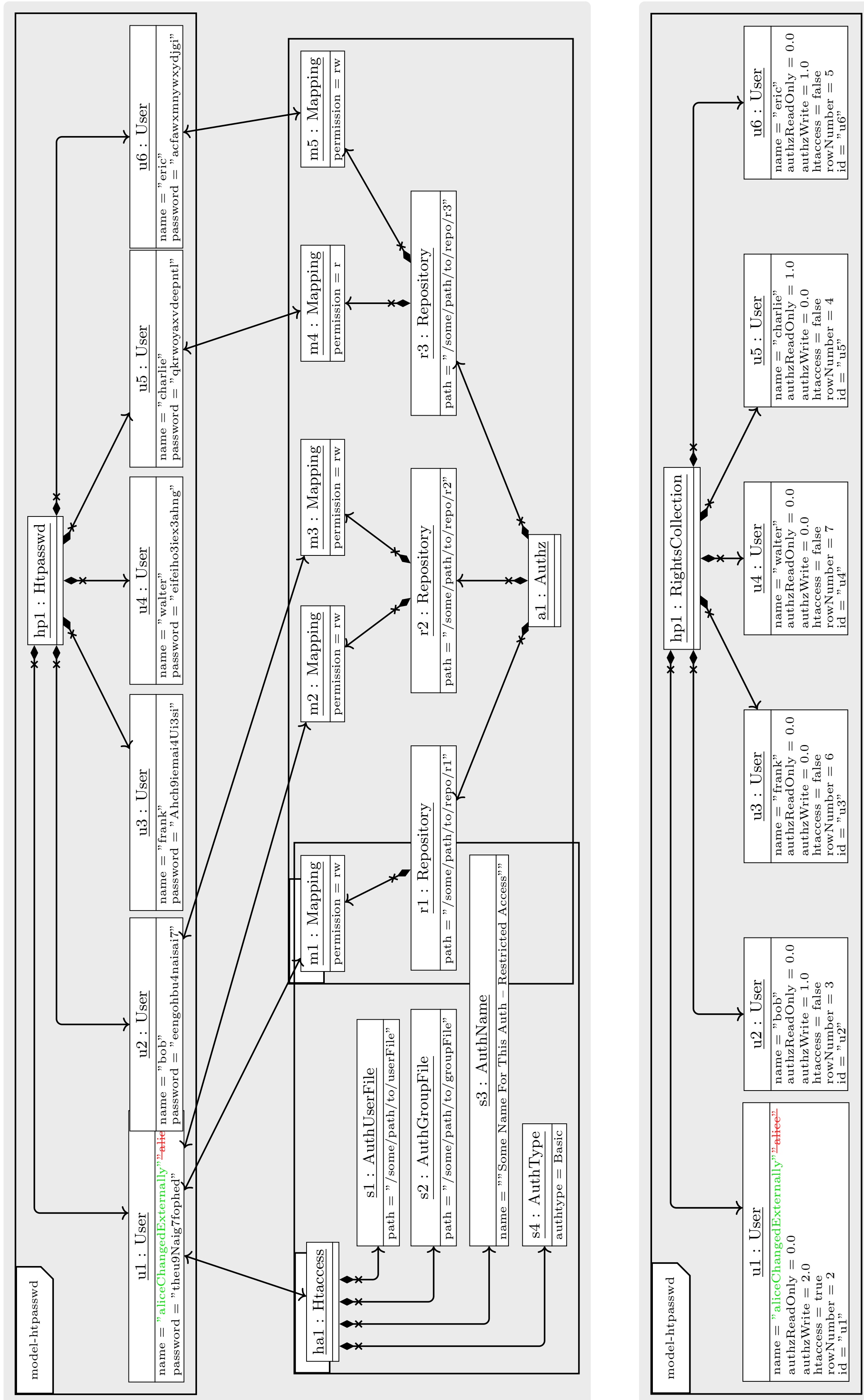
- In Htaccess, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Htaccess}}$). The model with highlighted changes is represented with its concrete rendering:

```
1 #ha1
2  AuthUserFile /some/path/to/userFile #s1
3 AuthGroupFile /some/path/to/groupFile #s2
4 AuthName "Some Name For This Auth -- Restricted Access" #s3
5 AuthType Basic #s4
6 require user alice aliceChangedExternally #hu1
```

The model with highlighted changes is represented graphically:



- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically (left):



- In Overview, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

A1 | fx name

	A	B	C	D	E	F	G	H
1	name	authzReadOr	authzWrite	htaccess	id			
2	aliceChangec	0	2	TRUE	u1			
3	bob	0	1	FALSE	u2			
4	charlie	1	0	FALSE	u5			
5	eric	0	1	FALSE	u6			
6	frank	0	0	FALSE	u3			
7	walter	0	0	FALSE	u4			
8								
9								
10								

The model with highlighted changes is represented graphically in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes $^{\text{User}}\Delta^{14}_{\text{Htaccess}}$ applied to Htaccess are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.4.8 Scenario: change Htaccess right for Bob (Overview)

Changing the Htaccess right in the new Overview view will be propagated back to the SUM.

The *User* applies the desired changes to Overview by changing its internal EMF model. Therefore, the external representation is not yet updated. The model differences are represented in textual form:

```
1 u2.remove(htaccess, 0, "false")
2 u2.add(htaccess, 0, "true")
```

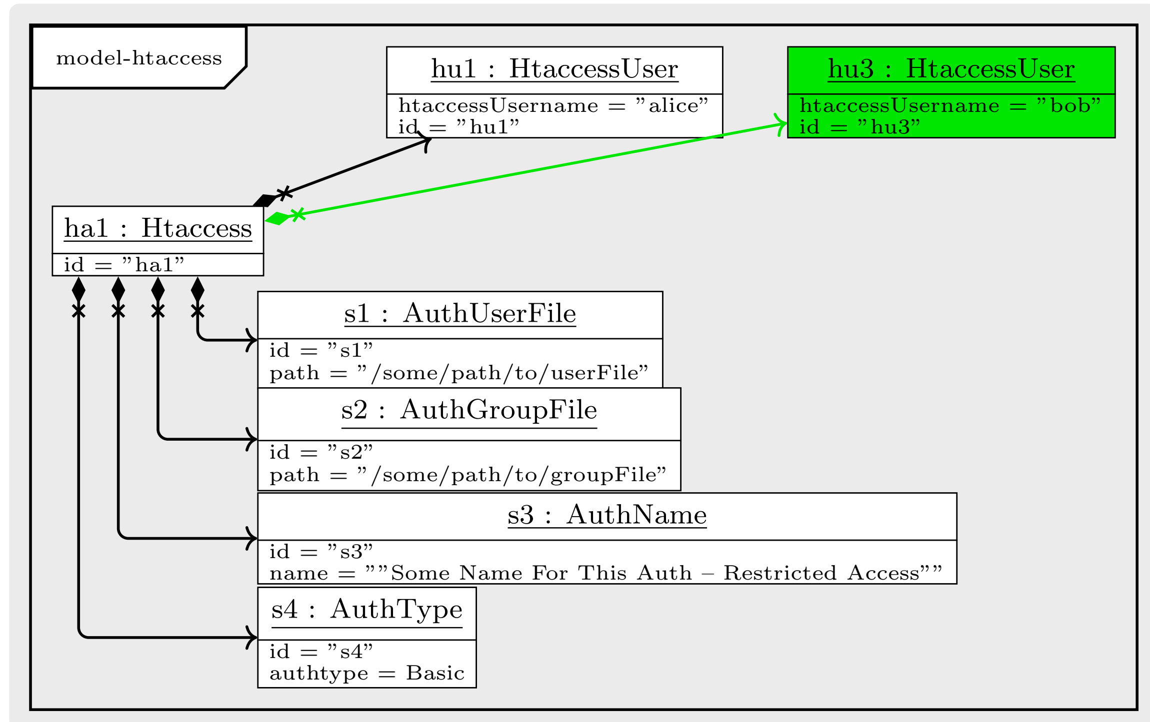
$^{\text{User}}\Delta^{14}_{\text{Overview}}$

As result after completing the synchronization, the following changes are expected:

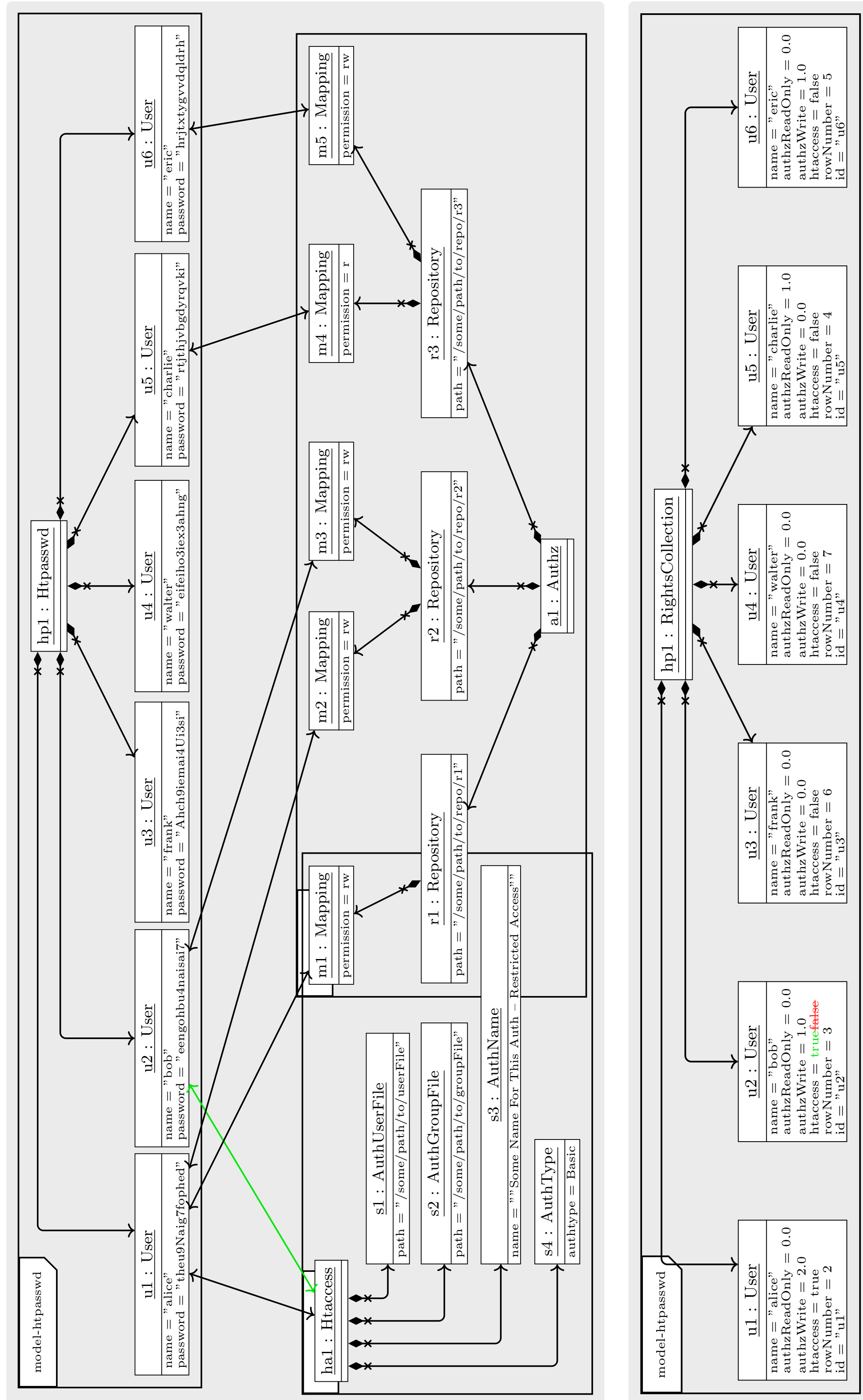
- In Htpasswd, no changes are expected in the model.
- In Authz, no changes are expected in the model.
- In Htaccess, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #ha1
2  AuthUserFile /some/path/to/userFile #s1
3 AuthGroupFile /some/path/to/groupFile #s2
4 AuthName "Some Name For This Auth -- Restricted Access" #s3
5 AuthType Basic #s4
6 require user bob #hu3
7 require user alice #hu1
```

The model with highlighted changes is represented graphically:



- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically (left):



- In Overview, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Overview}}$). The model with highlighted changes is represented with its concrete rendering:

A1 | fx | name

	A	B	C	D	E	F	G	H
1	name	authzReadOr	authzWrite	htaccess	id			
2	alice	0	2	TRUE	u1			
3	bob	0	1	TRUE	u2			
4	charlie	1	0	FALSE	u5			
5	eric	0	1	FALSE	u6			
6	frank	0	0	FALSE	u3			
7	walter	0	0	FALSE	u4			
8								
9								

The model with highlighted changes is represented graphically in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Overview}}$ applied to Overview are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.4.9 Scenario: change number of Authz rights for Bob (Overview)

Changes of the numbers of Authz rights inside the new Overview view will be reverted.

The *User* applies the desired changes to Overview by changing its internal EMF model. Therefore, the external representation is not yet updated. The model differences are represented in textual form:

```
1 u2.remove(authzReadOnly, 0, "0.0")
2 u2.add(authzReadOnly, 0, "5.0")
```

${}^{\text{User}}\Delta^{14}_{\text{Overview}}$

As result after completing the synchronization, the following changes are expected:

- In Htpasswd, no changes are expected in the model.
- In Authz, no changes are expected in the model.
- In Htaccess, no changes are expected in the model.
- In SUM, no changes are expected in the model.
- In Overview, no changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Overview}}$, which are reverted due to the viewpoint definition).

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Overview}}$ applied to Overview are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.4.10 Scenario: change the name of Bob to David (Overview)

Changing the name in the new Overview view will be propagated back to the SUM. Additionally, the order of users in Overview is changed, since Bob/David change the position with Charlie.

The *User* applies the desired changes to Overview by changing its internal EMF model. Therefore, the external representation is not yet updated. The model differences are represented in textual form:

```
1  u2.remove(name, 0, "bob")
2  u2.add(name, 0, "david")
```

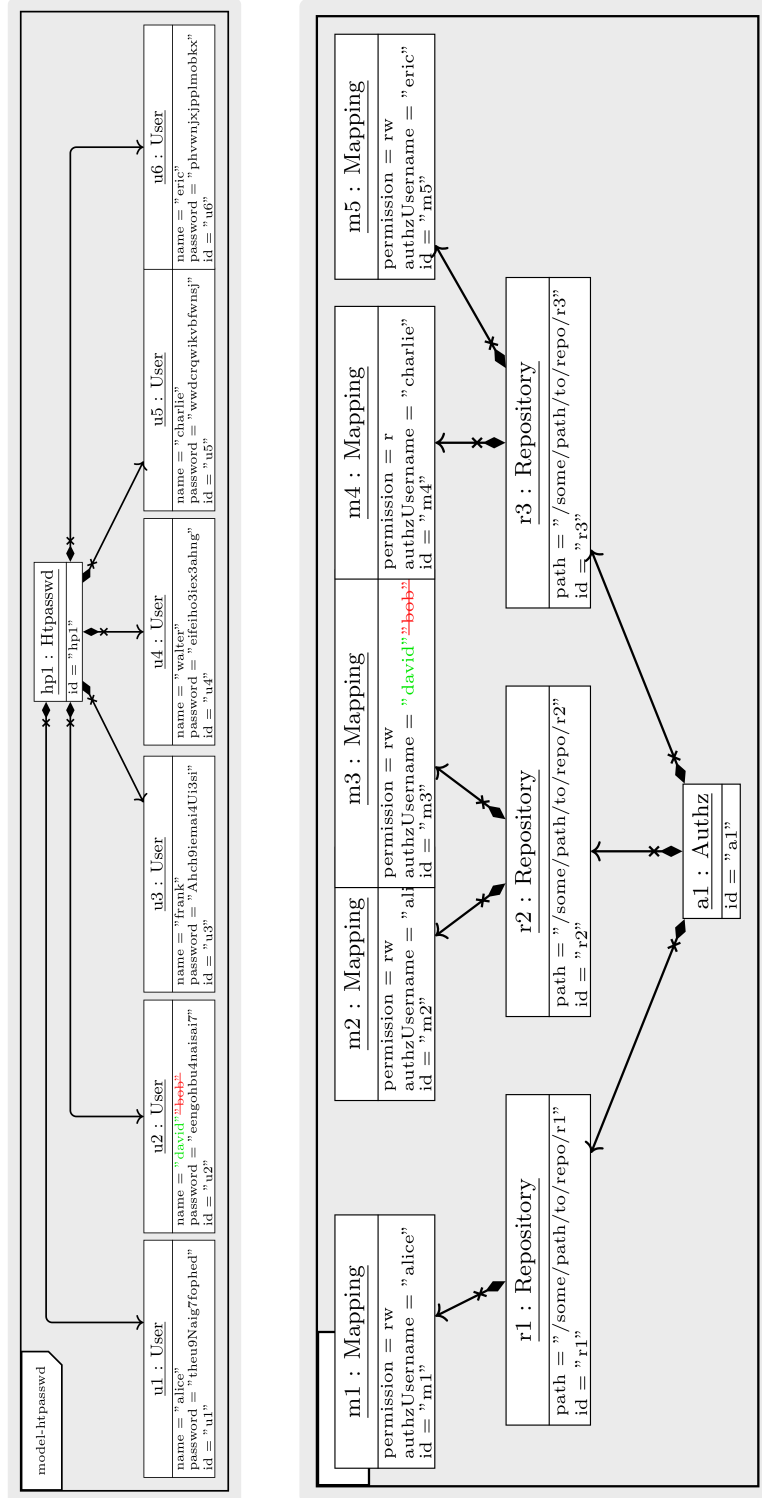
User $\Delta^{14}_{\text{Overview}}$

As result after completing the synchronization, the following changes are expected:

- In Htpasswd, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #hp1
2   alice : theu9Naig7fophed #u1
3 bobdavid : eengohbu4naisai7 #u2
4 frank : Ahch9iemai4Ui3si #u3
5 walter : eifeiho3iex3ahng #u4
6 charlie : wwdcrqwikvbfwnsj #u5
7 eric : phvwnjxjpplmobkx #u6
```

The model with highlighted changes is represented graphically (left):



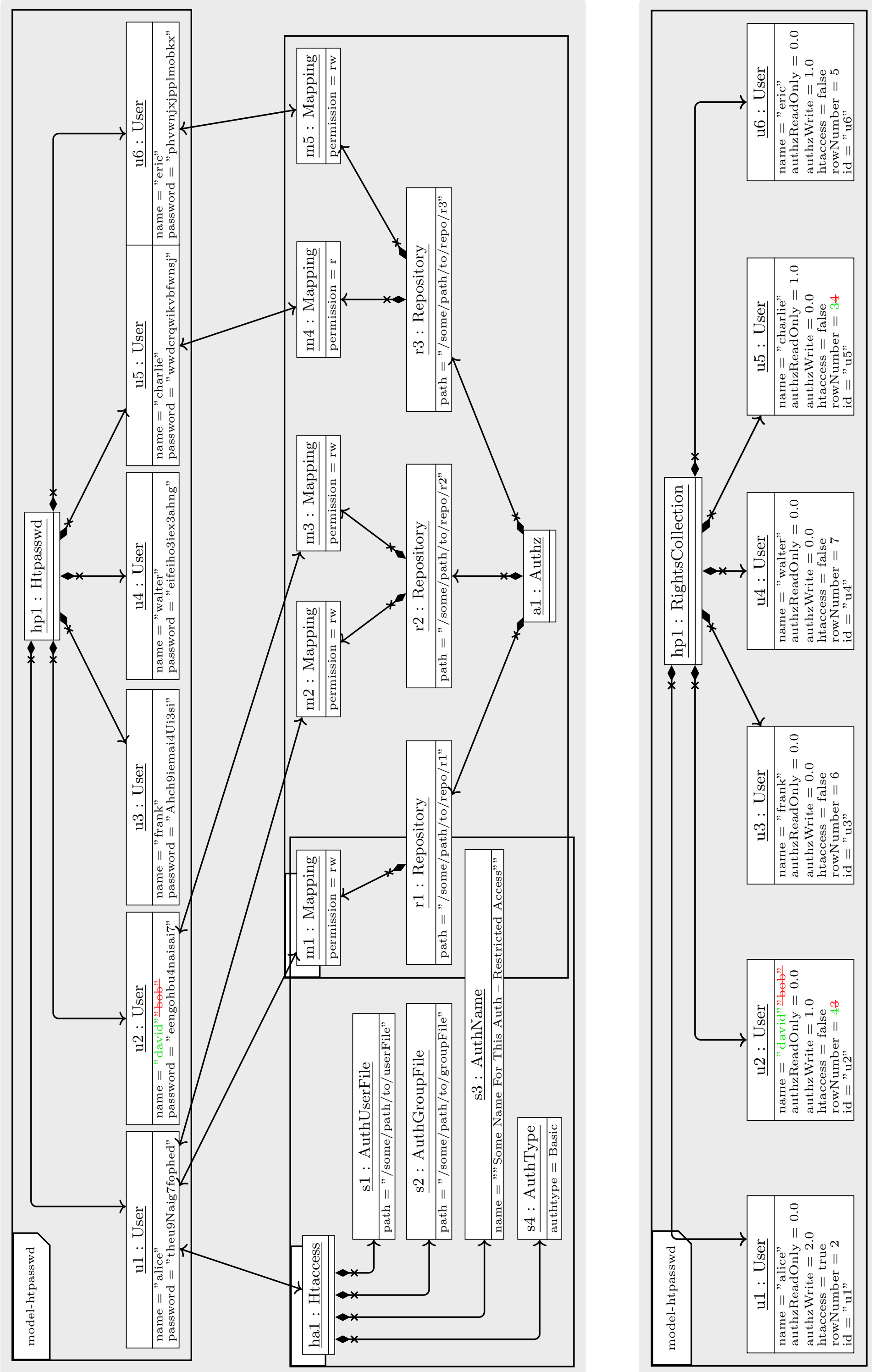
- In Authz, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #a1
2 [/some/path/to/repo/r1] #r1
3 alice = rw #m1

5 [/some/path/to/repo/r2] #r2
6 alice = rw #m2
7 bobdavid = rw #m3
```

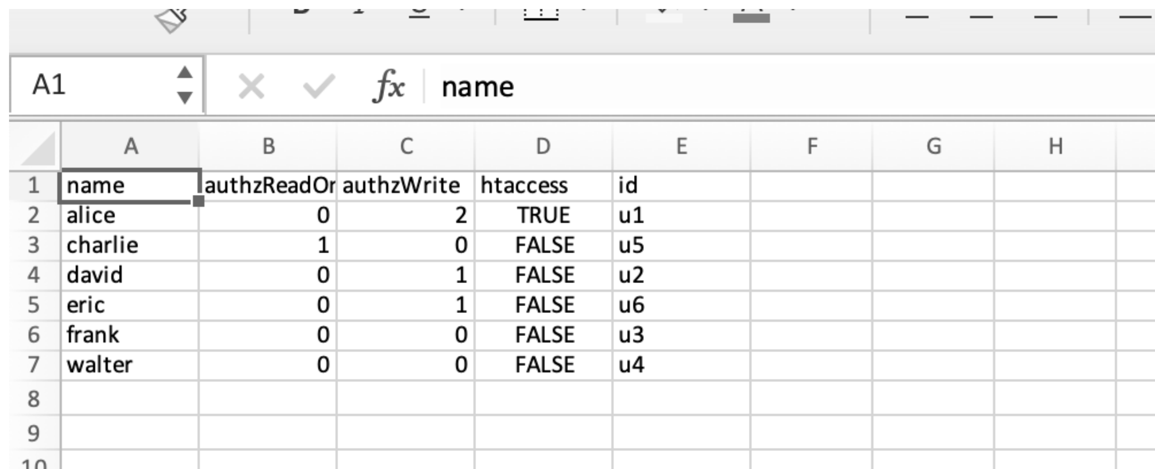
```
8  [/some/path/to/repo/r3] #r3
9  charlie = r #m4
10 eric = rw #m5
```

The model with highlighted changes is represented graphically in the figure before on the right.

- In Htaccess, no changes are expected in the model.
- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically (left):



- In Overview, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Overview}}$). The model with highlighted changes is represented with its concrete rendering:



A1 — name

	A	B	C	D	E
1	name	authzReadOr	authzWrite	htaccess	id
2	alice	0	2	TRUE	u1
3	charlie	1	0	FALSE	u5
4	david	0	1	FALSE	u2
5	eric	0	1	FALSE	u6
6	frank	0	0	FALSE	u3
7	walter	0	0	FALSE	u4

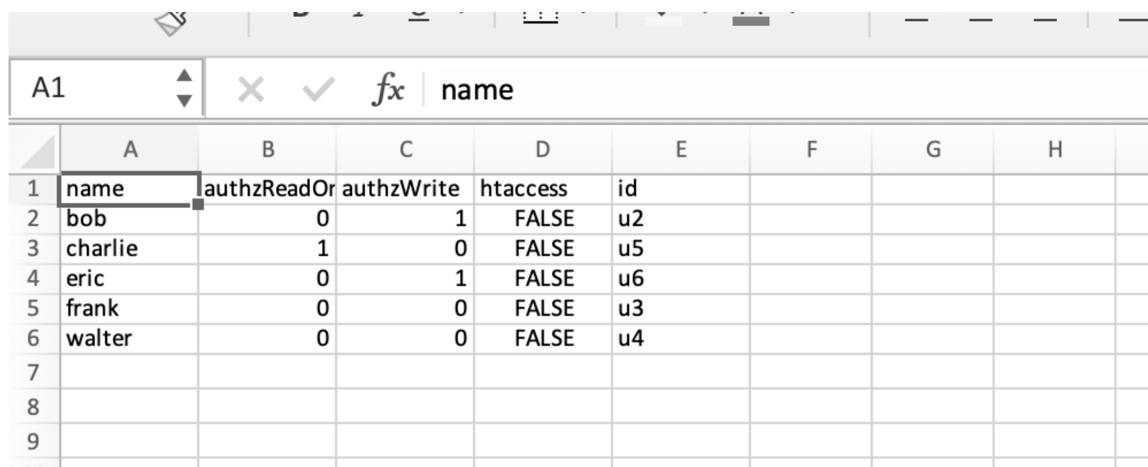
The model with highlighted changes is represented graphically in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Overview}}$ applied to Overview are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.4.11 Scenario: removed user, reload externally (Overview)

Removing of a user leads to the removal of all its access rights.

The *User* applies the desired changes to Overview by changing its external representation. The model with highlighted changes is represented with its concrete rendering:



A1 — name

	A	B	C	D	E
1	name	authzReadOr	authzWrite	htaccess	id
2	bob	0	1	FALSE	u2
3	charlie	1	0	FALSE	u5
4	eric	0	1	FALSE	u6
5	frank	0	0	FALSE	u3
6	walter	0	0	FALSE	u4

The model differences are represented in textual form:

```
1  u2.remove(rowNumber, 0, "3")
2  u2.add(rowNumber, 0, "2")
3  u5.remove(rowNumber, 0, "4")
4  u5.add(rowNumber, 0, "3")
5  u6.remove(rowNumber, 0, "5")
6  u6.add(rowNumber, 0, "4")
7  u3.remove(rowNumber, 0, "6")
8  u3.add(rowNumber, 0, "5")
9  u4.remove(rowNumber, 0, "7")
10 u4.add(rowNumber, 0, "6")
11 hp1.remove(users, 0, u1)
12 u1.remove(name, 0, "alice")
13 u1.remove(authzWrite, 0, "2.0")
14 u1.remove(htaccess, 0, "true")
15 u1.remove(rowNumber, 0, "2")
16 u1.remove(id, 0, "u1")
17 u1.changeNamespace(model-htpasswd ⇒ null)
18 u1.remove(authzReadOnly, 0, "0.0")
19 u1.changeType(htpasswd.User ⇒ null)
20 u1.deleteInstance()
```

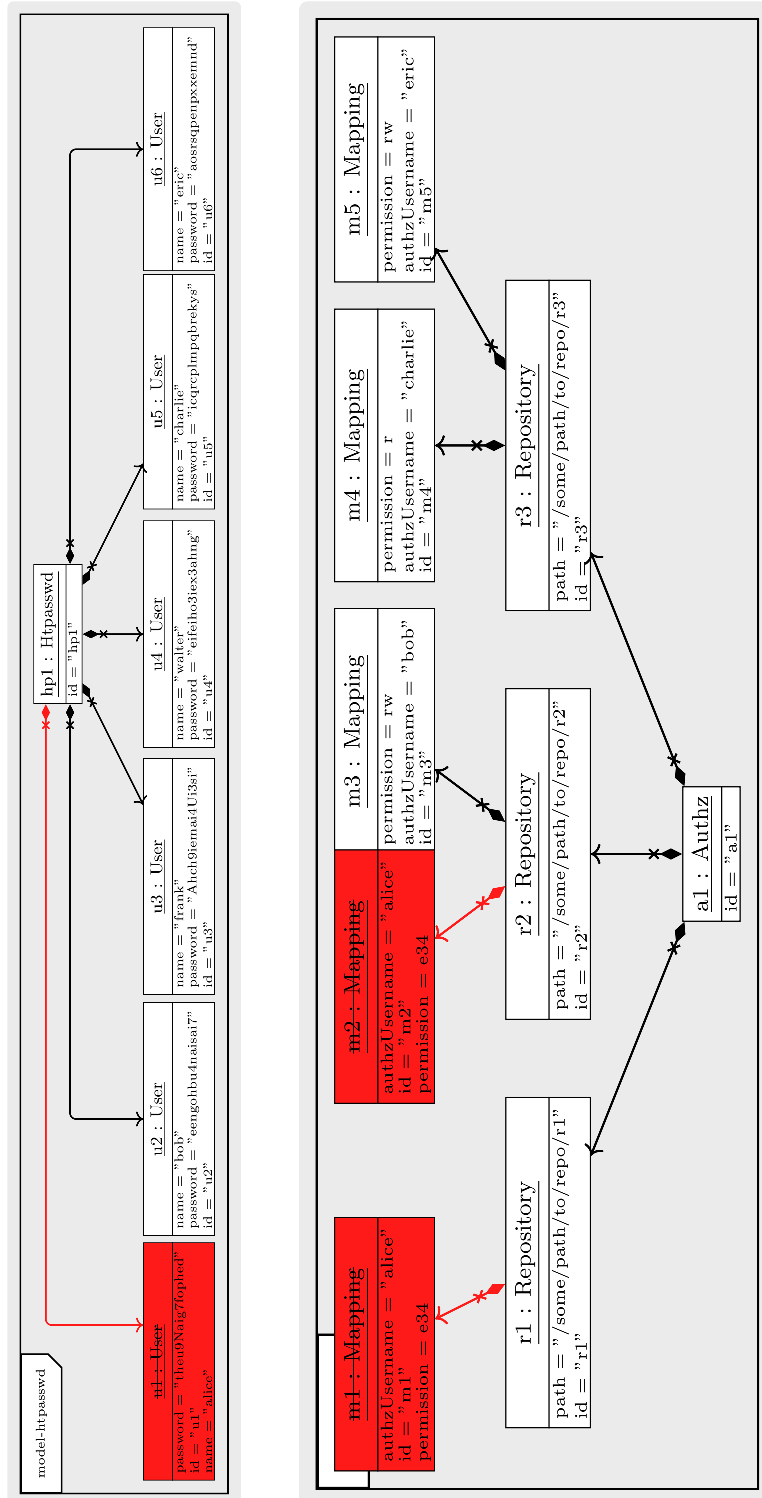
User $\Delta^{14}_{\text{Overview}}$

As result after completing the synchronization, the following changes are expected:

- In Htpasswd, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #hp1
2  alice : theu9Naig7fophed #u1
3 bob : eengohbu4naisai7 #u2
4 frank : Ahch9iemai4Ui3si #u3
5 walter : eifeiho3iex3ahng #u4
6 charlie : icqrcplmpqbrekys #u5
7 eric : aosrsqpenpxxemnd #u6
```

The model with highlighted changes is represented graphically (left):



- In Authz, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #a1
2 [/some/path/to/repo/r1] #r1
3 alice = rw #m1
4
5 [/some/path/to/repo/r2] #r2
6 alice = rw #m2
7 bob = rw #m3
```

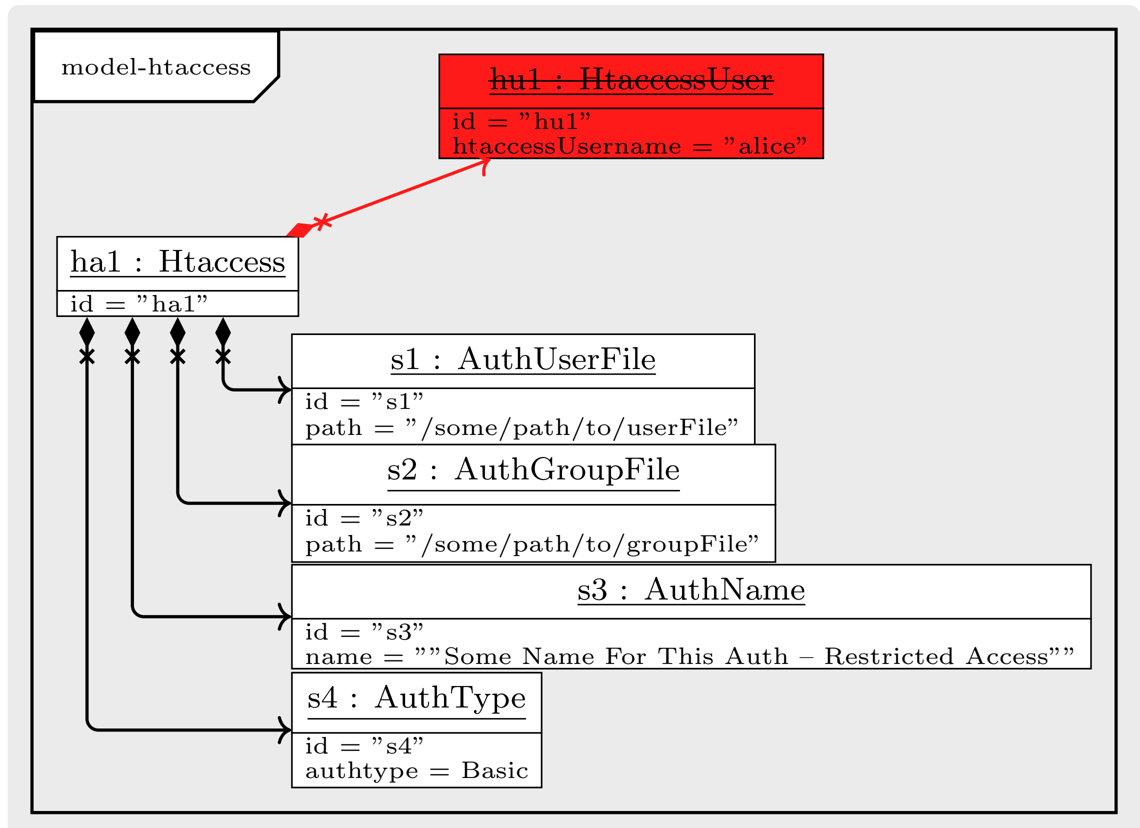
```
9  [/some/path/to/repo/r3] #r3
10 charlie = r #m4
11 eric = rw #m5
```

The model with highlighted changes is represented graphically in the figure before on the right.

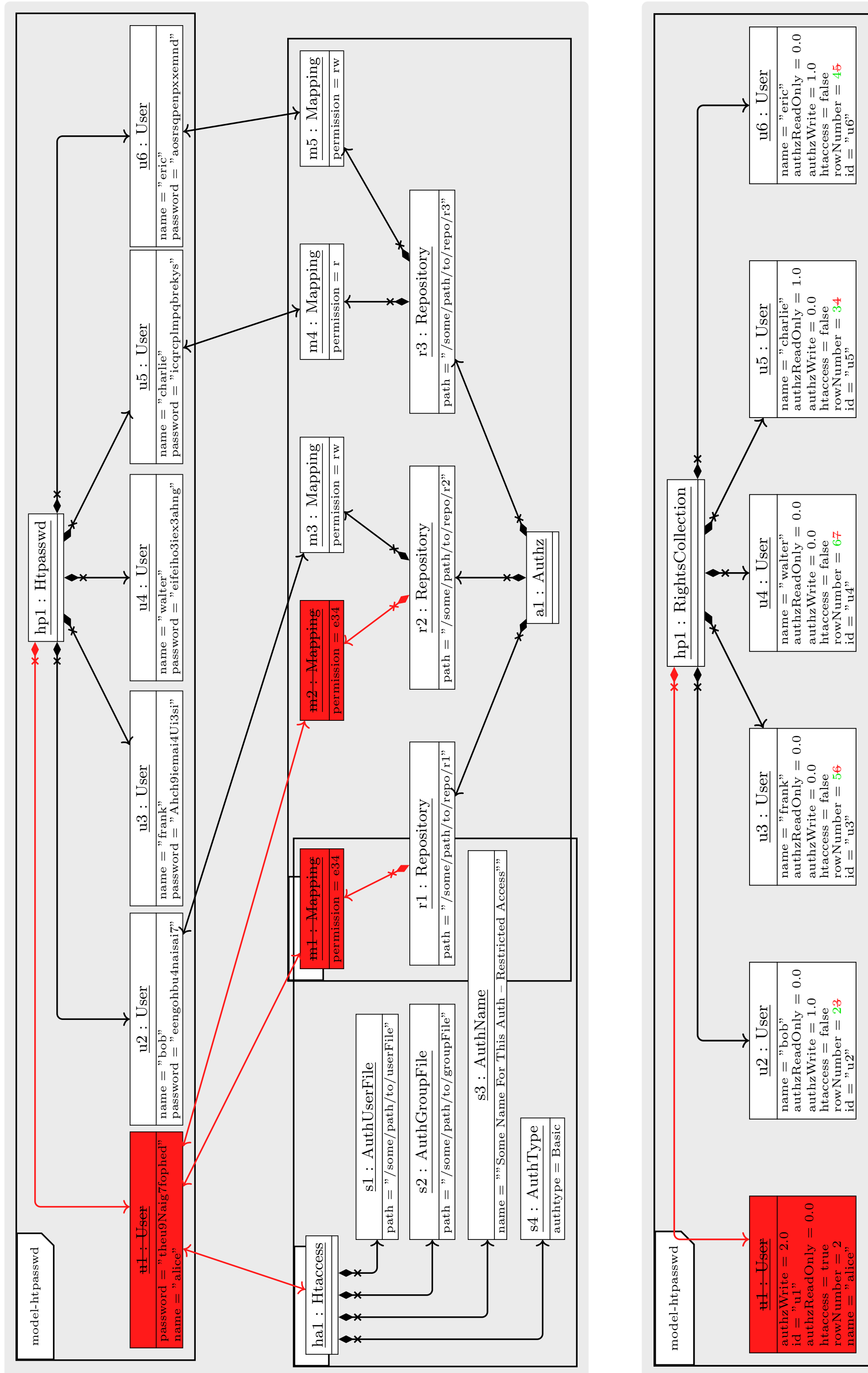
- In Htaccess, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

```
1 #ha1
2  AuthUserFile /some/path/to/userFile #s1
3 AuthGroupFile /some/path/to/groupFile #s2
4 AuthName "Some Name For This Auth -- Restricted Access" #s3
5 AuthType Basic #s4
6 require user alice #hu1
```

The model with highlighted changes is represented graphically:



- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically (left):



- In Overview, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Overview}}$). The model with highlighted changes is represented with its concrete rendering:

A1	fx name							
	A	B	C	D	E	F	G	H
1	name	authzReadOr	authzWrite	htaccess	id			
2	bob	0	1	FALSE	u2			
3	charlie	1	0	FALSE	u5			
4	eric	0	1	FALSE	u6			
5	frank	0	0	FALSE	u3			
6	walter	0	0	FALSE	u4			
7								
8								
9								

The model with highlighted changes is represented graphically in the figure before on the right.

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Overview}}$ applied to Overview are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

9.5 Summary: Contributions

After showing the configured operators and their purpose by ensuring the consistency in multiple test cases, the application of MoConseMI and its contributions are summarized now. All consistency goals and their rules of this application are listed below:

Collected Consistency Goals and their Rules

C 1 User with access rights given by Authz must be registered in Htpasswd.

- **C 1 a** If the user of an access right in Authz is not defined in Htpasswd, a corresponding user is added in Htpasswd with a random password.
- **C 1 b** If an access right is removed and its related user has no access rights anymore, that user is also removed.
- **C 1 c** If a user is removed in Htpasswd, all its access rights in Authz are removed.
- **C 1 d** If the user name of an access right is changed, the corresponding user is renamed accordingly, if the user has not more than this access right. Otherwise, a new user with the new user name is created.
- **C 1 e** If a user is renamed, all its rights in Authz are renamed accordingly.

C 2 User with access rights given by Htaccess must be registered in Htpasswd.

- **C 2 a** If the user of an access right in Htaccess is not defined in Htpasswd, this access right is removed in Htaccess.
- **C 2 b** If an access right is removed, only this access right is removed, while the related user remains.
- **C 2 c** If a user is removed in Htpasswd, all its access rights in Htaccess are removed.

C 2 d If the user name of an access right is changed, the corresponding user is renamed accordingly.

C 2 e If a user is renamed, all its rights in Htaccess are renamed accordingly.

The consistency goals C 1 and C 2 target two data sources and therefore represent *inter-model consistency* issues.

These consistency goals and their consistency rules are successfully tested by test cases documented in Section 9.4☞328. The mapping of test cases and their explicitly targeted consistency goals and rules are summarized in Table 9.1: The first column lists all consistency goals and their consistency rules. The second refers to the test cases which test explicitly the consistency goal or their consistency rule inside the same row in the first column.

Table 9.1: Mapping of Consistency Goals and their Consistency Rules tested in Sections for Access Data Management

Consistency	Test Cases
C 1	
C 1 a	9.4.4, 9.4.5
C 1 b	9.4.3, 9.4.6
C 1 c	
C 1 d	9.4.2
C 1 e	
C 2	
C 2 a	
C 2 b	
C 2 c	
C 2 d	9.4.7
C 2 e	

Table 9.1 shows, that all consistency goals and their consistency rules are successfully tested, only C 1 c, C 1 e, C 2 a, C 2 b, C 2 c and C 2 e have no test cases. Test cases for consistency rules test also their corresponding consistency goals.

The details of the test cases are summarized in Table 9.2☞370: Each scenario is represented by one row. The "Source" column indicates the view in the orchestration, at which the user applied the wanted changes. The "Kind" column indicates, if the user changed the external representation (E) or the internal EMF model (I). The "#" column contains the number of changes, made by the user. The following columns with the names of the data sources, SUM and new view contain the number of resulting changes, after finishing the synchronization. The last column "Testing" refers to the consistency goals and consistency rules, which are explicitly evaluated by the current test case.

The scenario 9.4.1 is described with more details above, but is not summarized in the table to keep it short. Scenarios with no reference to a subsection in the table are not documented in detail above.

Table 9.2: Summary of Test Cases for Access Data Management

	Description	Source	Kind	#	Htpasswd	Authz	Htaccess	SUM	Overview	Testing
	renamed User (Authz)	Authz	I	2	2	2	0	2	2	
9.4.2	renamed Mapping, reload externally (Authz)	Authz	E	2	7	2	0	10	22	C 1 d
	renamed User (Authz)	Authz	I	2	7	2	0	10	22	
9.4.3	removed Mapping, reload externally (Authz)	Authz	E	7	0	7	0	7	2	C 1 b
	delete Mapping (Authz)	Authz	I	7	7	7	0	13	18	
	delete Mapping (Authz)	Authz	I	7	0	7	0	7	2	
	added Mapping to Repository r1 (Authz)	Authz	I	6	7	7	0	13	12	C 1 a
9.4.5	added Mapping, reload externally (Authz)	Authz	E	6	0	7	0	7	2	C 1 a
	added Mapping to Repository r1 (Authz)	Authz	I	6	0	7	0	7	2	
	renamed Mapping, reload externally (Authz)	Authz	E	2	2	2	0	2	2	
9.4.6	removed Mapping, reload externally (Authz)	Authz	E	7	7	7	0	13	18	C 1 b
	added User (Htpasswd)	Htpasswd	I	6	7	0	0	6	12	
	renamed User, reload externally (Htpasswd)	Htpasswd	E	6	6	4	2	6	2	
	removed User, reload externally (Htpasswd)	Htpasswd	E	11	11	14	6	26	20	
	added User, reload externally (Htpasswd)	Htpasswd	E	10	11	0	0	10	12	
	renamed User (Htpasswd)	Htpasswd	I	2	2	4	2	2	2	
	removed User (Htpasswd)	Htpasswd	I	7	7	14	6	22	20	
	added HtaccessUser, reload externally (Htaccess)	Htaccess	E	5	0	0	6	2	2	
9.4.7	renamed HtaccessUser, reload externally (Htaccess)	Htaccess	E	2	2	4	2	2	2	C 2 d
	added HtaccessUser, reload externally (Htaccess)	Htaccess	E	5	0	0	0	0	0	
	added User (Htaccess)	Htaccess	I	5	0	0	0	0	0	

continued on next page

	Description	Source	Kind	#	Htpasswd	Authz	Htaccess	SUM	Overview	Testing
	remove HtaccessUser, reload externally (Htaccess)	Htaccess	E	6	0	0	6	2	2	
	renamed User (Htaccess)	Htaccess	I	2	2	4	2	2	2	
	removed User (Htaccess)	Htaccess	I	6	0	0	6	2	2	
	added User (Htaccess)	Htaccess	I	5	0	0	6	2	2	
9.4.8	change Htaccess right for Bob (Overview)	Overview	I	2	0	0	6	2	2	
9.4.9	change number of Authz rights for Bob (Overview)	Overview	I	2	0	0	0	0	0	
9.4.10	change the name of Bob to David (Overview)	Overview	I	2	2	2	0	2	6	
9.4.11	removed user, reload externally (Overview)	Overview	E	20	7	14	6	22	20	
	removed User (SUM)	SUM	I	12	7	14	6	22	20	
	added User, reload externally (SUM)	SUM	E	10	11	0	0	10	12	
	removed User, reload externally (SUM)	SUM	E	26	11	14	6	26	20	
	renamed User (SUM)	SUM	I	2	2	4	2	2	2	
	renamed User, reload externally (SUM)	SUM	E	6	6	4	2	6	2	
	added User (SUM)	SUM	I	6	7	0	0	6	12	

In the test case 9.4.9, the user changes are reverted due to the consistency goals and consistency rules or the definitions of new view(point)s, like read-only information. This shows an extreme strategy to ensure consistency, namely by reverting the wanted changes. Usually, the users change the data sources or the new views, since they are tailored for their needs: The test cases 9.4.8, 9.4.10 and 9.4.11 show one of the main benefits of the approach: The user can change a new view and the changes are propagated back into the initial data sources. The test cases 9.4.2, 9.4.3, 9.4.5, 9.4.6, 9.4.7 and 9.4.11 applied the wanted user changes to the external representation of the view, which correspond to the real usage by users in practise and represent acceptance tests for users. For the fast and easy definition of test cases, 9.4.8, 9.4.9 and 9.4.10 applied the wanted user changes to the internal EMF model, not to the external representation.

Benefits for distributed Access Management

Regarding the application domain of distributed access management, this application shows, that distributed access management can benefit from MoConseMI:

- Existing access systems and their rights can be kept, their data are reused and can be updated.
- The new view Overview presents an overview about all given access rights of all known users, together with central management of some access rights. This eases e. g. to delete a user completely with all given access rights, as demonstrated in Section 9.4.11 ☞363.

- In general, the use of the SUM containing all access rights of all users allows central changes in the data, e. g. to rename a user with the automated propagation into all views.

application outside SE

This example represents an application from applied computing science, but outside of software engineering, and demonstrates, that MoConseMI coming from and motivated by software engineering (see the ongoing example) is transferable to other disciplines.

support existing notations: Xtext for structured text

Additionally, no information encoded as technical models e. g. in EMF are used, but the concrete syntax of the domain is directly supported: Structured text in the formats of the existing data sources is supported by applying Xtext to transfer the notations of the users to EMF models and metamodels. The developed Xtext adapter ensures, that changes from the user within its known notation are propagated to MoConseMI, while changes inside other views are mirrored back into the notation, so that the user is supported within the known work environment. In general, this shows, that textual domain-specific languages can be supported by MoConseMI by adapters. The management of UUIDs inside the text-based notations show an alternative way to ensure stable UUIDs, while this approach is not the most user-friendly one, since UUIDs could be changed by users by accident.

high Test Coverage

The amount of test cases for the small number of concepts in the data sources ensures a *high test coverage*. While only some selected test cases are documented in details, lots of more test cases are available and successfully executed: Applying the same changes to the external Xtext-supported notation and to the internal EMF model shows, that the same results in form of changes for the other views are produced (except of different password hashes). This shows, that the transformation between text-based notations and EMF models by adapters works as expected. The additional test cases with changes in the SUM show, that not only dedicated views, but also the whole SUM is modifiable.

Contributions

Summarizing, the contributions of this application are the finding, that MoConseMI is applicable for managing distributed access rights, that additional adapters for additional technical spaces can be added, and that user changes can be applied to the SUM as well. In general, access rights management is another example which shows, that MoConseMI works in practice, i. e. it is successfully evaluated, that MoConseMI is able to integrate existing data sources into an explicit SU(M)M, to derive a new view(point) from the SU(M)M, and to automatically keep all these views consistent to each other.

Chapter 10

SEIS Viewpoints

This application sketches the description of architectures for sensor-based environmental information systems (SEIS). Since such architectures are described with six viewpoints (Kateule, 2019), these viewpoints need to be integrated in order to describe the architecture of a SEIS as a whole and in consistent form. Since the metamodels for these viewpoints, conforming models as well as the resulting orchestration are very large, in particular, too big to visualize them in reasonable way, only some parts of this application are sketched here. Parts of the conceptual ideas for the integration of these viewpoints are developed together with Ruthbetha Kateule during her PhD thesis (Kateule, 2019). Therefore, there are some overlaps regarding consistency goals, while the realization with MoConseMI is done by the author of this thesis.

10.1 Application Domain

Four of these six viewpoints base on the Siemens views (Hofmeister, Nord and Soni, 2000), i. e. the conceptual, module, execution and code viewpoints. The other two viewpoints, i. e. for data and topology, are added by Kateule (2019).

10.1.1 Conceptual Viewpoint

The conceptual viewpoint describes the main components and their connectors, extended with protocols for SEIS. The metamodel for the conceptual viewpoint is visualized in Figure 10.1☞374.

10.1.2 Module Viewpoint

The module viewpoint describes the modules, which are organized in subsystems and layers. Modules communicate directly with each other, if they are located inside the same layer. Otherwise, they have to communicate via interfaces. The metamodel for the module viewpoint is visualized in Figure 10.2☞375.

10.1.3 Execution Viewpoint

The execution viewpoint maps modules to runtime entities for their execution during runtime. Runtime entities are organized in software resources, which are organized in platform resources, which are organized in hardware resources. Additionally, communication paths are represented. The metamodel for the execution viewpoint is visualized in Figure 10.3☞376.

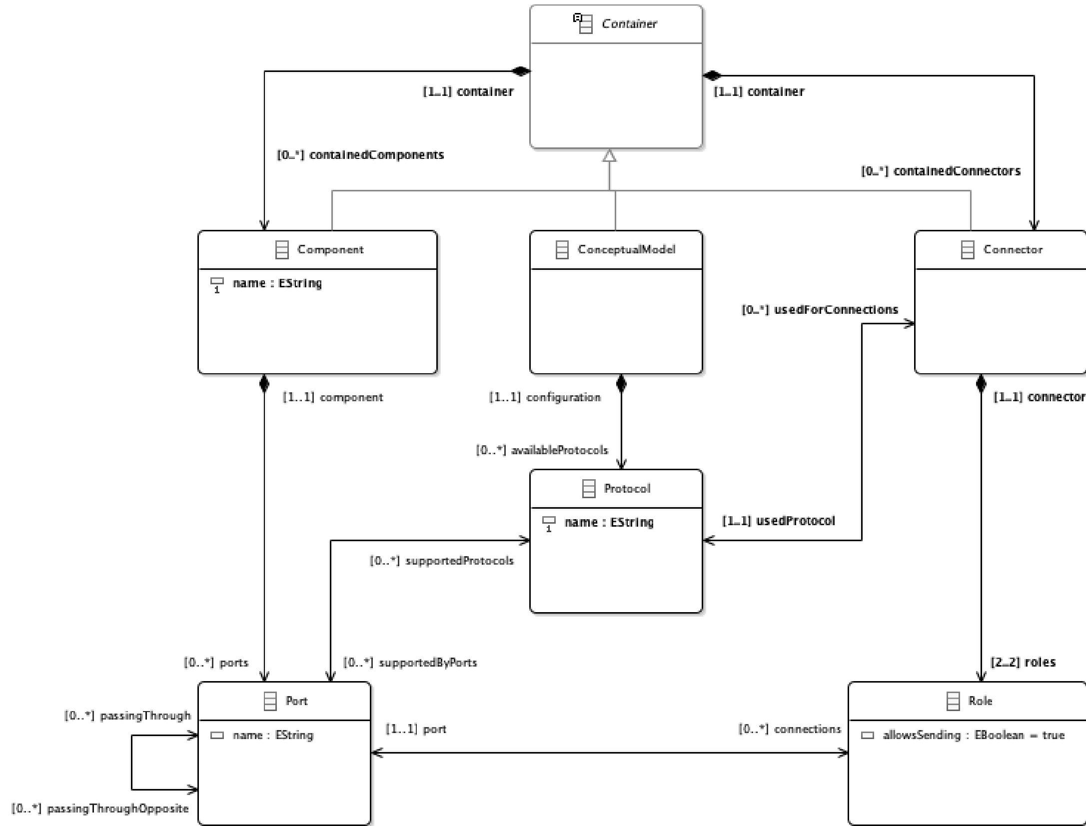


Figure 10.1: Metamodel for the Conceptual Viewpoint

10.1.4 Code Viewpoint

The code viewpoint organizes the artifacts of the implementation with source components, binaries, libraries, executables and configurations. The metamodel for the code viewpoint is visualized in Figure 10.4☞[377].

10.1.5 Topology Viewpoint

The topology viewpoint describes the topology of deployed nodes with their numbers of communications with other nodes, which can be constrained to reach certain topologies. The metamodel for the topology viewpoint is visualized in Figure 10.5☞[377].

10.1.6 Data Viewpoint

The data viewpoint describes concepts for data, which can be seen as subset of ECore (Section 2.5.3☞[87]). The metamodel for the data viewpoint is visualized in Figure 10.6☞[378].

10.2 Ensure Consistency between existing Data Sources

In order to integrate the viewpoints into the SUMM, possibilities for integration in terms of depending concepts must be identified, which are depicted in Figure 10.7☞[379] as overview. These consistency goals and consistency rules stem from the SEIS domain and are identified together with Ruthbetha Kateule (Kateule, 2019).

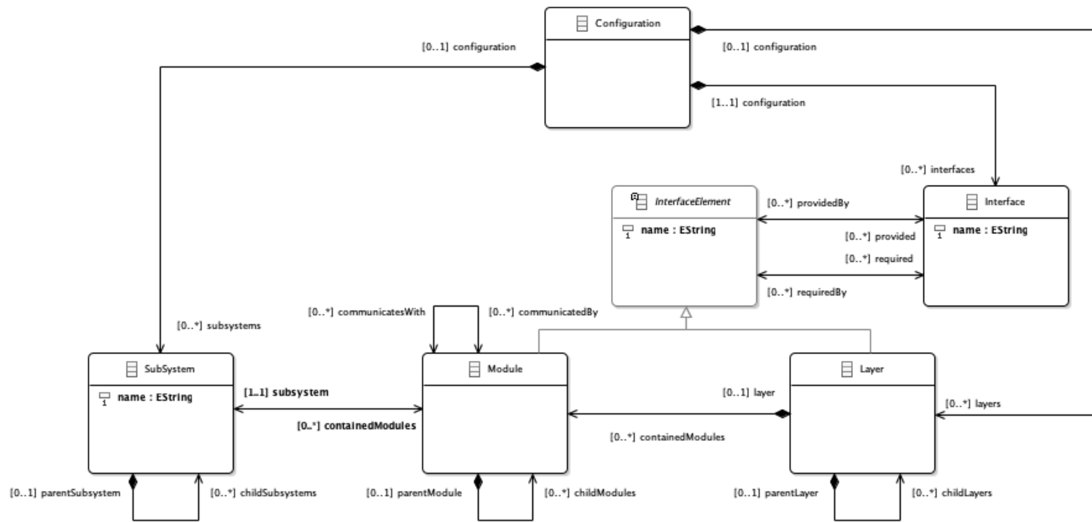


Figure 10.2: Metamodel for the Module Viewpoint

10.2.1 Conceptual — Module

The viewpoints for modules and conceptual components depend on each other regarding the following consistency rules:

Consistency Goal **C 1**	Module + Conceptual
Conceptual components for functionality are organized with modules for realization.	

This mapping must be done manually, since some components are realized by multiple modules or modules support parts of multiple components.

10.2.2 Module — Data

The viewpoints for modules and data depend on each other regarding the following consistency rules:

Consistency Goal **C 2**	Module + Data
Modules and Interfaces have to be linked with the packages which contain the data structures which are required for them.	

Modules and interfaces have to specify, which data are exchanged, which are designed with the data viewpoint.

Consistency Rule **C 2 a**	for C 2
Links between modules and interfaces and their used packages are added manually.	

This has to be done manually, since there is no automation for that.

Consistency Rule **C 2 b**	for C 2
If a module, an interface or a package is deleted, all its links must be deleted automatically, but not the other linked element.	

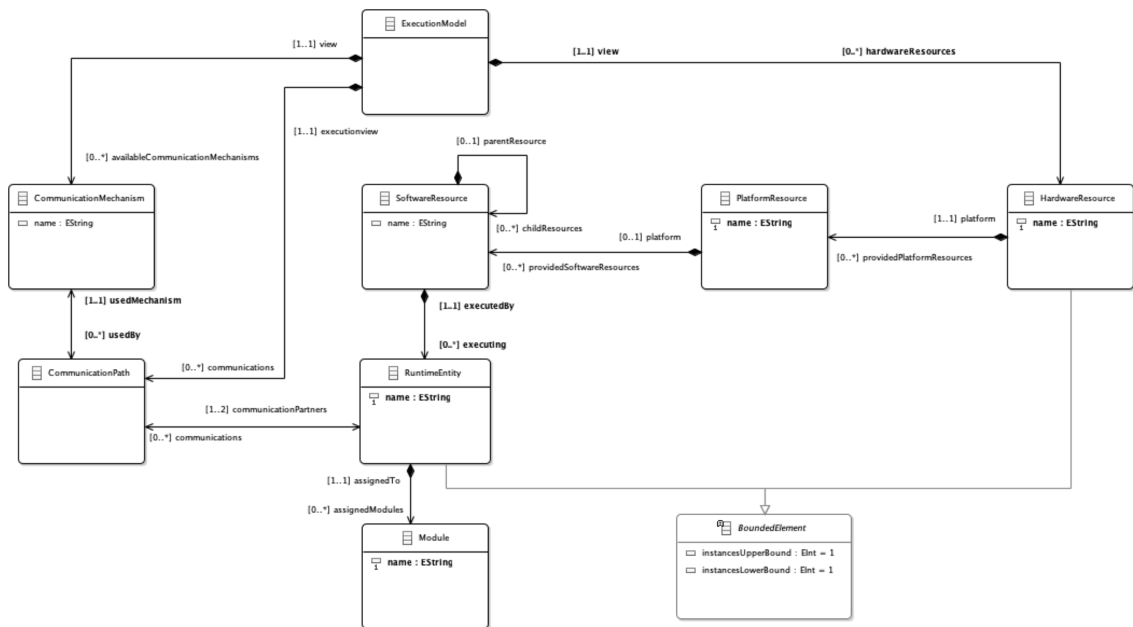
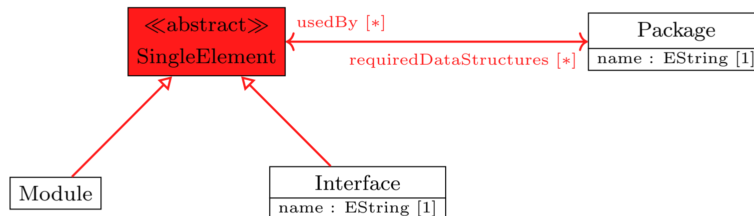


Figure 10.3: Metamodel for the Execution Viewpoint

After removing a module, an interface or a package, all direct usage links must be removed, too. The element at the other end of the link is kept. This is important, since these deletions can be done independently from the mappings, e.g. by removing the elements in their original data sources.

For the realization, a new association could be created between `Packages` and their related `Modules` and `Interfaces`, which are generalized before. This is depicted in red in the following graphic:



The following operators were configured to realize this:

1. →AddClass to the create the new class “SingleElement” with their (already existing) sub classes “Module” and “Interface”
2. →AddAssociation to create the new association in the metamodel, without adding any links in the model

10.2.3 Module — Code

The viewpoints for modules and code depend on each other regarding the following consistency rules:

Consistency Goal **C 3**	Module + Code
Modules and Interfaces are implemented in SourceComponents.	

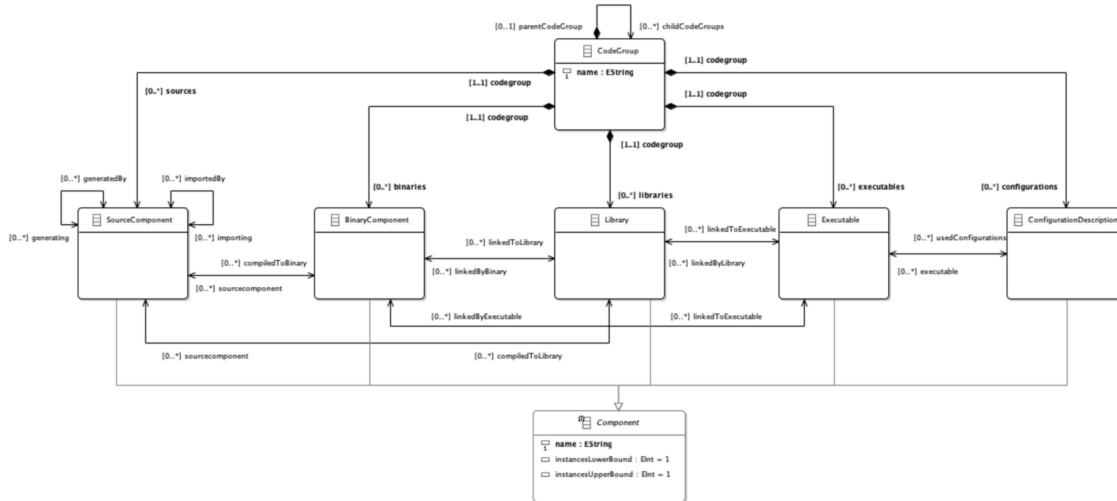


Figure 10.4: Metamodel for the Code Viewpoint

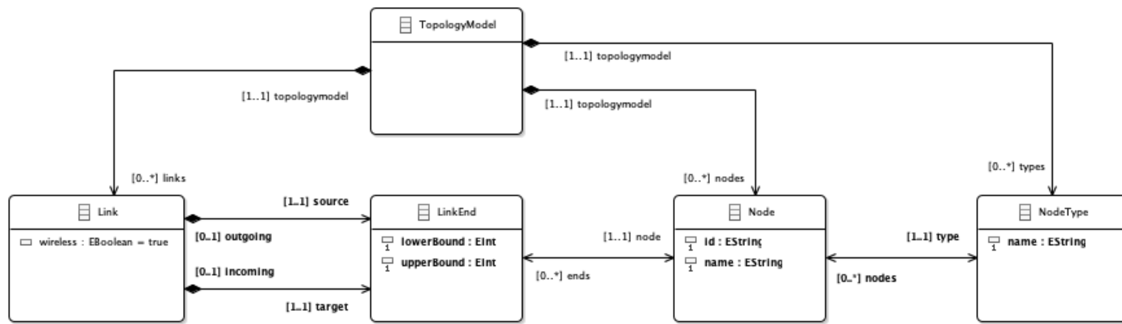


Figure 10.5: Metamodel for the Topology Viewpoint

For the execution at runtime, modules need a software-based realization in form of source code (source component).

Consistency Rule **C 3 a**	for C 3
Each Module and each Interface can have an unlimited number of implementations.	

Multiple implementations are required to realize modules and interfaces for different platforms. There might be modules or interfaces without any implementation.

Consistency Rule **C 3 b**	for C 3
One SourceComponent can be linked to an arbitrary number of Modules or Interfaces.	

Normally, each source component realizes only one module or interface. But there are some special situations, for example, if there are various modules within one sensor node, but they are all implemented within a single source component for that sensor node. There might be source components without any module or interface.

Consistency Rule **C 3 c**	for C 3
All mappings have to be specified manually.	

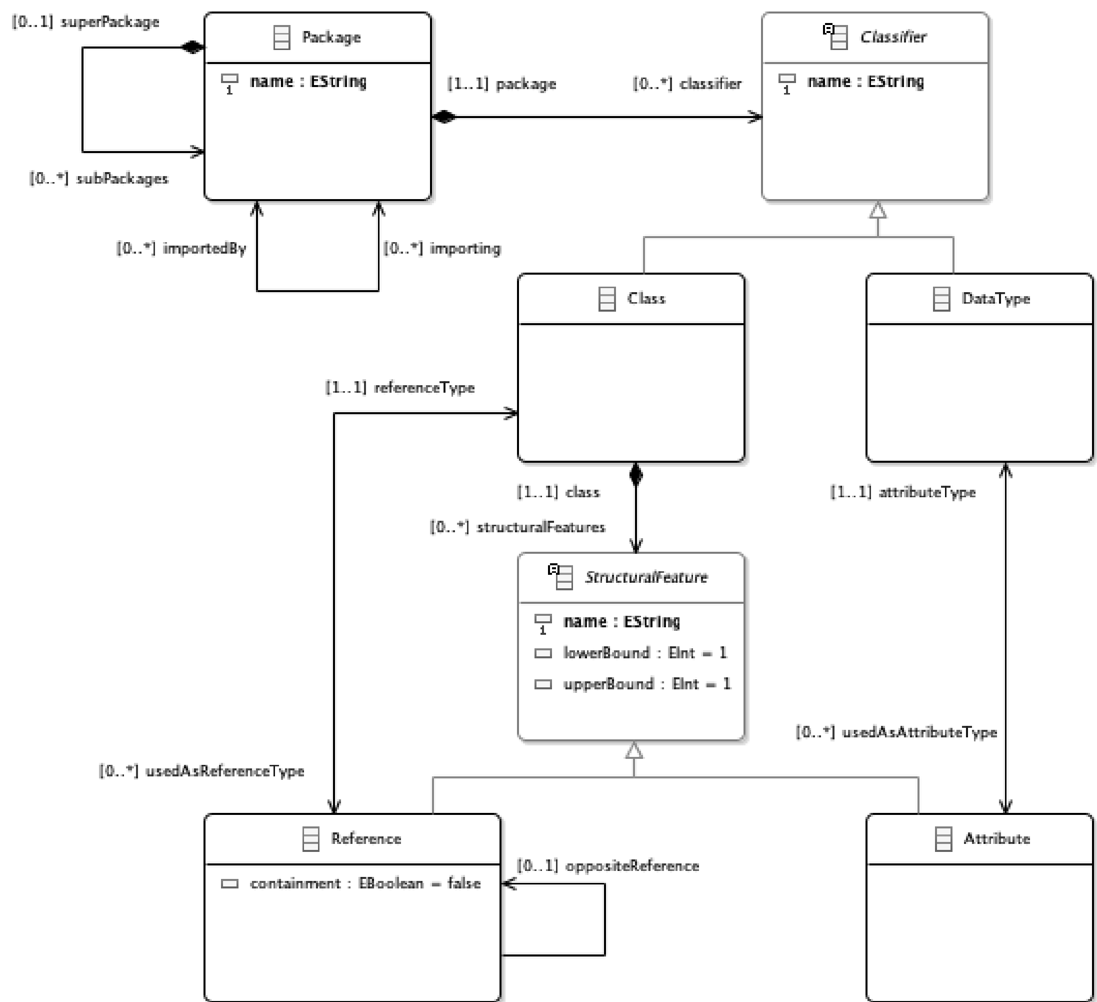


Figure 10.6: Metamodel for the Data Viewpoint

All mappings have to be specified manually, since there are no heuristics for automation. In particular, the many-to-many nature of this mapping prevents heuristics using name matching.

Consistency Goal **C 4**	Module + Code
The implementations of subsystems and layers from the Module view are contained in code groups from the Code view.	

While subsystems and layers are used to group modules, code groups are used to group source code. Due to the relationships between modules and source code (see above), the mechanisms for grouping modules and source code are related with each other as well.

Consistency Rule **C 4 a**	for C 4
Each subsystem and layer can have an unlimited number of implementations.	

Multiple implementations are required to realize subsystems and layers for different platforms. There might be subsystems or layers without any implementation.

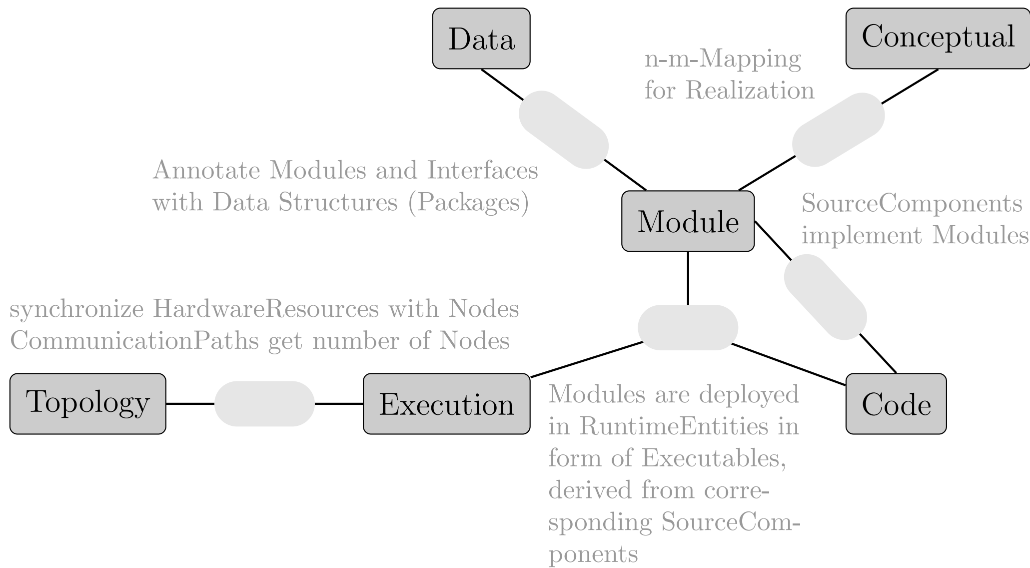


Figure 10.7: Overview about the Viewpoints and their Integrations

Consistency Rule **C 4 b**	for C 4
One code group realizes at maximum one subsystem or layer.	

Normally, each code group realizes one subsystem or layer. But code groups can be used also for hierarchical structuring, without an explicitly linked subsystem or layer. There might be source components without any module or interface.

Consistency Rule **C 4 c**	for C 4
Some mappings can be automated by comparing their names regarding equality.	

Therefore, corresponding heuristics can be configured in model decisions in order to detect and establish such mappings.

10.2.4 Module — Execution — Code

The viewpoints for modules, execution and code depend on each other regarding the following consistency rules:

Consistency Goal **C 5**	Module + Execution
The communication between runtime entities in the form of communication paths in the Execution view is derived and aggregated from the (internal and external) communication between modules in the Module view.	

Communicating modules needs communication paths, when they are deployed at different runtime entities.

Consistency Rule **C 5 a**	for C 5
If two modules communicate with each other (whether they use an interface or not, does not care), the corresponding runtime entities are connected with a communication path.	

Two different communicating modules inside the same runtime entity are not represented by a communication path. If a module communicates with itself ("loop"), there will be a "looping" communication path for the corresponding runtime entity.

Consistency Goal **C 6** Module + Execution + Code

Each module in the Module view can be deployed at an arbitrary number of runtime entities in the Execution view.

Since the deployment of modules on runtime entities is done in form of executables, which are derived from the source code for the modules, the Code view, the Execution view and the Module view need to fit together for the deployment of modules.

Consistency Rule **C 6 a** for C 6

If a module is deployed at a runtime entity, an arbitrary number of executables can be specified (which are runnable on the hardware resource and are derived from the implementation of the module).

An arbitrary number of executables is helpful, since there might by several executables which are required to execute one module. On the other side, the executable might not be available yet.

Consistency Rule **C 6 a** for C 6

The complete mapping for deployment has to be decided manually and cannot be automated.

Deployment of modules strongly depends on their functionalities as well as on non-functional needs like performance, which must be decided manually.

10.2.5 Execution — Topology

The viewpoints for execution and topology depend on each other regarding the following consistency rules:

Consistency Goal **C 7** Execution + Topology

Each hardware resource in the Execution view is represented by exactly one node in the Topology view (and vice versa).

For each hardware resource, it embedding into topologies should be available. Therefore, all hardware resources must be part of the Topology view.

Consistency Rule **C 7 a** for C 7

Newly created hardware resources are added to the other view, too.

Topology information must be available also for newly created hardware resource.

Consistency Rule **C 7 b** for C 7

Deleted hardware resources are deleted in the other view, too.

Topology information is not required for removed hardware resources anymore.

Consistency Goal C 8 Execution + Topology

Each communication path in the Execution view is represented by zero or one link in the Topology view (and vice versa).

Similar to the nodes (see above), also the links in terms of communication path are important for topology considerations. In the SEIS domain, there is at maximum one communication path between two hardware resources. There are no cases, where more than one communication path between the same two hardware resources exist.

Consistency Rule C 8 a for C 8

If the communication path represents external communication, it has a corresponding link in the Topology view.

This counts for communication paths which connect two (different) runtime entities which belong to different hardware resources. Communication paths which connect the same runtime entity (loop) are external, if the related hardware resource might exist more than once (upper bound of instances).

Consistency Rule C 8 b for C 8

If the communication path represents internal communication, it is not reflected in the Topology view.

This counts for communication paths which connect two (different) runtime entities which belong to the same hardware resource. Communication paths which connect the same runtime entity (loop) are internal, if the related hardware resource exists only once (upper bound of instances).

Consistency Rule C 8 c for C 8

New (external) communication paths in the Execution view result in new links in the Topology view.

Due to the strong relationships of Execution and Topology, links in the Topology view could be automatically derived and maintained.

Consistency Rule C 8 d for C 8

New links in the Topology view are mapped to newly created communication paths in the Execution view, if possible.

If the involved hardware resources have exactly one runtime entity, the corresponding communication path could be automatically created between these two runtime entities. If some of the involved hardware resources have zero or one runtime entity, missing runtime entities are created, before they are used (see above). If some of the involved hardware resources have more than one runtime entity, the new link in the Topology view will be removed, since information about the related runtime entities to use are missing and Topology view and Execution view have to be consistent to each other further on.

Consistency Rule **C 8 d**	for C 8
Deleted Links or CommunicationsPaths will be deleted in the other View, too.	

Due to the strong relationships of Execution and Topology, links in the Topology view could be automatically derived and maintained.

10.2.6 Realization Overview

A possible resulting technical integration with configured operators to fulfill the sketched consistency goals is depicted in Figure 10.8. It is not shown in detail for brevity. Its main purpose is to show, a lot of operators are required, since there are lots of viewpoints to integrate and lots of consistency goals to ensure. With other words, this long chain of operators show, that MoConseMI scales regarding the numbers of data sources and consistency goals by adding more operators.

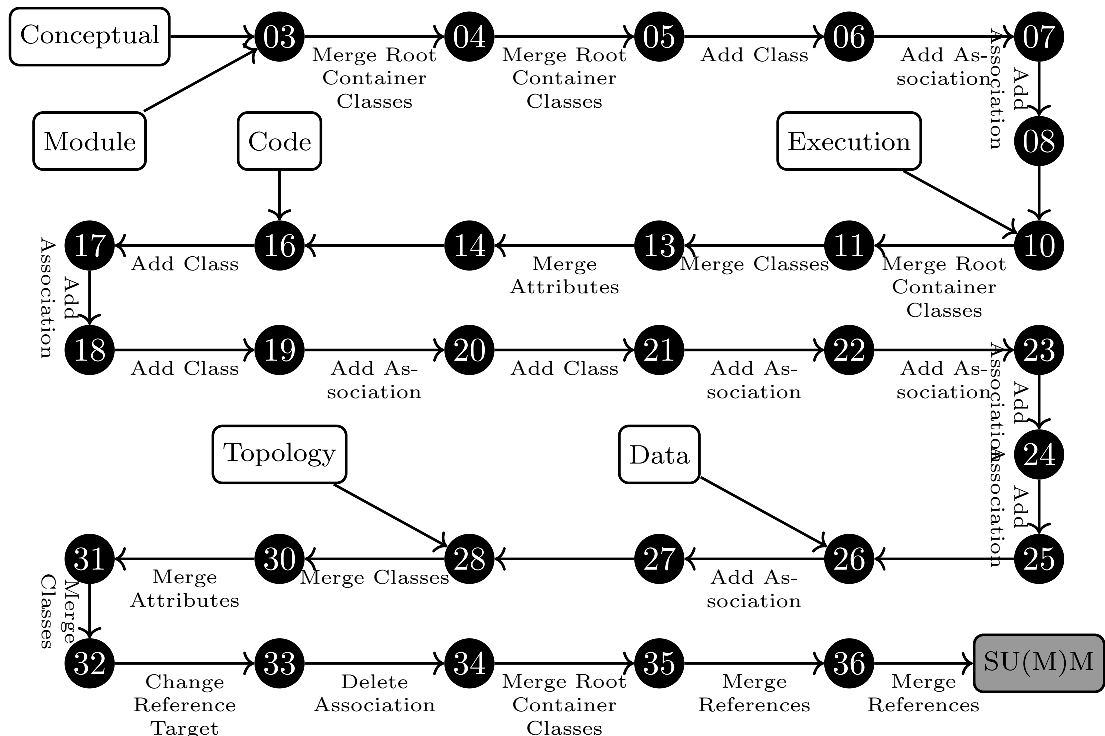


Figure 10.8: Operator Orchestration for the technical Integration

10.3 Define new Viewpoints

In order to provide additional presentations of the integrated architecture in order to support additional stakeholders and their additional concerns, new view(point)s can be defined on top of the SU(M)M. Some ideas for derived new view(point)s are presented in the following publication:

Related MoConseMI Publication

Johannes Meier, Ruthbetha Kateule and Andreas Winter (2020): *Operator-based viewpoint definition.* In: MODELSWARD 2020 - Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, pp. 401–408.

This publication is cited as Meier, Kateule and Winter (2020) in this thesis.

In the following sections, some ideas for new view(point)s are motivated and their realization is partially sketched. Figure 10.9 shows the simplified orchestration for two of the discussed new view(point)s.

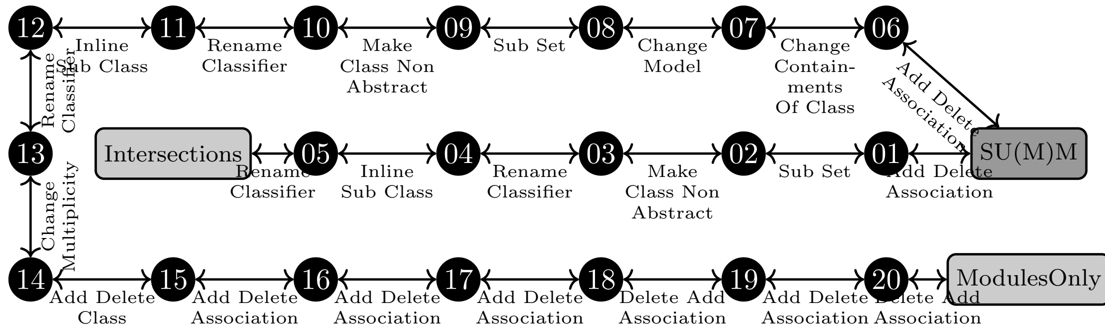


Figure 10.9: Simplified Orchestration for the Definition of two new Viewpoints

10.3.1 Conceptual-Module-Mappings

According to Consistency Goal C 1☞375, the mapping of components in the Conceptual view with the modules in the Module view need to be done manually. In order to support architects during this task, a "table" for the mappings should be introduced, similar to Kateule (2019, Table 7.3) and with e. g. EXCEL support. Without such a dedicated view, architects have to modify the SUM directly, since these mappings are neither in Conceptual nor in Module.

10.3.2 Intersections

The Intersections view(point) provides a subset of information of the Module view(point) in order to focus on intersections between subsystems and layers regarding shared modules only, as depicted in Figure 10.10. This view is beneficial for project managers, who have to organize developers according modules, layers and subsystems for large SEISs. The definition for this view(point) with a configured operator chain is depicted in simplified way in Figure 10.9.

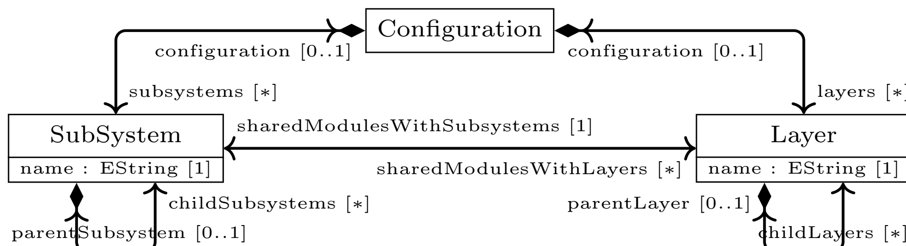


Figure 10.10: Metamodel for the new Viewpoint Intersections

10.3.3 ModulesOnly

The ModulesOnly view(point) provides a subset of information of the Module view(point) in order to focus on modules with their interfaces, as depicted in Figure 10.11. This view is beneficial for developers, who can focus on the details of particular modules and their communications with each other, while subsystems and layers might be more important for architects. The definition for this view(point) with a configured operator chain is depicted in simplified way in Figure 10.9☞383.

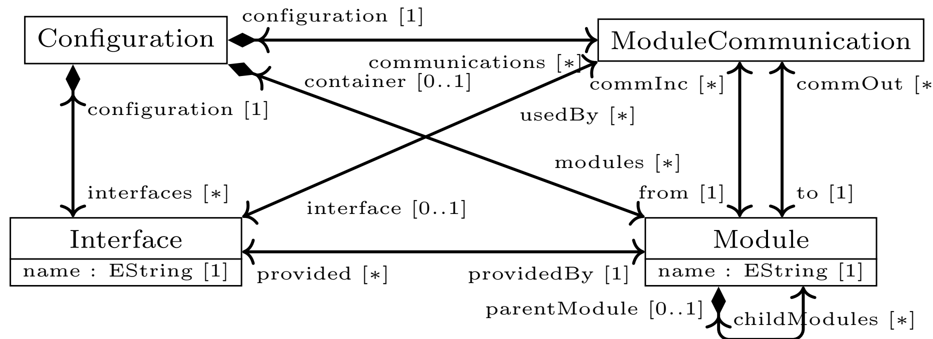


Figure 10.11: Metamodel for the new Viewpoint ModulesOnly

10.3.4 LayersOnly

The LayersOnly view(point) would be the counterpart to ModulesOnly and provides only layers and subsystems and their communications, but no modules as a subset of information of the Module view(point). For large SEISs, this "simplification" might be beneficial for architects to get an overview about the particular SEIS.

10.4 Validation Scenarios

Since models for views and the SUM are too large for small SEISs, this section does not show concrete test scenarios for consistency. Nevertheless, for all presented consistency goals and the details of their consistency rules, corresponding change scenarios can be defined and evaluated. As a simple example, Consistency Goal C 7☞380 can be tested by adding a new and removed an existing hardware resource in the Execution view in order to check, whether the Topology view is updated accordingly. Additionally, test cases for more special situations like looping communications sould be evaluated.

10.5 Summary: Contributions

This application of MoConseMI contributes an example from related work, where metamodels and consistency goals are not artificially developed, but are mainly predefined. The integration of *six data sources* is sketched, while additional new view(point)s are derived from the SU(M)M. Furthermore, this application of MoConseMI contributes an explicit integration of four viewpoints for architectures, which is indicated by Hofmeister, Nord and Soni (2000), but neither detailed nor technically integrated. As another benefit, MoConseMI can ensure consistency between views, which complements the integration. Additionally, two more viewpoints of Kateule (2019) are integrated as well. In general, this integration of viewpoints for architectures fit to the definitions of the ISO Standard for Architecture Description 42010:2011 (IEEE, 2011), since there are multiple architectural

viewpoints which describe *one* architecture, which is made explicit by MoConseMI in form of the SUMM. Interesting in detail is Consistency Goal C 6☞ 380, since *three* data sources are involved within the same consistency goal, pointing to n-ary consistency goals. Summarizing, this section contributes a large application of MoConseMI in terms of numbers of viewpoints and consistency goals in practical setting.

Chapter 11

Knowledge Management

Knowledge Management ...

This application applies MoConseMI to *knowledge management*. Objective of knowledge management is to use existing information which is spread over various files, data bases or other data sources in order to identify relationships between these volumes of information. These relationships can be established and provided as additional information. Combining this information along relationships enables to integrate data sources regarding their semantics.

... as Application for MoConseMI

Since knowledge management deals with heterogeneous information which are spread over multiple data sources which should be integrated regarding semantics in order to provide newly derived information, MoConseMI is applied to knowledge management in order to demonstrate, that MoConseMI is able to realize knowledge management. Main ideas of this application are, that existing information are reused as data sources in MoConseMI, that they are integrated regarding their semantics into a SU(M)M, and that the SU(M)M is exploited to provide newly derived information in new view(point)s.

Feasibility Study

Wegner (2021) applied MoConseMI for two simple and artificial examples in knowledge management. The first example describes existing knowledge in research projects with scientific employees, developed software components as research prototypes and supervised theses. Since scientific employees develop components and supervise theses, while some components are also developed in some theses, all this information has semantic relationships, which are not explicitly usable. Instead, the information is spread over multiple CSV and Excel files and indirectly refer to each other by names or IDs. This example served as feasibility study and showed, that spread knowledge can be integrated with MoConseMI in order to provide relationships between knowledge in explicit way in form of new view(point)s.

Knowledge Management in a Software Development Company

Therefore, this section presents a more elaborated example for knowledge management, which is realized with MoConseMI. It bases on ideas from the second example of Wegner (2021) and provides additional integration scenarios: This application example covers the knowledge management in a fictitious software development company with information about the employees (with roles and salaries), development tasks (with planned costs), used materials (like servers or offices with costs) and performed work. While the performed work is done by employees for a task using some materials, these implicit relationships are not directly traceable, since the information are spread over four CSV files which refer to each other with some IDs. This makes the calculation and the management of the real costs hard for managers.

Objective of this application is to show the ability of MoConseMI to integrate existing knowledge (Section 11.2 ☞404) with an explicit SU(M)M as result (Section 11.1.5 ☞396). As preparation, Section 11.1 ☞388 introduces the data sources in more detail and formulates the information to integrate with consistency goals in Section 11.1.7 ☞401. Basing on the SU(M)M, this application shows, that additional knowledge can be derived from this in-

tegration and provided in a new view(point) (Section 11.3 ☞425). Section 11.4 ☞434 shows some validation scenarios, which demonstrate, how the knowledge can be manually updated, while the integration of knowledge is automatically updated as well. More contributions of this application are summarized in Section 11.5 ☞452.

technical Hints

Note, that the presented application does not use real data: All data are artificial and developed for this fictitious software development company. Most data are reused from Wegner (2021). In some visualizations of models, root objects are hidden for clarity, since they are required only for the containment tree of ECore (Section 2.5.3 ☞87) and do not carry information which is required for knowledge management here. Additionally, the role names in the UML diagrams for the models are hidden for an improved readability.

11.1 Application Domain

The domain consists of the data sources Work in Section 11.1.1, Employees in Section 11.1.2 ☞391, Tasks in Section 11.1.3 ☞393 and Materials in Section 11.1.4 ☞395 as input. All relevant information for the domain are contained in the SUM (Section 11.1.5 ☞396). Parts of the SU(M)M are represented in the new view(point) Costs (Section 11.1.6 ☞398).

11.1.1 DataSource Work

This data source stores all knowledge about the done work of employees for tasks using materials. Each work entry documents the work of one employee for one day and for one task. The company manages its work in a CSV file. The initial concrete syntax before the initialization of Work is shown in the format CSV in Table 11.1.

Table 11.1: The initial input of Work in CSV format

#	ID	MID	APID	Day	Materials
1	1	PN11	AP1	4012021	M02
2	2	PN12	AP1	4012021	M02
3	3	PN12	AP1	5012021	M02
4	4	PN13	AP2	5012021	M01
5	5	PN13	AP2	6012021	M01
6	6	PN13	AP2	10012021	M01, M03
7	7	PN14	AP2	11012021	M01, M03
8	8	PN11	AP3	7012021	
9	9	PN14	AP3	6012021	M03

The first column (“#”) depicts the row numbers in the CSV file. The second column (“ID”) contains IDs for each work which are unique for all work entries of the company. The third column (“MID”) indicates the employee by its personnal number who did the particular work. The fourth column (“APID”) indicates the task by its ID of the particular work. The fifth column (“Day”) stores the day of the particular work. The sixth column (“Materials”) stores all used materials for the particular task, indicated by their IDs and concatenated by commas. Each work entry can use an arbitrary number of different materials. The metamodel of Work is shown in Figure 11.1 ☞389.

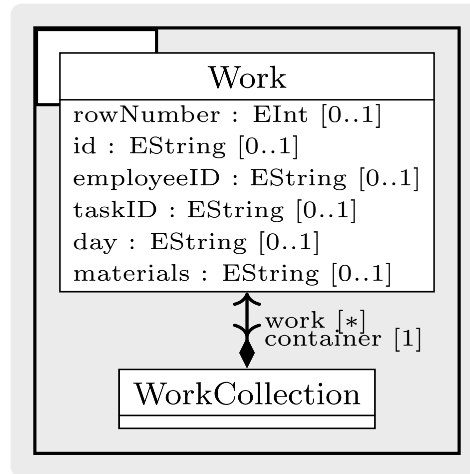


Figure 11.1: Metamodel of Work

The metamodel represents work done in the company as instances of `Work`, which are collected in a `WorkCollection`. The names in the first row of the CSV file are used for the attributes in `Work`, according to the CSV adapter as discussed in Section 8.4.4☞275. All attributes have `EString` as data type, since CSV files store only text. The initial input model of Work is shown in Figure 11.2☞390.

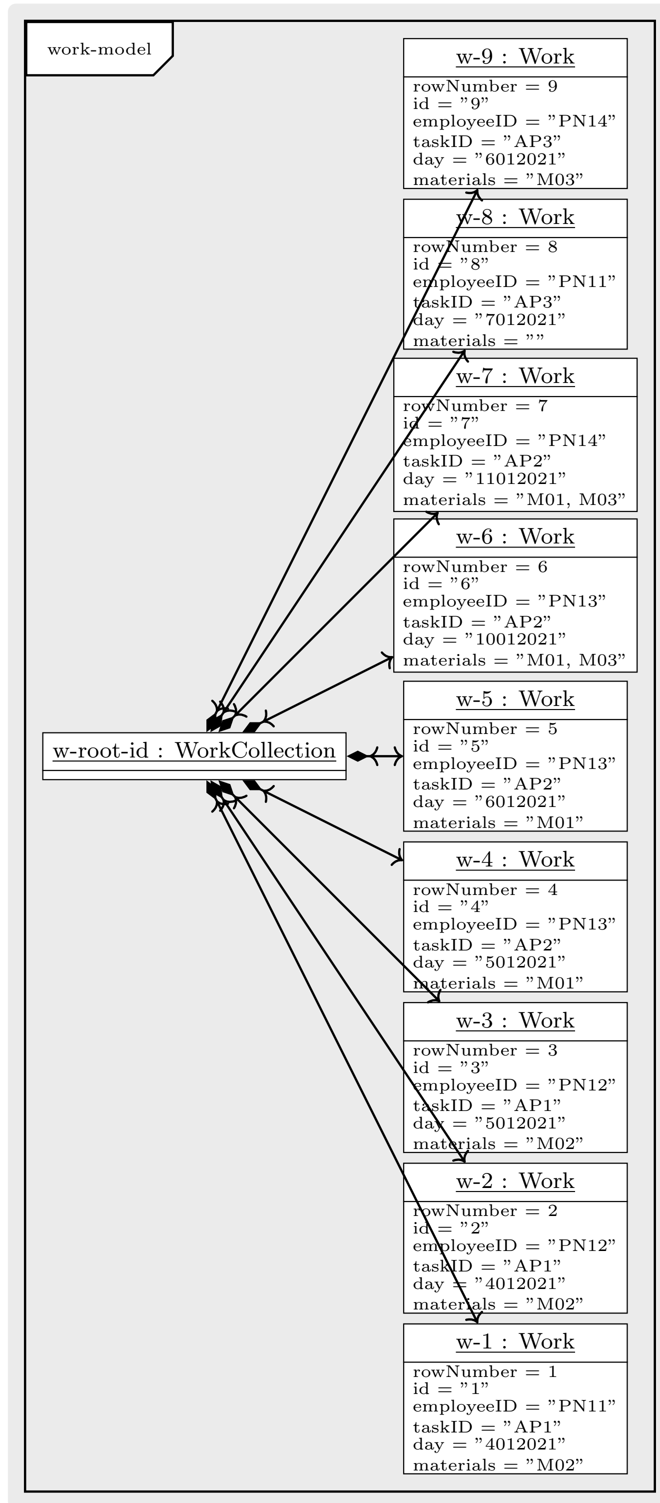


Figure 11.2: The initial input model of Work

According to the metamodel and the transformations of the CSV adapter, the model contains one object with `Work` as type for each row with content in the CSV file. The attribute slots are assigned with the content of the corresponding cell in the CSV file. All `Work` objects are stored in an instance of `WorkCollection`.

11.1.2 DataSource Employees

This data source stores all knowledge about employees of the software development company. The company manages its employees in a CSV file. The initial concrete syntax before the initialization of Employees is shown in the format CSV in Table 11.2.

Table 11.2: The initial input of Employees in CSV format

#	Personnel Number	Lastname	Firstname	Role	Salary
1	PN11	Patterson	Lori	Developer	60000
2	PN12	Cortez	Paulette	Developer Sr	75000
3	PN13	Glass	Venita	Developer Sr	73000
4	PN14	Holland	Scarlett	Developer	76000

The first column ("#") depicts the row numbers in the CSV file. The second column ("Personnal Number") contains IDs for each employee which are unique for all employees of the company. The third and fourth columns contain the lastnames and firstnames of the employees. The fifth column ("Role") indicates the position of employees in the company. The sixth column ("Salary") stores the annual salary (in Euro). The metamodel of Employees is shown in Figure 11.3.

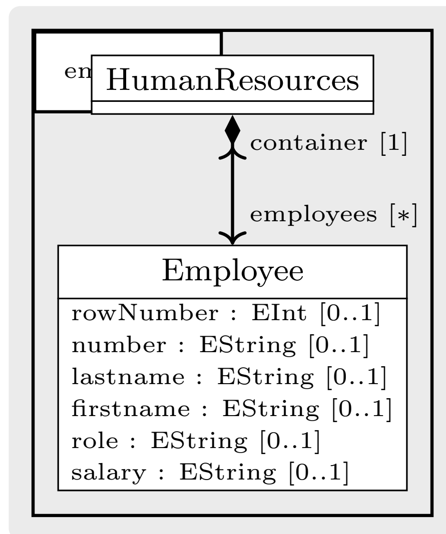


Figure 11.3: Metamodel of Employees

The metamodel represents employees of the company as instances of `Employee`, which are collected in `HumanResources`. The names in the first row of the CSV file are used for the attributes in `Employee`, according to the CSV adapter as discussed in Section 8.4.4 ☞275. All attributes have `EString` as data type, since CSV files store only text. The initial input model of Employees is shown in Figure 11.4 ☞392.

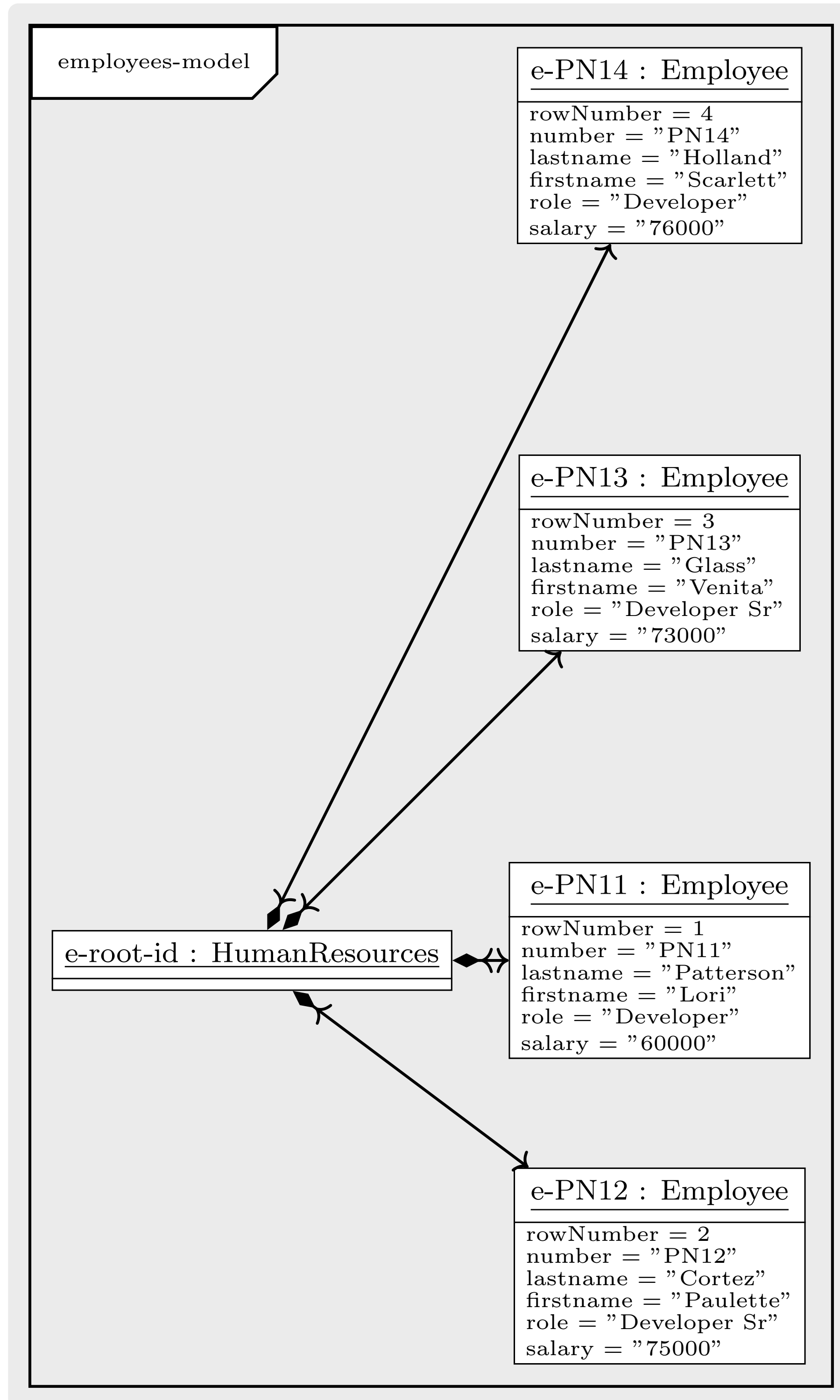


Figure 11.4: The initial input model of Employees

According to the metamodel and the transformations of the CSV adapter, the model contains one object with `Employee` as type for each row with content in the CSV file. The attribute slots are assigned with the content of the corresponding cell in the CSV file. All `Employee` objects are stored in an instance of `HumanResources`.

11.1.3 DataSource Tasks

This data source stores all knowledge about tasks in the software development company. The company manages its tasks in a CSV file. The initial concrete syntax before the initialization of Tasks is shown in the format CSV in Table 11.3.

Table 11.3: The initial input of Tasks in CSV format

#	ID	Description	Plan Duration	Plan People Costs	Plan Material Costs
1	AP1	Architecture	3	800	700
2	AP2	Database	6	1600	1400
3	AP3	Eventbus	5	1400	1100
4	AP4	Network	6	1600	1400

The first column ("#") depicts the row numbers in the CSV file. The second column ("ID") contains IDs for each task which are unique for all tasks of the company. The third column ("Description") contains a short description for the tasks. The fourth column ("Plan Duration") stores the planned duration in people-days for each task. The fifth and sixth columns store the planned costs (in Euro) for people, i. e. employees who worked for the particular task, and for materials, which are used by employees during their work for the particular task. The metamodel of Tasks is shown in Figure 11.5.

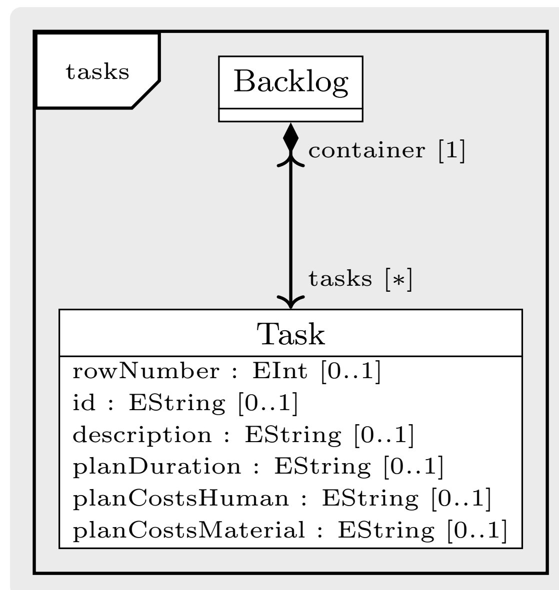


Figure 11.5: Metamodel of Tasks

The metamodel represents tasks as instances of `Task`, which are collected in a `Backlog`. The names in the first row of the CSV file are used for the attributes in `Task`, according to the CSV adapter as discussed in Section 8.4.4☞275. All attributes have `EString` as data type, since CSV files store only text. The initial input model of Tasks is shown in Figure 11.6☞394.

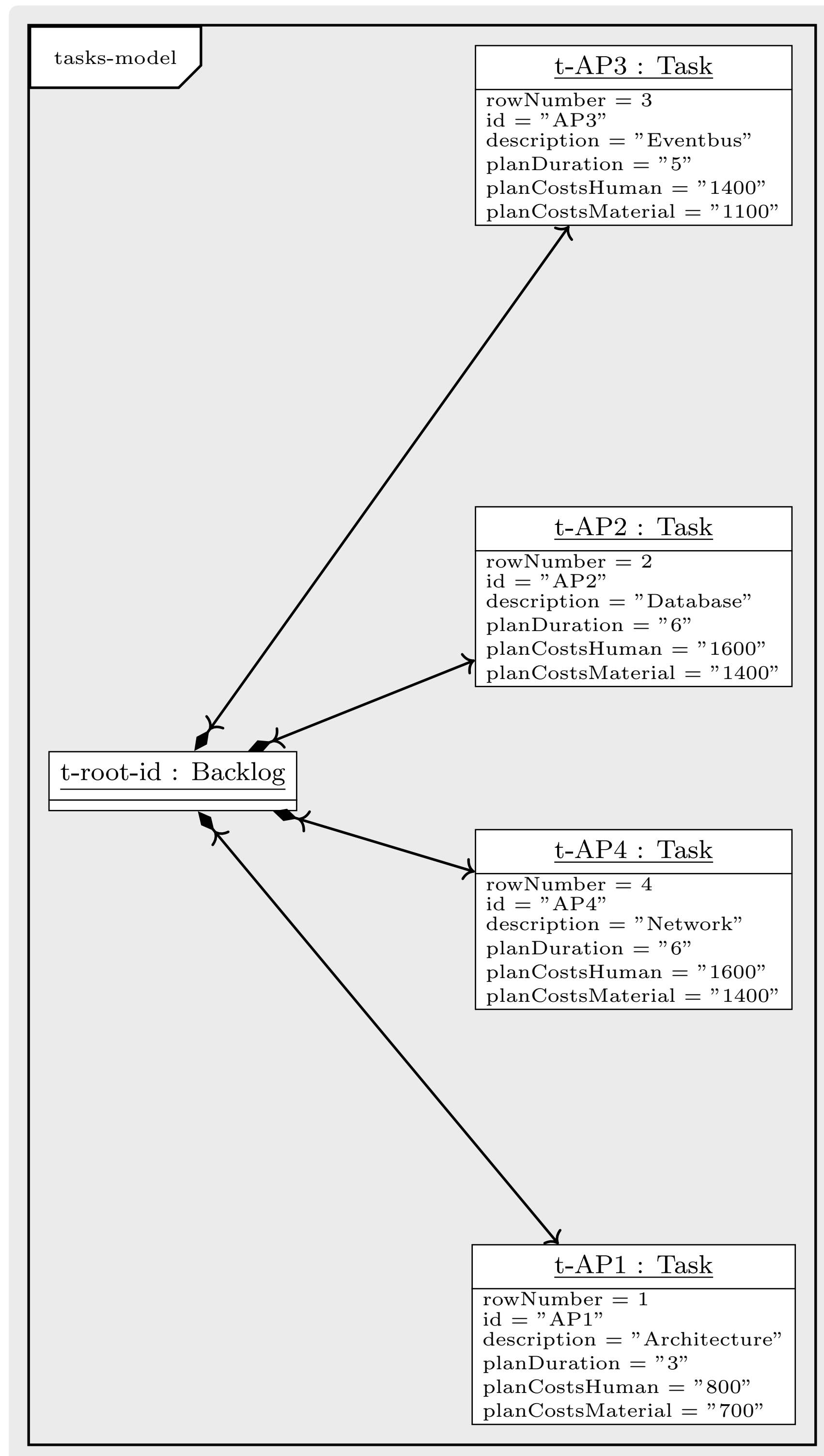


Figure 11.6: The initial input model of Tasks

According to the metamodel and the transformations of the CSV adapter, the model contains one object with `Task` as type for each row with content in the CSV file. The attribute slots are assigned with the content of the corresponding cell in the CSV file. All `Task` objects are stored in an instance of `Backlog`.

11.1.4 DataSource Materials

This data source stores all knowledge about materials which are used in the software development company. The company manages its available materials in a CSV file. The initial concrete syntax before the initialization of Materials is shown in the format CSV in Table 11.4.

Table 11.4: The initial input of Materials in CSV format

#	ID	Label	Costs per Day
1	M01	Server	20
2	M02	IDE	35
3	M03	Office	100

The first column ("#") depicts the row numbers in the CSV file. The second column ("ID") contains IDs for each material which are unique for all materials of the company. The third column ("Label") contains a short name for the material. The fourth column ("Costs per Day") stores the daily costs (in Euro, no decimals), which arise for each employee and for each day, when the materials are used. Currently, employees can use servers, IDEs and offices in the company. The metamodel of Materials is shown in Figure 11.7.

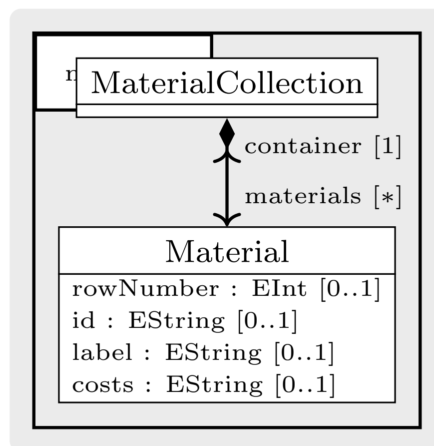


Figure 11.7: Metamodel of Materials

The metamodel represents materials as instances of `Material`, which are collected in a `MaterialCollection`. The names in the first row of the CSV file are used for the attributes in `Material`, according to the CSV adapter as discussed in Section 8.4.4 ☞275. All attributes have `EString` as data type, since CSV files store only text. The initial input model of Materials is shown in Figure 11.8 ☞396.

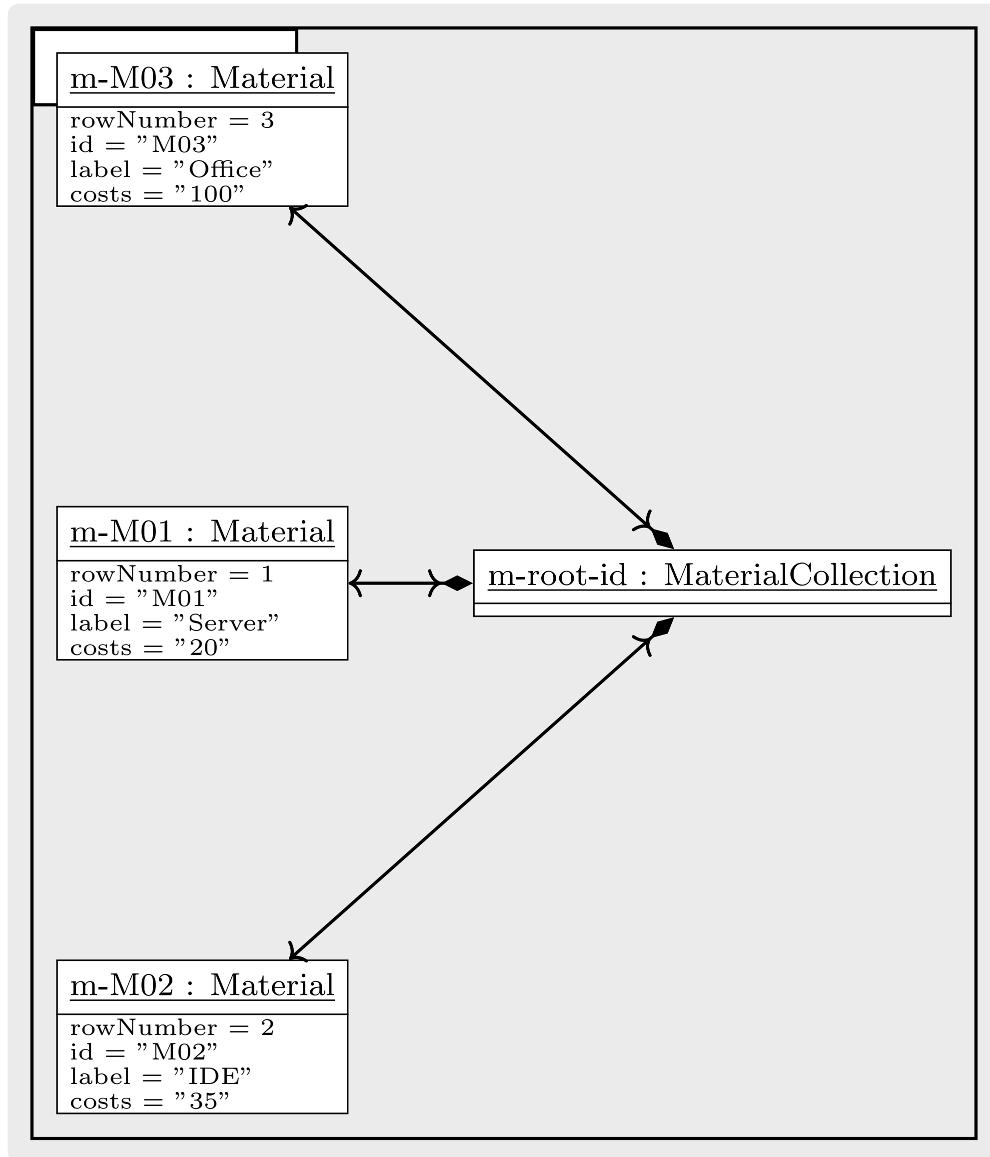


Figure 11.8: The initial input model of Materials

According to the metamodel and the transformations of the CSV adapter, the model contains one object with `Material` as type for each row with content in the CSV file. The attribute slots are assigned with the content of the corresponding cell in the CSV file. All `Material` objects are stored in an instance of `MaterialCollection`.

11.1.5 SU(M)M

Initially, there is no SU(M)M, but it is created for the first time during the initialization by executing the configured operators with the introduced data sources as starting point. The configurations of the operators control the final structure of the SU(M)M, which are documented in detail in Section 11.2 ☞404. This section serves as look-ahead and is useful for understanding the new view(point). The metamodel of SUMM is shown in Figure 11.9 ☞397.

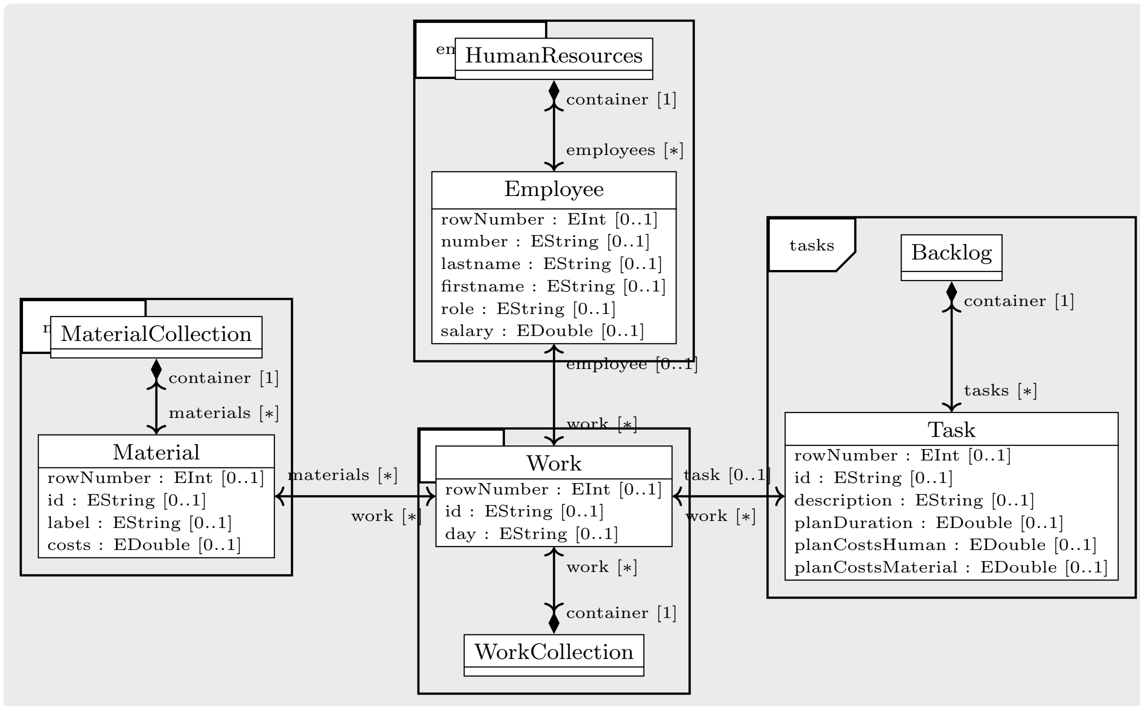


Figure 11.9: Metamodel of SUMM

The four namespaces `employees`, `tasks`, `work` and `materials` of the four data sources are still existing in the SUMM and indicate the origins of the contained classes. Additionally, some integrations are visible in form of associations crossing the bounds of these namespaces, which enable to make the implicit relationships via IDs explicit with links. The details of this integration are explained later. The final model after the initialization of SUMM is shown in Figure 11.10☞ 398.

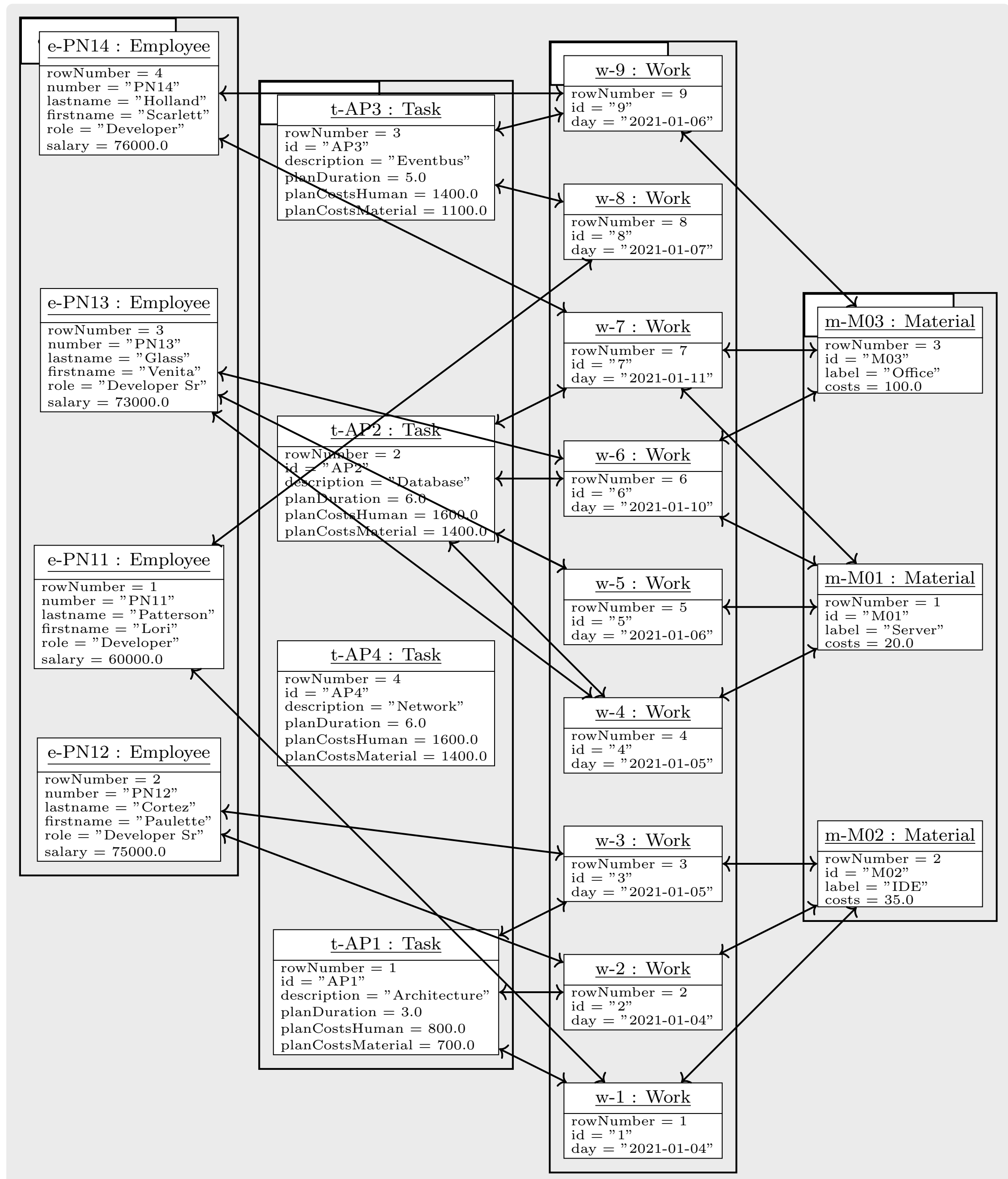


Figure 11.10: The final model after the initialization of SUMM

Corresponding to the SUMM, also the SUM still contains the namespaces of the four data sources on model level. There are lots of links connecting `Employees`, `Tasks` and `Materials` with `Work` entries, which replace the indirect links via IDs.

11.1.6 New ViewPoint Costs

In order to exploit the integrated knowledge about employees, tasks, materials and work in the software development company, a new view(point) is aimed, which support managers to calculate real costs of tasks and to compare the real costs with the planned costs. Therefore, the new view should contain one entry for each task with planned costs, real costs and their difference. This new view is provided as Excel file to the manager in order to support

number formats in contrast to CSV. The final concrete syntax after the initialization of Costs is shown in the format Excel in Figure 11.11.

A1 | fx | id

	A	B	C	D	E	F	G	H	I	J	K
1	id	description	planDuratio	planCostsHu	planCostsMa	realDuration	realCostsHur	realCostsMat	planCostsTot	realCostsTot	additionalCosts
2	AP1	Architecture	3	800	700	3	807,69	105	1500	912,69	-587,3
3	AP2	Database	6	1600	1400	4	1134,61	280	3000	1414,61	-1585,39
4	AP3	Eventbus	5	1400	1100	2	523,07	100	2500	623,07	-1876,92
5	AP4	Network	6	1600	1400	0	0	0	3000	0	-3000
6											
7											
8											

Figure 11.11: The final concrete syntax of Costs in Excel format

The row numbers for tasks in Excel are depicted not as own column, but in the sidebar. The first five columns contain the same knowledge about tasks as in Tasks, since they represent the planned duration and costs of the tasks. Additionally, column F ("realDuration") contains the real number of days, which were worked by employees for the particular task. Column G ("realCostsHuman") contains the real costs for employees, depending on their salary and the number of worked days. Column H ("realCostsMaterial") contains the real costs for materials, depending on their costs and the number of worked days with used materials. The columns I ('planCostsTotal') and J ("realCostsTotal") sum the planned respectively real costs of humans and materials. Depending on these total costs, column K ("additionalCosts") calculates the additionally required money for each task, which might be negative. The metamodel of Costs is shown in Figure 11.12.

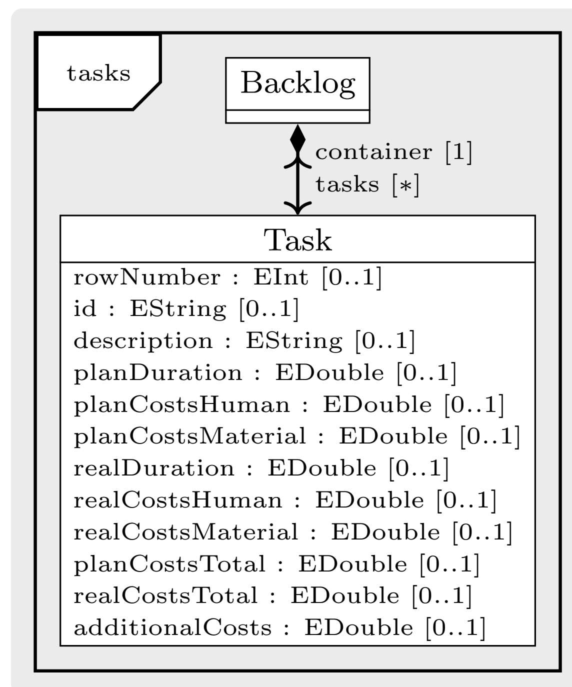


Figure 11.12: Metamodel of Costs

The metamodel is very similar to the metamodel of Tasks, but with additional attributes for the additional columns in the Excel file. The names in the first row of the Excel file are used for the attributes in `Task`, according to the Excel adapter as discussed

in Section 8.4.3☞273. The attributes for durations and costs have `EDouble`, since the Excel adapter supports only `EDouble` as data type for numbers. The final model after the initialization of Costs is shown in Figure 11.13.

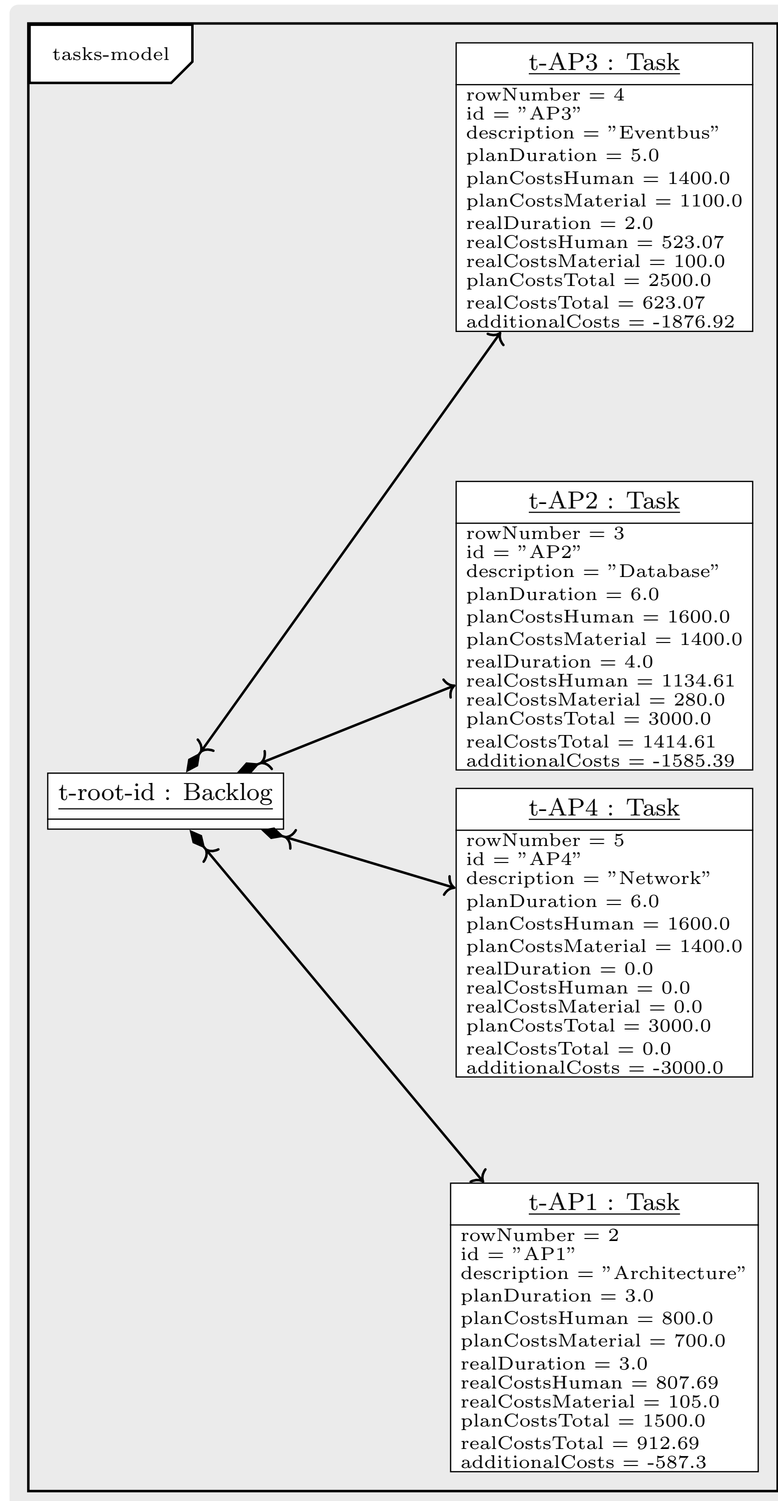


Figure 11.13: The final model after the initialization of Costs

According to the metamodel and the transformations of the Excel adapter, the model contains one object with `Task` as type for each row with content in the Excel file. The attribute slots are assigned with the content of the corresponding cell in the Excel file.

All `Task` objects are stored in an instance of `Backlog`.

11.1.7 Realization Overview

Figure 11.14 presents an overview about all consistency goals, annotated along the edges. Hyperlinks at the consistency goals allow to jump to their introductions. The nodes in the graphic represent the data sources in this application. Hyperlinks at the nodes allow to jump to their introductions.

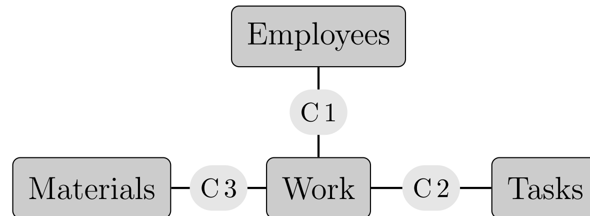


Figure 11.14: Overview about Consistency Goals in Knowledge Integration

The consistency is described by the following consistency goals and their concretizing consistency rules. Figure 11.14 visualizes the consistency goals and their involved data sources or new viewpoints.

Consistency Goal **C 1**	Work + Employees
Each work entry refers to its employee.	

Each work entry refers to the employee who did this work. Since the employee is not defined by its name, but by its personnal numbers, there is no need to react on changed names of employees in the work entries, since the personnal number is stable in the company. Additionally, employees are never removed, since they are kept for internal documentation of the work, even when they leave the company. This consistency goal is explicitly tested by the test cases documented in Section 11.4.2 ☞ 437, Section 11.4.4 ☞ 442 and Section 11.4.6 ☞ 449.

Consistency Goal **C 2**	Work + Tasks
Each work entry refers to a task.	

Each work entry refers to the task which was handled by this work. Since the task is not defined by its description, but by its ID, there is no need to react on changed descriptions of tasks in the work entries, since the ID is stable in the company. Additionally, tasks are never removed, since they are kept for internal documentation of the work, even when they become invalid. This consistency goal is explicitly tested by the test cases documented in Section 11.4.2 ☞ 437, Section 11.4.4 ☞ 442, Section 11.4.5 ☞ 446 and Section 11.4.6 ☞ 449.

Consistency Goal **C 3**	Work + Materials
Each work entry refers to all used materials.	

Each work entry refers to all the materials which were used for this work. Since the material is not defined by its label, but by its ID, there is no need to react on changed labels of materials in the work entries, since the ID is stable in the company. Additionally, materials are never removed, since they are kept for internal documentation of the work,

even when they are no longer used. This consistency goal is explicitly tested by the test cases documented in Section 11.4.2☞437, Section 11.4.4☞442 and Section 11.4.6☞449.

All operators configured for the realization are visualized in Figure 11.15☞403 along the edges. Data sources are rendered as white rectangles. New viewpoints are rendered as gray rectangles. Intermediate (meta)models are rendered as black circles. The operators for the integration of the data sources into the SU(M)M are described in Section 11.2☞404. The operators to derive the new view(point) from the SU(M)M are described in Section 11.3☞425.

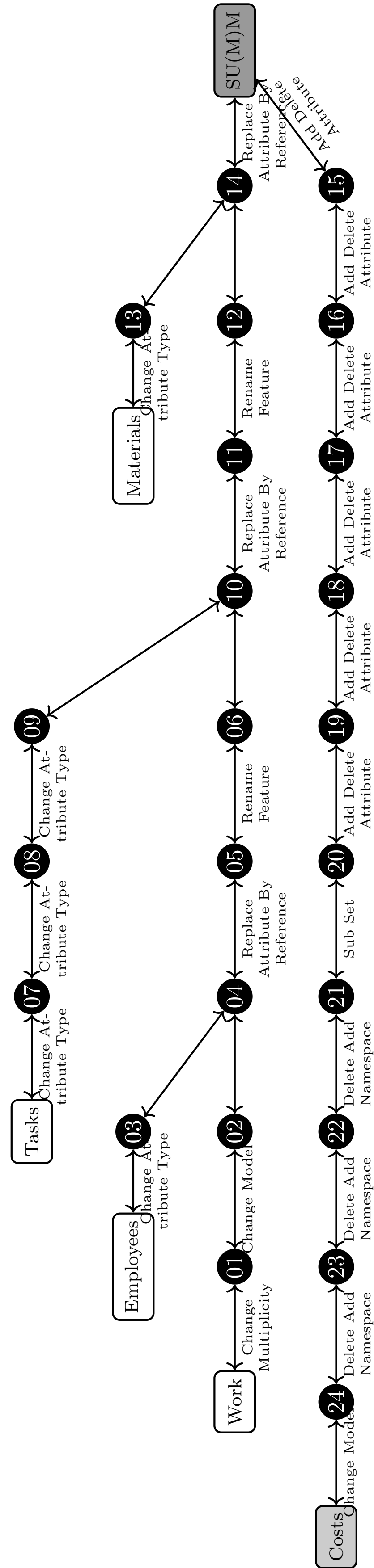


Figure 11.15: Configured Tree of Operators for Knowledge Integration

11.2 Integration of existing Data Sources

This section documents, how the existing data sources are integrated into the SU(M)M. For each used operator, its impact is highlighted and its configuration is sketched. The changes of the operator within the current metamodel are graphically visualized. The changes of the operator within the current model are graphically visualized. Only the combination of two (meta)models is not shown, since only two (meta)models are combined into one (meta)model on technical level without semantic changes.

In order to improve the readability, the documentation focuses on the most important information about the operators: For brevity, only the forward unidirectional operator is depicted in detail, while its inverse unidirectional operator is only mentioned. For some neighbored operators which realize similar objectives or work together for the same objective, the visualization of their impact is combined into a single graphic. Configurations for model decisions which are predefined by the operator and reused here, are not repeated again.

11.2.1 Improve Work

The orchestration starts with the central data source for work and improves it representation in order to simplify the following integrations with other data sources.

Work ⟷ 01: ChangeMultiplicity

In order to make the different materials explicit, the concatenation of materials with commas as separator is prevented by changing multiplicity of the attribute in order to allow an arbitrary number of materials in the metamodel. Additionally, the single materials are calculated by splitting their concatenation at the commas in the model. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.16.

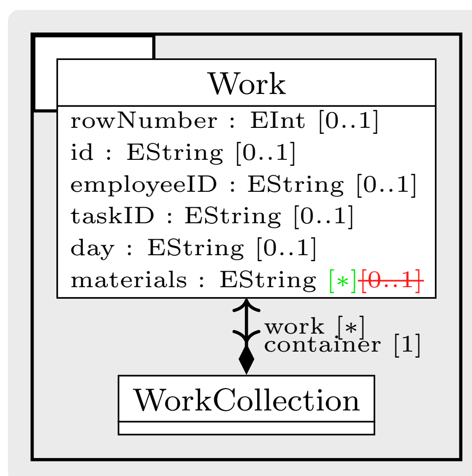


Figure 11.16: Metamodel Changes from Work to 01

Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.17 ☞ 405.

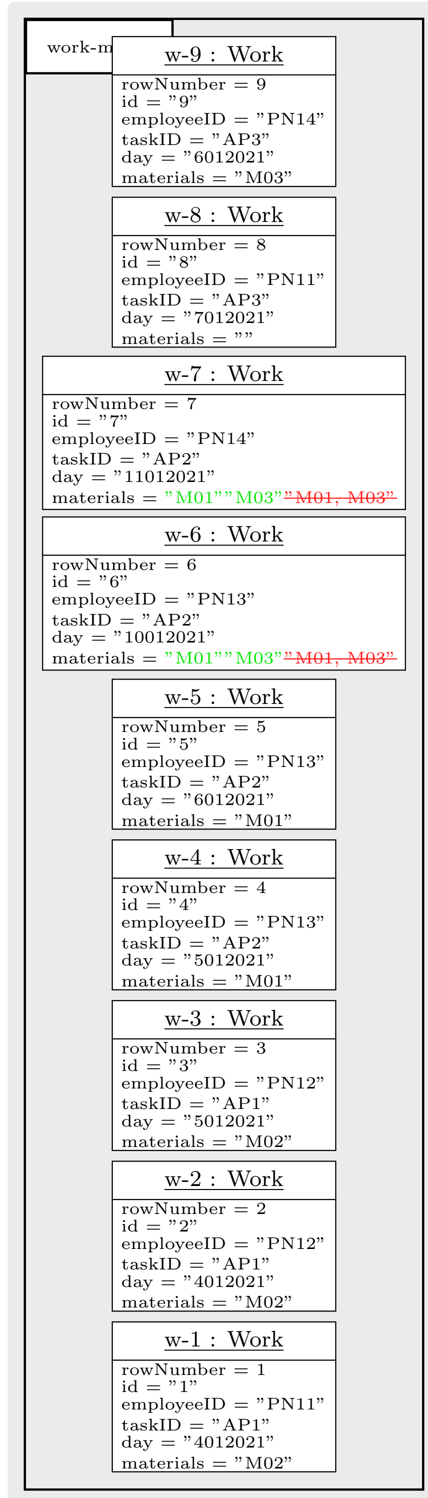


Figure 11.17: Model Changes from Work to 01

For the direction Work→01, the unidirectional operator is →ChangeMultiplicity (work.Work.materials: lower bound to 0, upper bound to -1).

Metamodel Decisions configured for →ChangeMultiplicity:

- fullyQualifiedFeatureName = work.Work.materials
- newLowerBound = 0

- newUpperBound = -1

Model Decisions configured for →ChangeMultiplicity: Configurations for model decisions are realized in `de▸unioldenburg▸se▸mmi▸example▸knowledge▸KnowledgeConfiguration▸1`. All their configurations for model decisions are listed here:

- handleInstanceWithHurtLowerBound (arg0 : `Slot`, arg1 : `int`)
  This case does not occur here.
- handleInstanceWithHurtUpperBound (arg0 : `Slot`, arg1 : `int`)
  This case does not occur here.
- handleInstanceWithValidBounds (arg0 : `Slot`, arg1 : `int`, arg2 : `int`)
  Splits the concatenation of materials into the single materials at the commas.

For the inverse direction Work←01, the unidirectional operator is ←ChangeMultiplicity (work.Work.materials: lower bound to 0, upper bound to 1).

01 ⟷ 02: ChangeModel

In order to improve the readability of the date information, the formatting is changed from values like 22112000 to 2000-11-22 (and vice versa). Therefore, this part of the orchestration does not change the metamodel. Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.18☞407.

w-9 : Work
rowNumber = 9
id = "9"
employeeID = "PN14"
taskID = "AP3"
day = "2021-01-06"~~"6012021"~~
materials = "M03"

w-8 : Work
rowNumber = 8
id = "8"
employeeID = "PN11"
taskID = "AP3"
day = "2021-01-07"~~"7012021"~~
materials = ""

w-7 : Work
rowNumber = 7
id = "7"
employeeID = "PN14"
taskID = "AP2"
day = "2021-01-11"~~"11012021"~~
materials = "M01""M03"

w-6 : Work
rowNumber = 6
id = "6"
employeeID = "PN13"
taskID = "AP2"
day = "2021-01-10"~~"10012021"~~
materials = "M01""M03"

w-5 : Work
rowNumber = 5
id = "5"
employeeID = "PN13"
taskID = "AP2"
day = "2021-01-06"~~"6012021"~~
materials = "M01"

w-4 : Work
rowNumber = 4
id = "4"
employeeID = "PN13"
taskID = "AP2"
day = "2021-01-05"~~"5012021"~~
materials = "M01"

w-3 : Work
rowNumber = 3
id = "3"
employeeID = "PN12"
taskID = "AP1"
day = "2021-01-05"~~"5012021"~~
materials = "M02"

w-2 : Work
rowNumber = 2
id = "2"
employeeID = "PN12"
taskID = "AP1"
day = "2021-01-04"~~"4012021"~~
materials = "M02"

w-1 : Work
rowNumber = 1
id = "1"
employeeID = "PN11"
taskID = "AP1"
day = "2021-01-04"~~"4012021"~~
materials = "M02"

Figure 11.18: Model Changes from 01 to 02

For the direction 01→02, the unidirectional operator is →ChangeModel (ChangeModel).

Metamodel Decisions are not used by →ChangeModel.

Model Decisions configured for →ChangeModel: Configurations for model decisions are realized in `de▸unioldenburg▸se▸mmi▸example▸knowledge▸KnowledgeConfig-`

uration‣3. All their configurations for model decisions are listed here:

- changeModel ( )
  Converts values like 22112000 to 2000-11-22 and adds a possibly missing leading zero before.

For the inverse direction 01←02, the unidirectional operator is ←ChangeModel (ChangeModel).

11.2.2 Integrate Employees with Work

Adds the data source for employees, improves it and integrates it with work.

Employees ⟷ 03: ChangeAttributeType

Since the CSV format supports only text, some values are converted from String to double in order to ease numeric calculations. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.19.

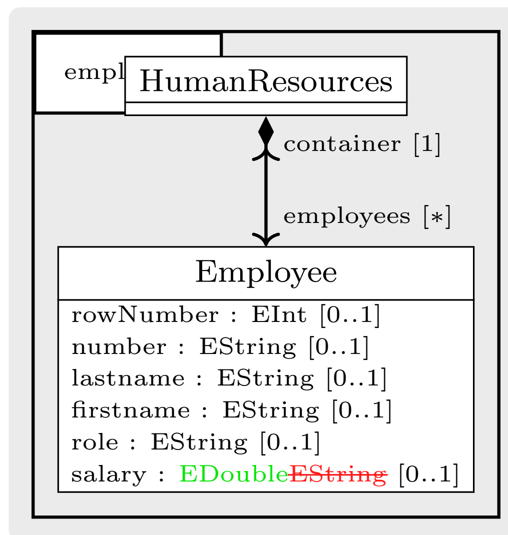


Figure 11.19: Metamodel Changes from Employees to 03

Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.20 ☞ 409.

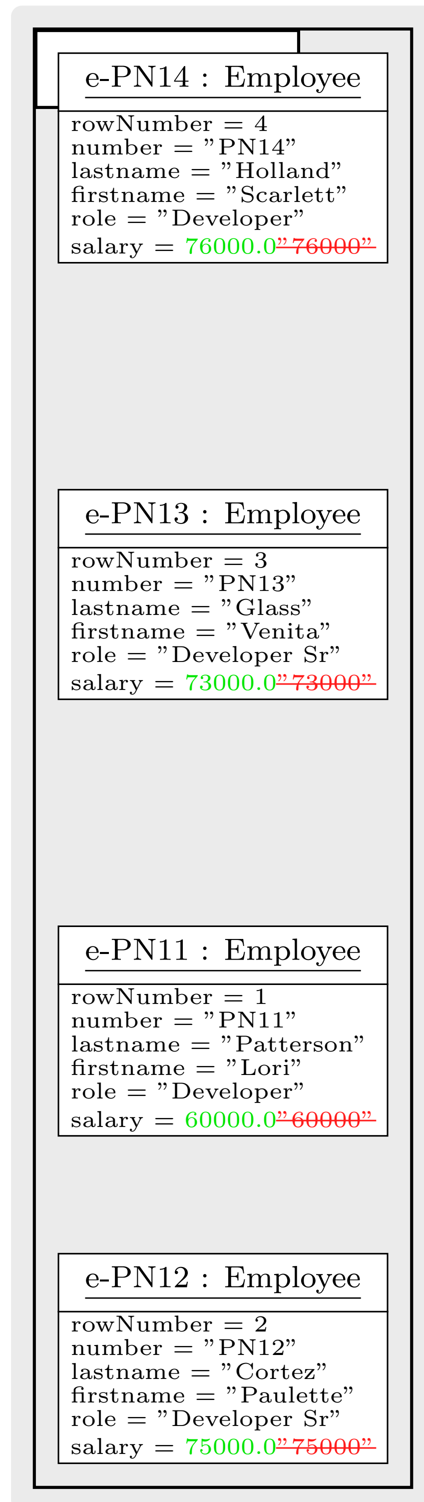


Figure 11.20: Model Changes from Employees to 03

For the direction Employees→03, the unidirectional operator is →ChangeAttributeType (attribute employees.Employee.salary: ecore.EString → ecore.EDouble).

Metamodel Decisions configured for →ChangeAttributeType:

- fullyQualifiedAttributeName = employees.Employee.salary
- fullyQualifiedNewTypeName = ecore.EDouble

Model Decisions configured for →ChangeAttributeType: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣ConvertStringToDouble`. All their configurations for model decisions are listed here:

- convert (input : `Object`) : `Object`
  Converts the String value to double by using the default Java parsing method. This is a default configuration, reused from `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣ConvertStringToDouble`.

For the inverse direction Employees←03, the unidirectional operator is ←ChangeAttributeType (attribute employees.Employee.salary: ecore.EDouble → ecore.EString).

04 ⟷ 05: ReplaceAttributeByReference

In order to realize C 1 by linking each work entry explicitly to the employee, this operator replaces the personnal numbers by explicit links pointing to the corresponding employees. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.21.

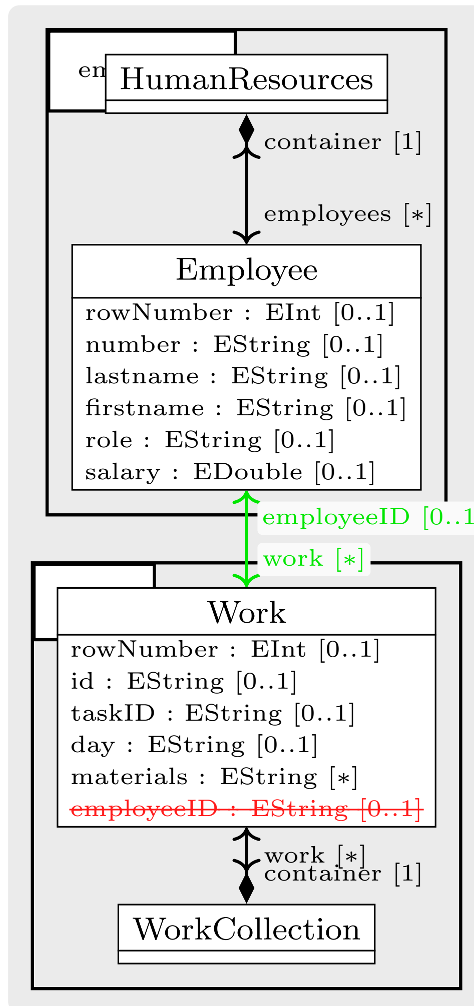


Figure 11.21: Metamodel Changes from 04 to 05

Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.22☞411.

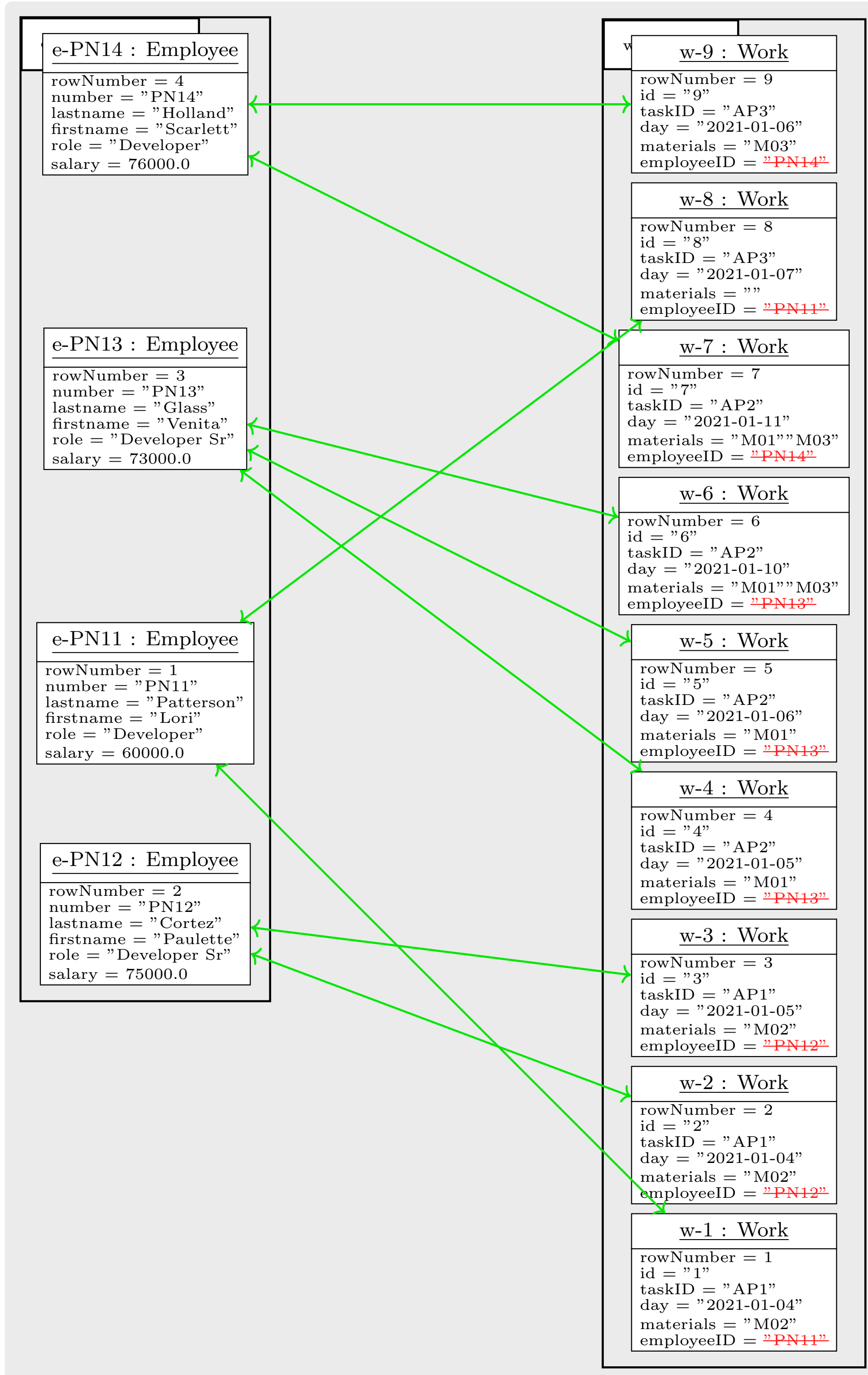


Figure 11.22: Model Changes from 04 to 05

For the direction 04→05, the unidirectional operator is →REPLACEATTRIBUTEBY-REFERENCE (work.Work.employeeID ⟶ employees.Employee).

Metamodel Decisions configured for →REPLACEATTRIBUTEBYREFERENCE:

- sourceClassName = work.Work
- attributeName = employeeID

- targetClassName = employees.Employee
- oppositeReferenceName = work
- oppositeLowerBound = 0
- oppositeUpperBound = -1

Model Decisions configured for →REPLACEATTRIBUTEBYREFERENCE: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣example‣knowledge‣KnowledgeConfiguration‣5`. All their configurations for model decisions are listed here:

- replaceValue (arg0 : `AttributeSlot`, arg1 : `Object`, arg2 : `ReplaceAttributeByReference`) : `Instance`
  Searches all employees to find the employee with a matching number.

For the inverse direction 04←05, the unidirectional operator is ←REPLACEREFERENCEBYATTRIBUTE (work.Work.employeeID : employees.Employee).

05 ⟷ 06: RenameFeature

After the previous replacement, the reference points directly to the objects, which should be replected by its name. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.23☞413.

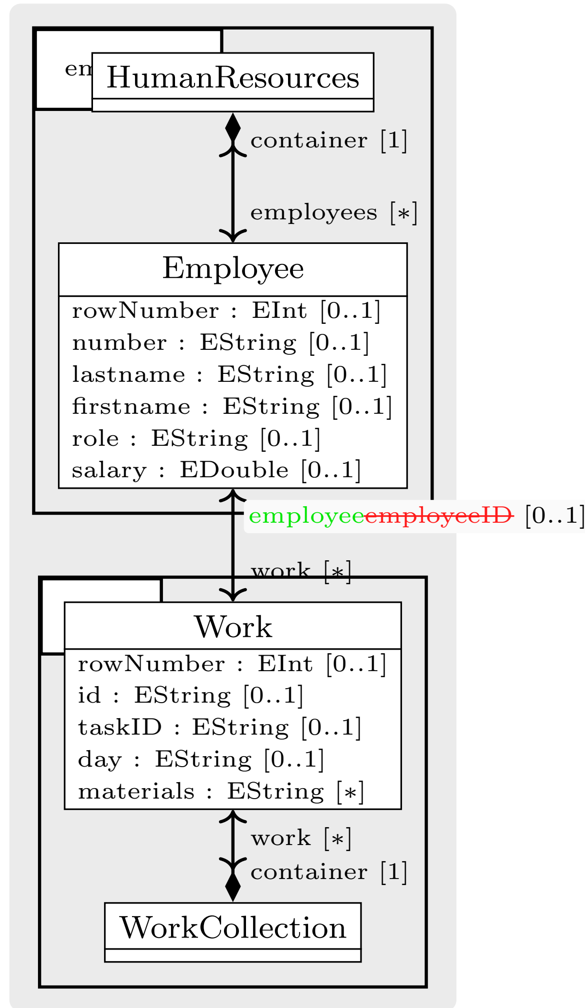


Figure 11.23: Metamodel Changes from 05 to 06

Accordingly, this part of the orchestration does not change the model. For the direction 05→06, the unidirectional operator is →RENAMEFEATURE (work.Work.employeeID → employee).

Metamodel Decisions configured for →RENAMECLASSIFIER:

- elementFullyQualified = work.Work.employeeID
- name = employee

Model Decisions configured for →RENAMECLASSIFIER: This operator has no configurations for model decisions.

For the inverse direction 05←06, the unidirectional operator is ←RENAMEFEATURE (work.Work.employee → employeeID).

11.2.3 Integrate Tasks with Work

Adds the data source for tasks, improves it and integrates it with work.

Tasks ⟷ 10

Since the CSV format supports only text, some values are converted from String to double in order to ease numeric calculations. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.24.

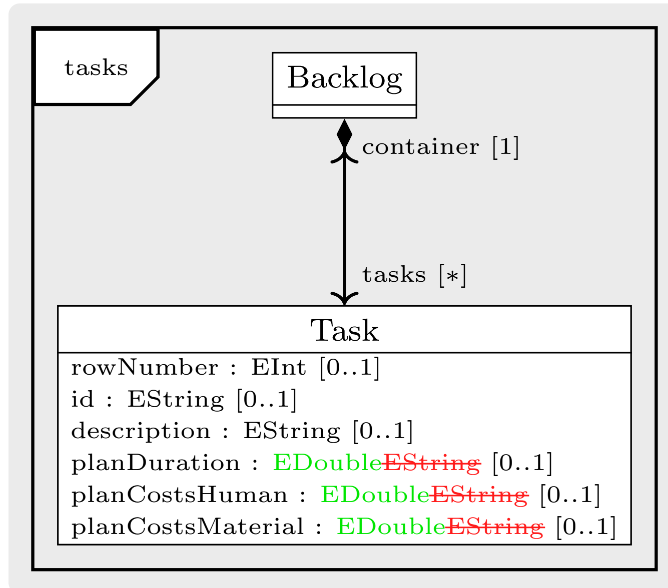


Figure 11.24: Metamodel Changes from Tasks to 10

Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.25 ☞ 415.

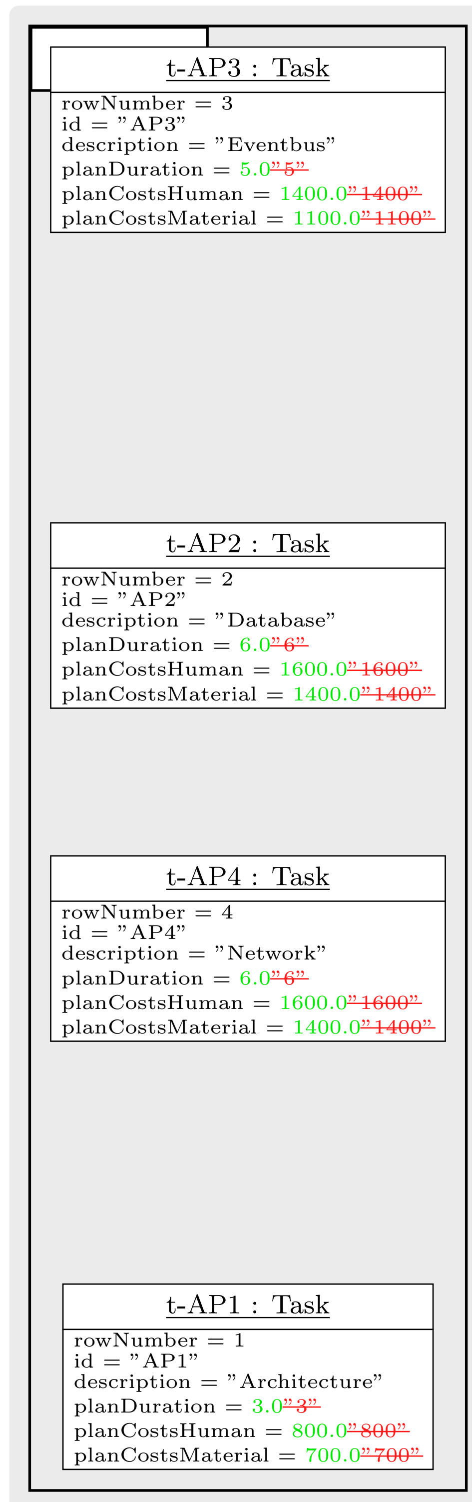


Figure 11.25: Model Changes from Tasks to 10

This is realized by the following three operators:

1. For the direction Tasks→07, the unidirectional operator is →ChangeAttributeType (attribute tasks.Task.planDuration: ecore.EString → ecore.EDouble).

Metamodel Decisions configured for →ChangeAttributeType:

- fullyQualifiedAttributeName = tasks.Task.planDuration
- fullyQualifiedNewTypeName = ecore.EDouble

Model Decisions configured for →ChangeAttributeType: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣ConvertStringToDouble`. All their configurations for model decisions are listed here:

- convert (input : `Object`) : `Object`
  Converts the String value to double by using the default Java parsing method. This is a default configuration, reused from `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣ConvertStringToDouble`.

For the inverse direction Tasks←07, the unidirectional operator is ←ChangeAttributeType (attribute tasks.Task.planDuration: ecore.EDouble → ecore.EString).

2. For the direction 07→08, the unidirectional operator is →ChangeAttributeType (attribute tasks.Task.planCostsHuman: ecore.EString → ecore.EDouble).

Metamodel Decisions configured for →ChangeAttributeType:

- fullyQualifiedAttributeName = tasks.Task.planCostsHuman
- fullyQualifiedNewTypeName = ecore.EDouble

Model Decisions configured for →ChangeAttributeType: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣ConvertStringToDouble`. All their configurations for model decisions are listed here:

- convert (input : `Object`) : `Object`
  Converts the String value to double by using the default Java parsing method. This is a default configuration, reused from `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣ConvertStringToDouble`.

For the inverse direction 07←08, the unidirectional operator is ←ChangeAttributeType (attribute tasks.Task.planCostsHuman: ecore.EDouble → ecore.EString).

3. For the direction 08→09, the unidirectional operator is →ChangeAttributeType (attribute tasks.Task.planCostsMaterial: ecore.EString → ecore.EDouble).

Metamodel Decisions configured for →ChangeAttributeType:

- fullyQualifiedAttributeName = tasks.Task.planCostsMaterial
- fullyQualifiedNewTypeName = ecore.EDouble

Model Decisions configured for →ChangeAttributeType: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣framework‣oper-`

ator‣unidirectional‣decisions‣ConvertStringToDouble. All their configurations for model decisions are listed here:

- convert (input : `Object`) : `Object`
  Converts the String value to double by using the default Java parsing method. This is a default configuration, reused from `de‣unioldenburg‣se‣mmi‣framework‣operator‣unidirectional‣decisions‣ConvertStringToDouble`.

For the inverse direction ⓿⑧←⓿⑨, the unidirectional operator is ←ChangeAttributeType (attribute tasks.Task.planCostsMaterial: ecore.EDouble → ecore.EString).

❿ ⟷ ⓫: ReplaceAttributeByReference

In order to realize C 2 by linking each work entry explicitly to the task, this operator replaces the ID of the task by explicit links pointing to this task. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.26.

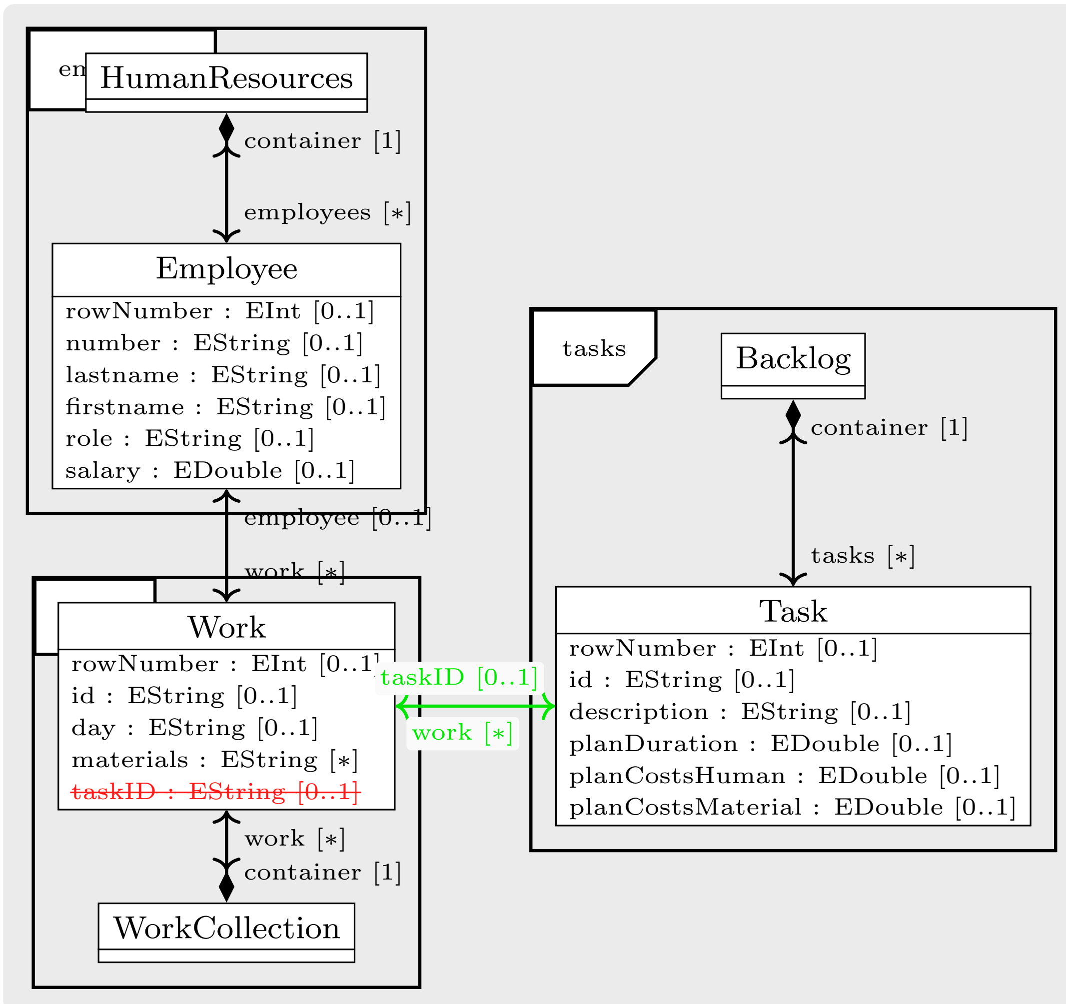


Figure 11.26: Metamodel Changes from ❿ to ⓫

Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.27☞418.

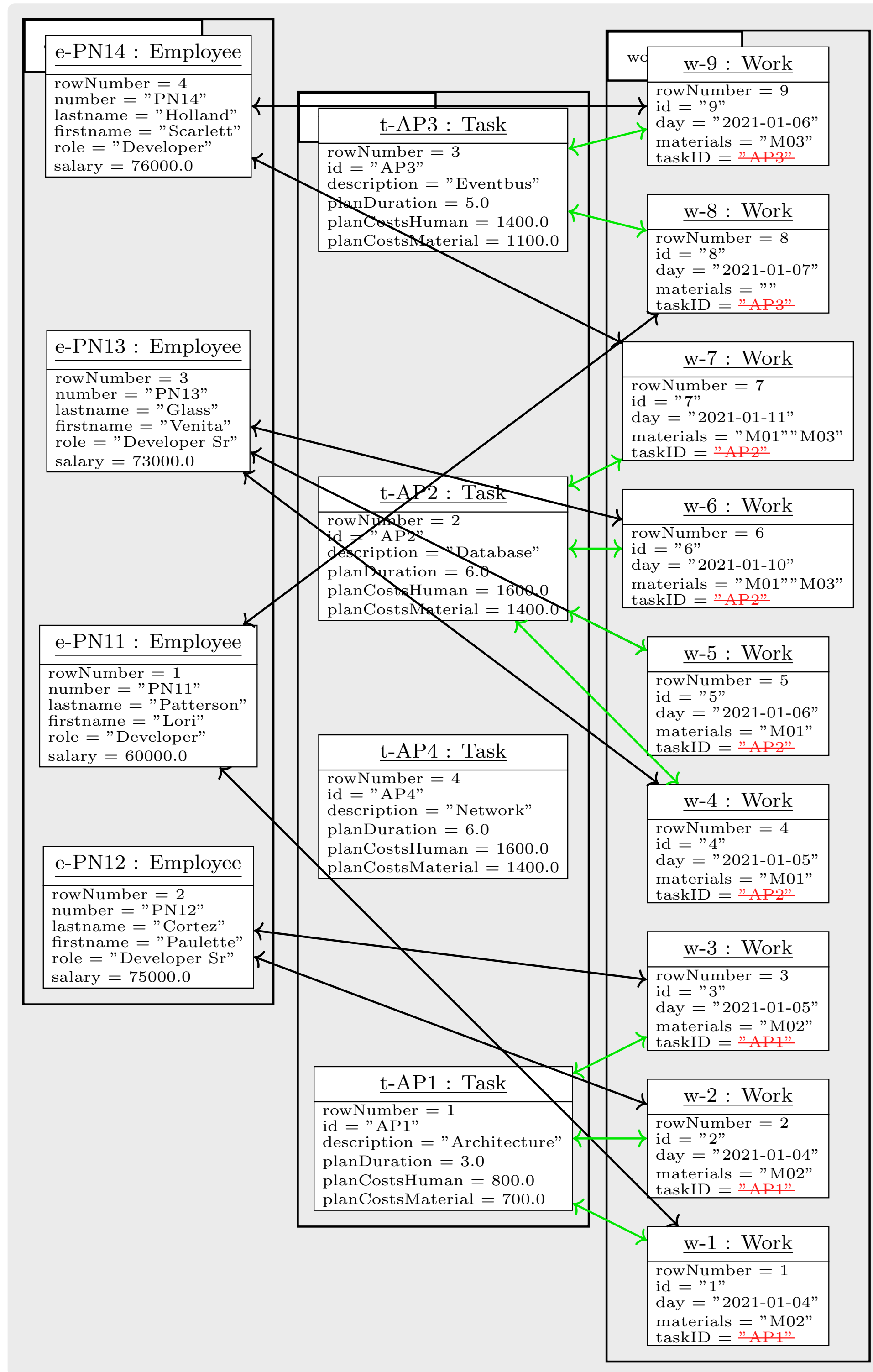


Figure 11.27: Model Changes from ⓾ to ⑪

For the direction ⓾→⑪, the unidirectional operator is →ReplaceAttributeByReference (work.Work.taskID ⟶ tasks.Task).

Metamodel Decisions configured for →ReplaceAttributeByReference:

- sourceClassName = work.Work
- attributeName = taskID

- targetClassName = tasks.Task
- oppositeReferenceName = work
- oppositeLowerBound = 0
- oppositeUpperBound = -1

Model Decisions configured for →ReplaceAttributeByReference: Configurations for model decisions are realized in `de▸unioldenburg▸se▸mmi▸example▸knowledge▸KnowledgeConfiguration▸7`. All their configurations for model decisions are listed here:

- replaceValue (arg0 : `AttributeSlot`, arg1 : `Object`, arg2 : `ReplaceAttributeByReference`) : `Instance`
  Searches all tasks to find the task with a matching id.

For the inverse direction ❿←⓫, the unidirectional operator is ←ReplaceReferenceByAttribute (work.Work.taskID : tasks.Task).

⓫ ⟷ ⓬: RenameFeature

After the previous replacement, the reference points directly to the objects, which should be replected by its name. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.28☞420.

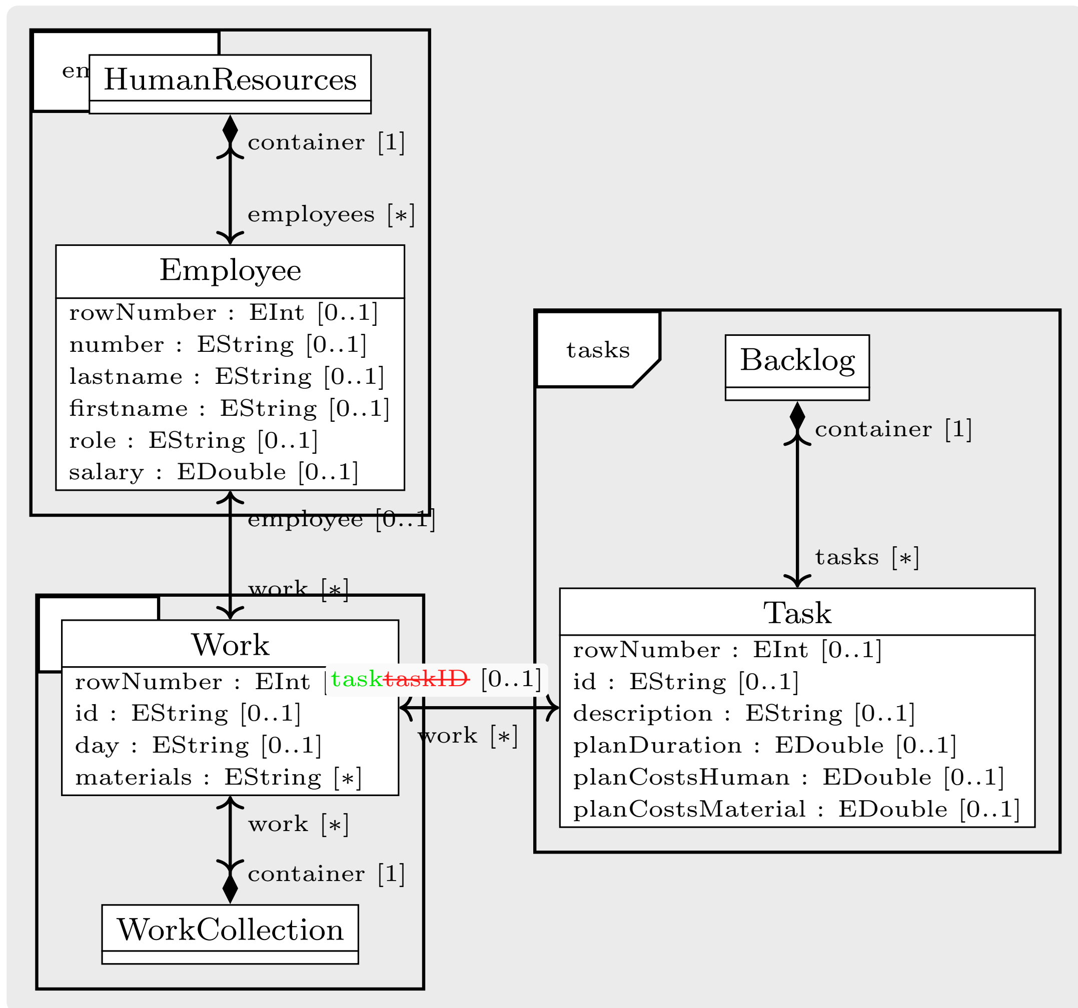


Figure 11.28: Metamodel Changes from ⓫ to ⓬

Accordingly, this part of the orchestration does not change the model. For the direction ⓫→⓬, the unidirectional operator is →RenameFeature (work.Work.taskID → task).

Metamodel Decisions configured for →RenameClassifier:

- elementFullyQualified = work.Work.taskID
- name = task

Model Decisions configured for →RenameClassifier: This operator has no configurations for model decisions.

For the inverse direction ⓫←⓬, the unidirectional operator is ←RenameFeature (work.Work.task → taskID).

11.2.4 Integrate Materials with Work

Adds the data source for materials, improves it and integrates it with work.

Materials ⟷ ⓭: ChangeAttributeType

Since the CSV format supports only text, some values are converted from String to double in order to ease numeric calculations. Therefore, this part of the orchestration changes the

metamodel, as depicted in Figure 11.29.

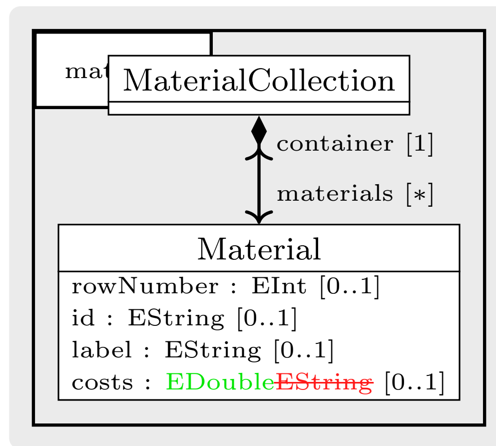


Figure 11.29: Metamodel Changes from Materials to 13

Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.30 ☞ 422.

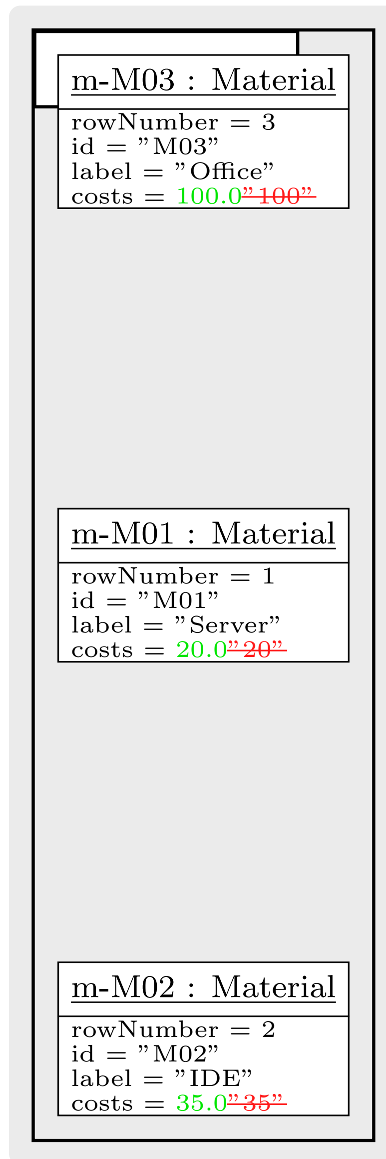


Figure 11.30: Model Changes from Materials to 13

For the direction Materials→13, the unidirectional operator is →ChangeAttributeType (attribute materials.Material.costs: ecore.EString → ecore.EDouble).

Metamodel Decisions configured for →ChangeAttributeType:

- fullyQualifiedAttributeName = materials.Material.costs
- fullyQualifiedNewTypeName = ecore.EDouble

Model Decisions configured for →ChangeAttributeType: Configurations for model decisions are realized in `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertStringToDouble`. All their configurations for model decisions are listed here:

- convert (input : `Object`) : `Object`
  Converts the String value to double by using the default Java parsing method. This is a default configuration, reused from `de▸unioldenburg▸se▸mmi▸framework▸operator▸unidirectional▸decisions▸ConvertStringToDouble`.

For the inverse direction Materials←13, the unidirectional operator is ←ChangeAttributeType (attribute materials.Material.costs: ecore.EDouble → ecore.EString).

14 ⟷ SUMM: ReplaceAttributeByReference

In order to realize C 3 by linking each work entry explicitly to the material, this operator replaces the ID of the material by explicit links pointing to this material. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.31.

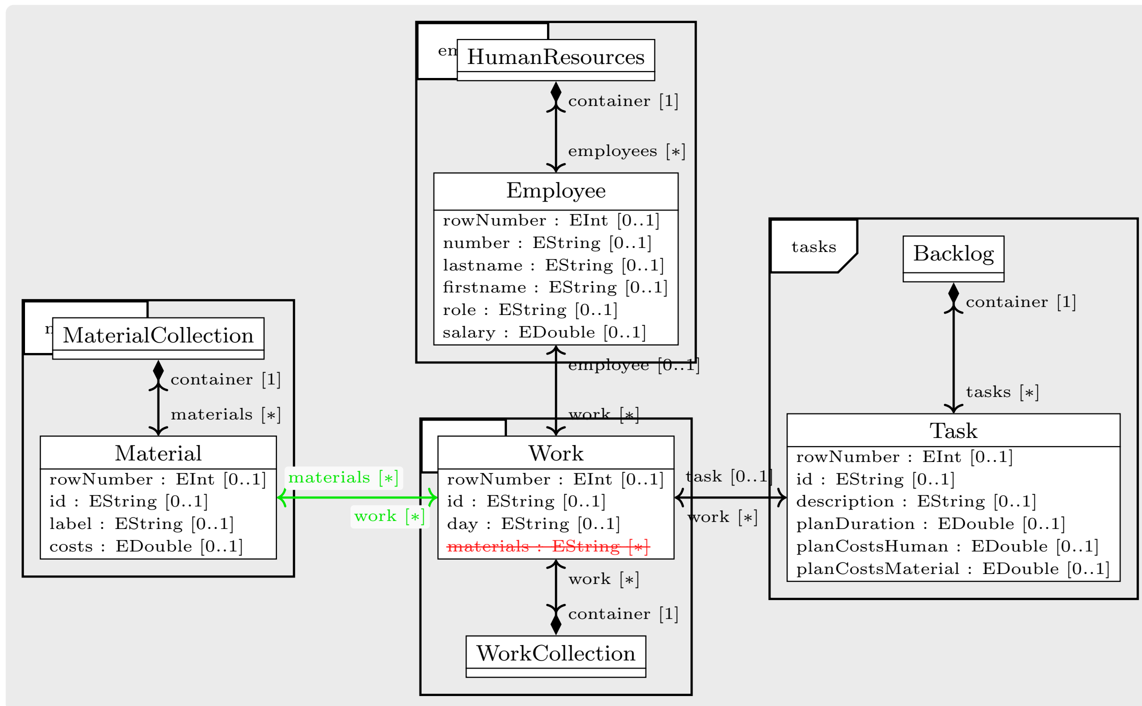


Figure 11.31: Metamodel Changes from 14 to SUMM

Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.32☞424.

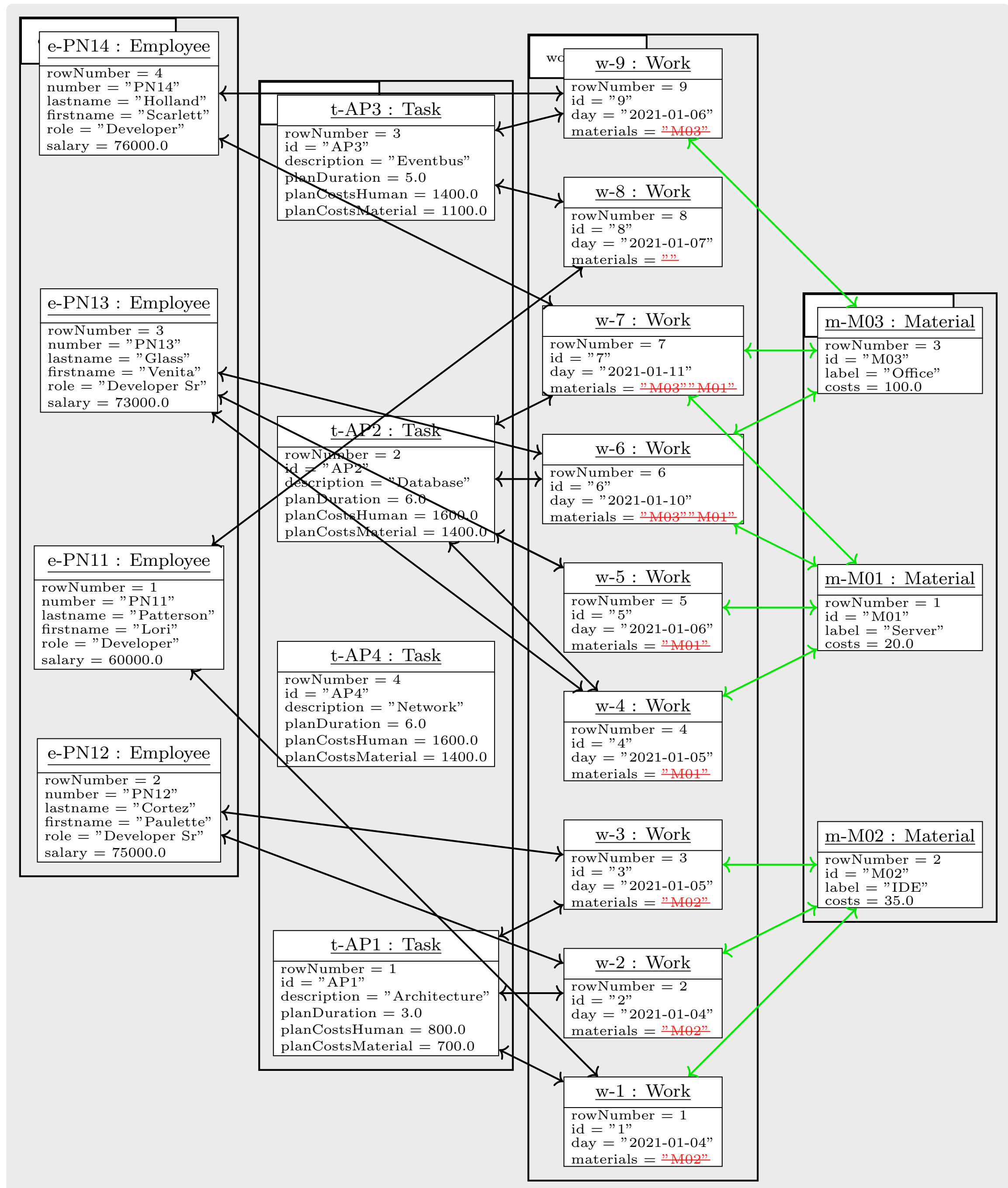


Figure 11.32: Model Changes from 14 to SUMM

For the direction 14→SUMM, the unidirectional operator is →ReplaceAttributeByReference (work.Work.materials ⟶ materials.Material).

Metamodel Decisions configured for →ReplaceAttributeByReference:

- sourceClassName = work.Work
- attributeName = materials
- targetClassName = materials.Material
- oppositeReferenceName = work
- oppositeLowerBound = 0

- oppositeUpperBound = -1

Model Decisions configured for →REPLACEATTRIBUTEBYREFERENCE: Configurations for model decisions are realized in `de▸unioldenburg▸se▸mmi▸example▸knowledge▸KnowledgeConfiguration▸9`. All their configurations for model decisions are listed here:

- replaceValue (arg0 : `AttributeSlot`, arg1 : `Object`, arg2 : `ReplaceAttributeByReference`) : `Instance`
  Searches all materials to find the material with a matching id.

For the inverse direction 14←SUMM, the unidirectional operator is ←REPLACEREFERENCEBYATTRIBUTE (work.Work.materials : materials.Material).

11.3 Definition of a new View(point)

This section documents, how the new view(point) is derived from the SU(M)M. For each used operator, its impact is highlighted and its configuration is sketched.

SUMM ⟷ 20

Basing on the whole information in the SUM, the additional values for each task are derived. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.33.

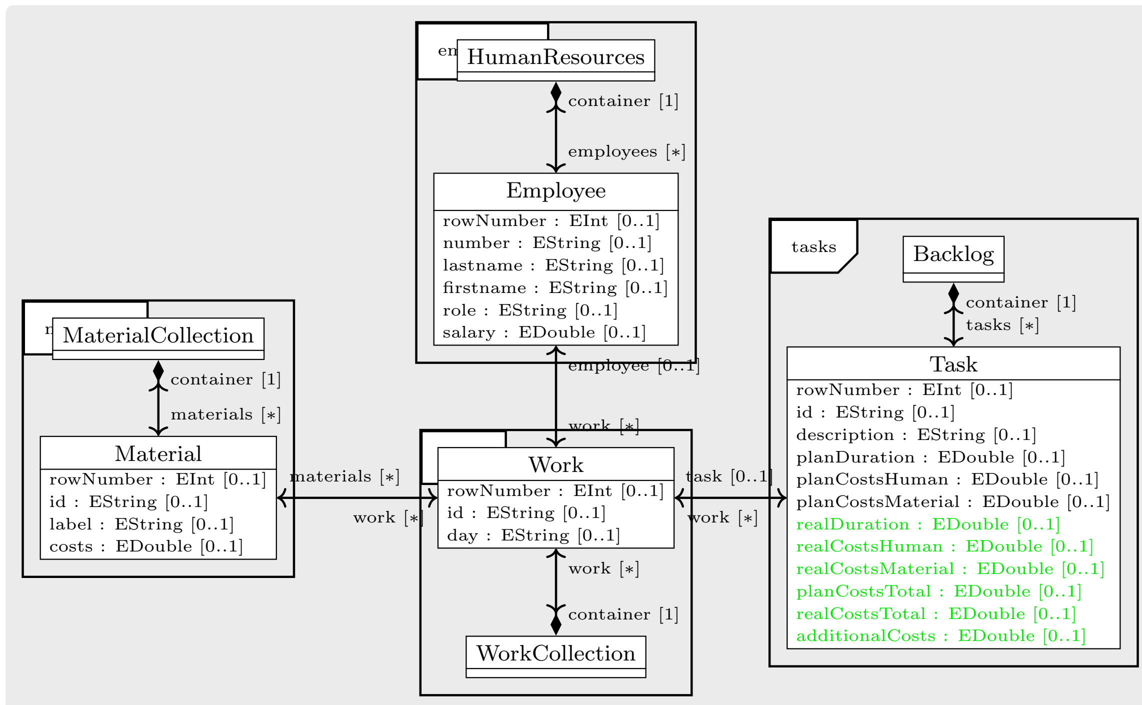


Figure 11.33: Metamodel Changes from SUMM to 20

Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.34☞426.

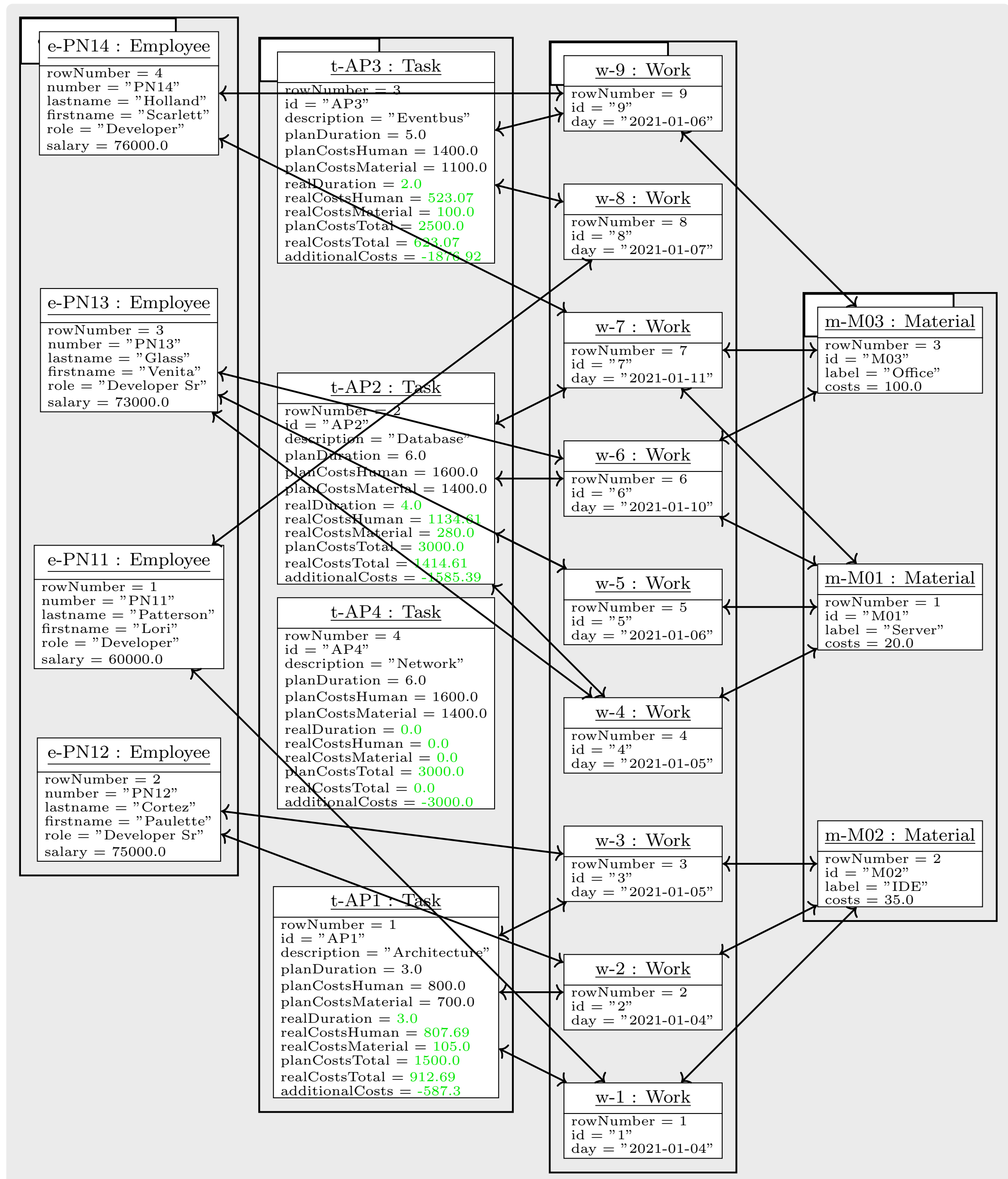


Figure 11.34: Model Changes from SUMM to 20

This is realized by the following six operators:

1. For the direction SUMM→15, the unidirectional operator is →AddAttribute (tasks.Task.realDuration).

   Calculates the real duration for each task and stores these read-only values in a new attribute.

Metamodel Decisions configured for →AddAttribute:

- classWithNewAttributeFullyQualified = tasks.Task
- attributeName = realDuration

- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EDouble

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣example‣knowledge‣KnowledgeConfiguration‣11`. All their configurations for model decisions are listed here:

- computeInitialValue (arg0 : `Instance`, arg1 : `EAttribute`) : `Object`
  Calculates the real duration by counting the number of registered work for the current task.

For the inverse direction SUMM←⓯, the unidirectional operator is ←DeleteAttribute (tasks.Task.realDuration).

2. For the direction ⓯→⓰, the unidirectional operator is →AddAttribute (tasks.Task.realCostsHuman).

   Calculates the real costs for humans for each task and stores these read-only values in a new attribute.

Metamodel Decisions configured for →AddAttribute:

- classWithNewAttributeFullyQualified = tasks.Task
- attributeName = realCostsHuman
- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EDouble

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣example‣knowledge‣KnowledgeConfiguration‣12`. All their configurations for model decisions are listed here:

- computeInitialValue (arg0 : `Instance`, arg1 : `EAttribute`) : `Object`
  Calculates the real costs for humans by calculating the day-costs for the employees who worked for the current task.

For the inverse direction ⓯←⓰, the unidirectional operator is ←DeleteAttribute (tasks.Task.realCostsHuman).

3. For the direction ⓰→⓱, the unidirectional operator is →AddAttribute (tasks.Task.realCostsMaterial).

   Calculates the real costs for materials for each task and stores these read-only values in a new attribute.

Metamodel Decisions configured for →AddAttribute:

- classWithNewAttributeFullyQualified = tasks.Task
- attributeName = realCostsMaterial

- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EDouble

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣example‣knowledge‣KnowledgeConfiguration‣13`. All their configurations for model decisions are listed here:

- computeInitialValue (arg0 : `Instance`, arg1 : `EAttribute`) : `Object`
  Calculates the real costs for materials by calculating the day-costs for all used meterials of each registered work for the current task.

For the inverse direction ⓰←⓱, the unidirectional operator is ←DeleteAttribute (tasks.Task.realCostsMaterial).

4. For the direction ⓱→⓲, the unidirectional operator is →AddAttribute (tasks.Task.planCostsTotal).

   Calculates the planned total costs for each task and stores these read-only values in a new attribute.

Metamodel Decisions configured for →AddAttribute:

- classWithNewAttributeFullyQualified = tasks.Task
- attributeName = planCostsTotal
- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EDouble

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣example‣knowledge‣KnowledgeConfiguration‣14`. All their configurations for model decisions are listed here:

- computeInitialValue (arg0 : `Instance`, arg1 : `EAttribute`) : `Object`
  Sums the planned costs for humans and the planned costs for materials for the current task.

For the inverse direction ⓱←⓲, the unidirectional operator is ←DeleteAttribute (tasks.Task.planCostsTotal).

5. For the direction ⓲→⓳, the unidirectional operator is →AddAttribute (tasks.Task.realCostsTotal).

   Calculate the real total costs for each task and stores these read-only values in a new attribute.

Metamodel Decisions configured for →AddAttribute:

- classWithNewAttributeFullyQualified = tasks.Task
- attributeName = realCostsTotal

- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EDouble

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣example‣knowledge‣KnowledgeConfiguration‣15`. All their configurations for model decisions are listed here:

- computeInitialValue (arg0 : `Instance`, arg1 : `EAttribute`) : `Object`
  Sums the real costs of humans and materials for the current task.

For the inverse direction ⓲←⓳, the unidirectional operator is ←DeleteAttribute (tasks.Task.realCostsTotal).

6. For the direction ⓳→⓴, the unidirectional operator is →AddAttribute (tasks.Task.additionalCosts).

   Subtracts the additional costs for each task and stores these read-only values in a new attribute.

Metamodel Decisions configured for →AddAttribute:

- classWithNewAttributeFullyQualified = tasks.Task
- attributeName = additionalCosts
- attributeLowerBound = 0
- attributeUpperBound = 1
- attributeDataTypeFullyQualified = ecore.EDouble

Model Decisions configured for →AddAttribute: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣example‣knowledge‣KnowledgeConfiguration‣16`. All their configurations for model decisions are listed here:

- computeInitialValue (arg0 : `Instance`, arg1 : `EAttribute`) : `Object`
  Subtracts the planned costs from the real costs for the current task.

For the inverse direction ⓳←⓴, the unidirectional operator is ←DeleteAttribute (tasks.Task.additionalCosts).

⓴ ⟷ ㉔

After calculating the values for the additional attributes for each task, most parts of the (meta)model are not required anymore and are removed. Therefore, this part of the orchestration changes the metamodel, as depicted in Figure 11.35☞430.

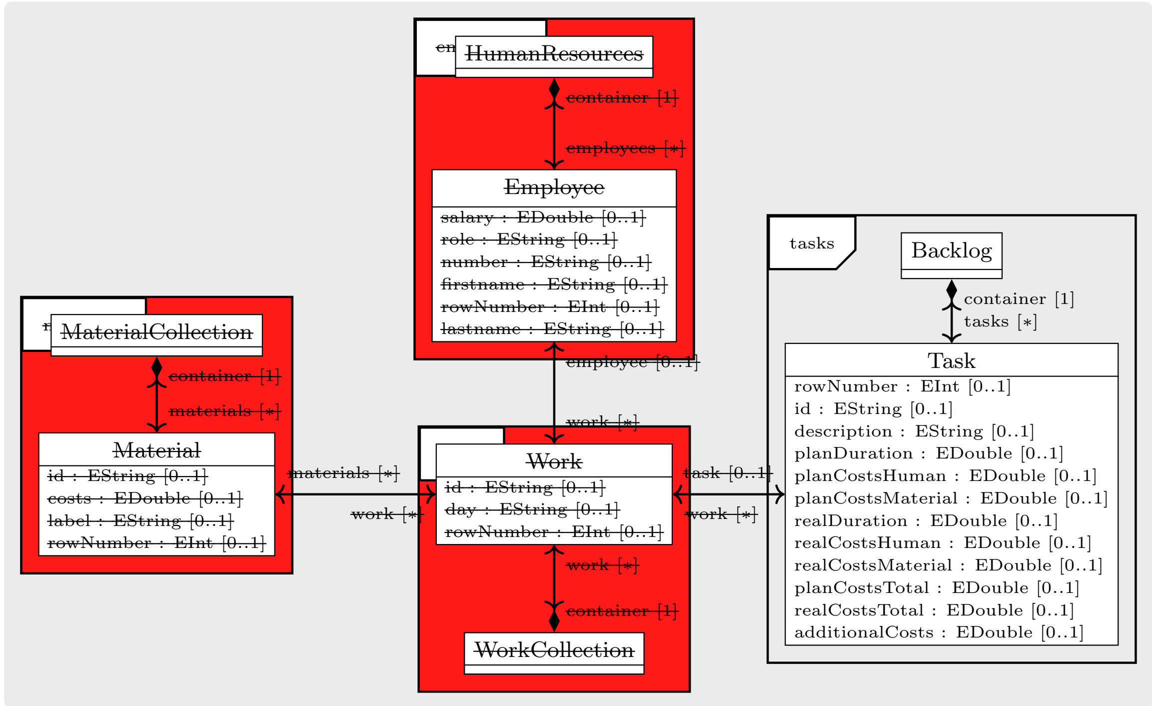


Figure 11.35: Metamodel Changes from ⓴ to ㉔

Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.36 ☞ 431.

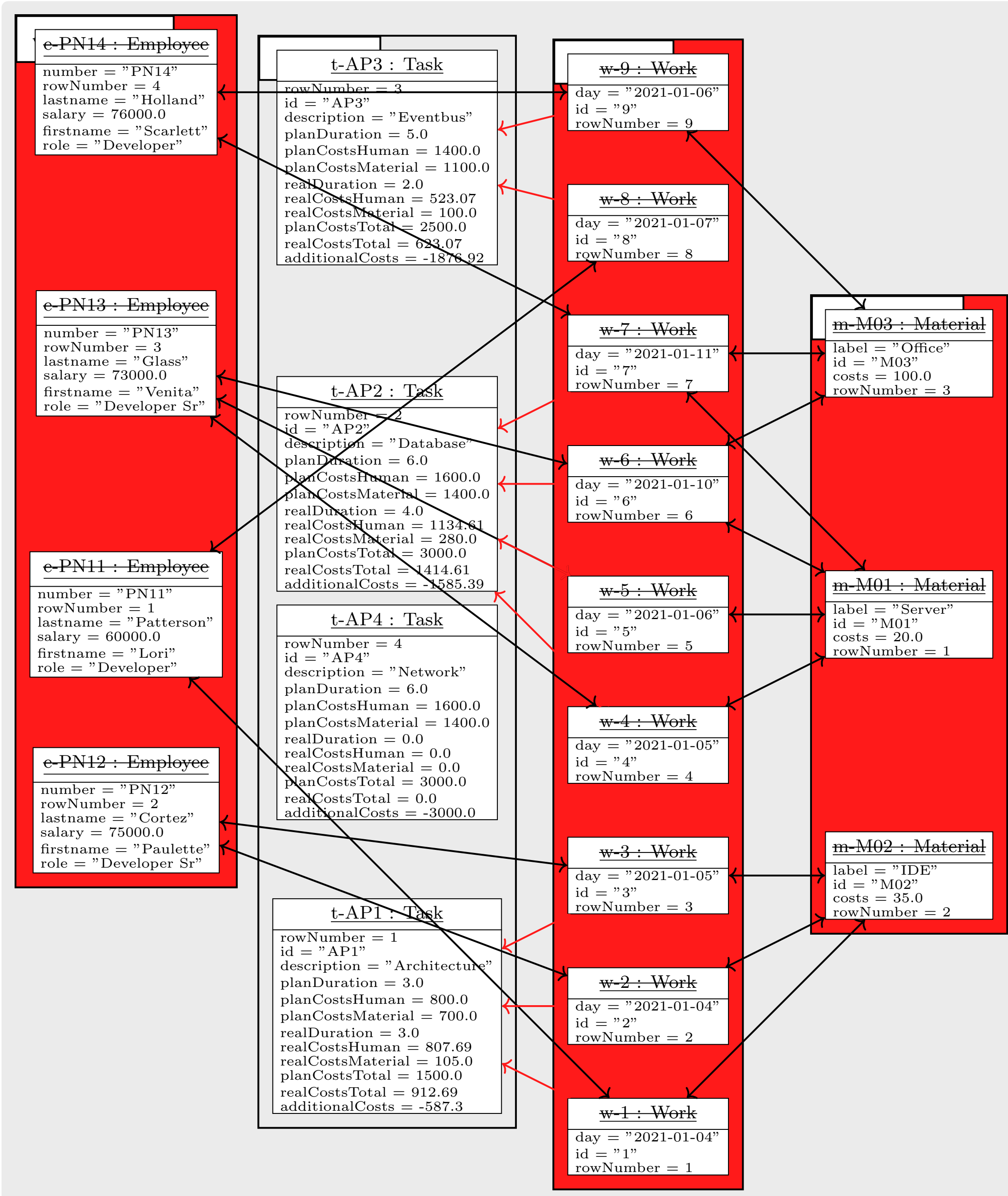


Figure 11.36: Model Changes from ⑳ to ㉔

This is realized by the following four operators:

1. For the direction ⑳→㉑, the unidirectional operator is →SubSetFilter (tasks.Task.work, work, employees, materials).

Metamodel Decisions are not used by →SubSetFilter.

Model Decisions configured for →SubSetFilter: This operator has no configurations for model decisions.

For the inverse direction ⑳←㉑, the unidirectional operator is ←SubSetRecreate (tasks.Task.work, work, employees, materials).

2. For the direction ㉑→㉒, the unidirectional operator is →DeleteNamespace (work-model).

Metamodel Decisions configured for →DeleteNamespace:

- namespaceFullName = work-model

Model Decisions configured for →DeleteNamespace: This operator has no configurations for model decisions.

For the inverse direction ㉑←㉒, the unidirectional operator is ←AddNamespace (work-model).

3. For the direction ㉒→㉓, the unidirectional operator is →DeleteNamespace (employees-model).

Metamodel Decisions configured for →DeleteNamespace:

- namespaceFullName = employees-model

Model Decisions configured for →DeleteNamespace: This operator has no configurations for model decisions.

For the inverse direction ㉒←㉓, the unidirectional operator is ←AddNamespace (employees-model).

4. For the direction ㉓→㉔, the unidirectional operator is →DeleteNamespace (materials-model).

Metamodel Decisions configured for →DeleteNamespace:

- namespaceFullName = materials-model

Model Decisions configured for →DeleteNamespace: This operator has no configurations for model decisions.

For the inverse direction ㉓←㉔, the unidirectional operator is ←AddNamespace (materials-model).

㉔ ⟷ Costs: ChangeModel

Since the row numbers of tasks start with 0 due to its initial CSV format, the row numbers are incremented by one, since Excel rows start with 1 (and vice versa). Therefore, this part of the orchestration does not change the metamodel. Accordingly, this part of the orchestration changes the model, as depicted in Figure 11.37 ☞433.

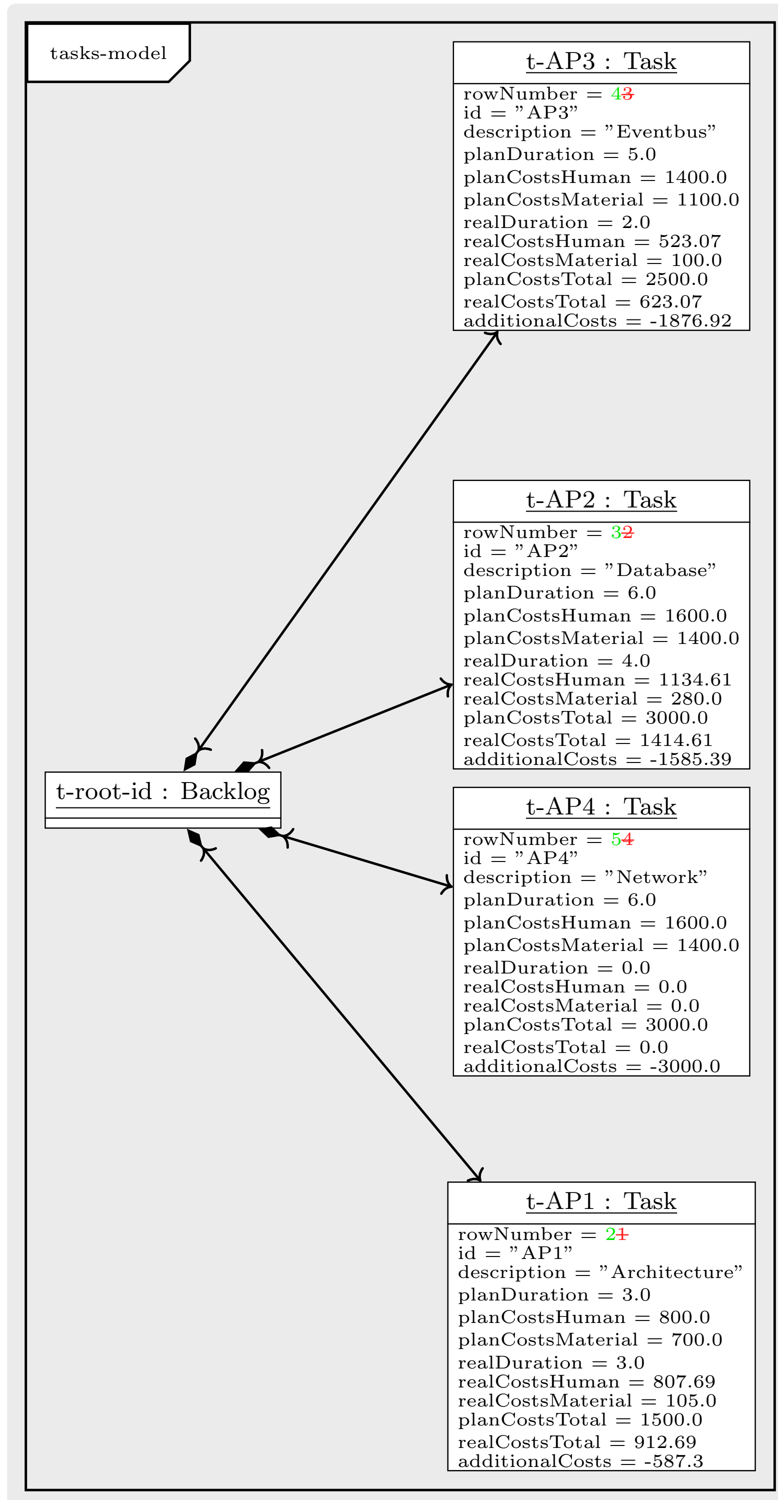


Figure 11.37: Model Changes from 24 to Costs

For the direction 24→Costs, the unidirectional operator is →ChangeModel (ChangeModel).

Metamodel Decisions are not used by →ChangeModel.

Model Decisions configured for →ChangeModel: Configurations for model decisions are realized in `de‣unioldenburg‣se‣mmi‣example‣knowledge‣KnowledgeConfig-`

uration▸17. All their configurations for model decisions are listed here:

- changeModel ( )
  Increments each row number by one.

For the inverse direction 24←Costs, the unidirectional operator is ←ChangeModel (ChangeModel).

11.4 Validation Scenarios

This section documents acceptance test cases for this application domain. Each test case is documented in its own section. The first section shows the initialization of the SU(M)M before running the test case. Additionally, the initialization shows the models for all views, before they might be changed in the particular test case. The initialization is the same for all following test cases and is documented only once.

11.4.1 Initialization by Execution

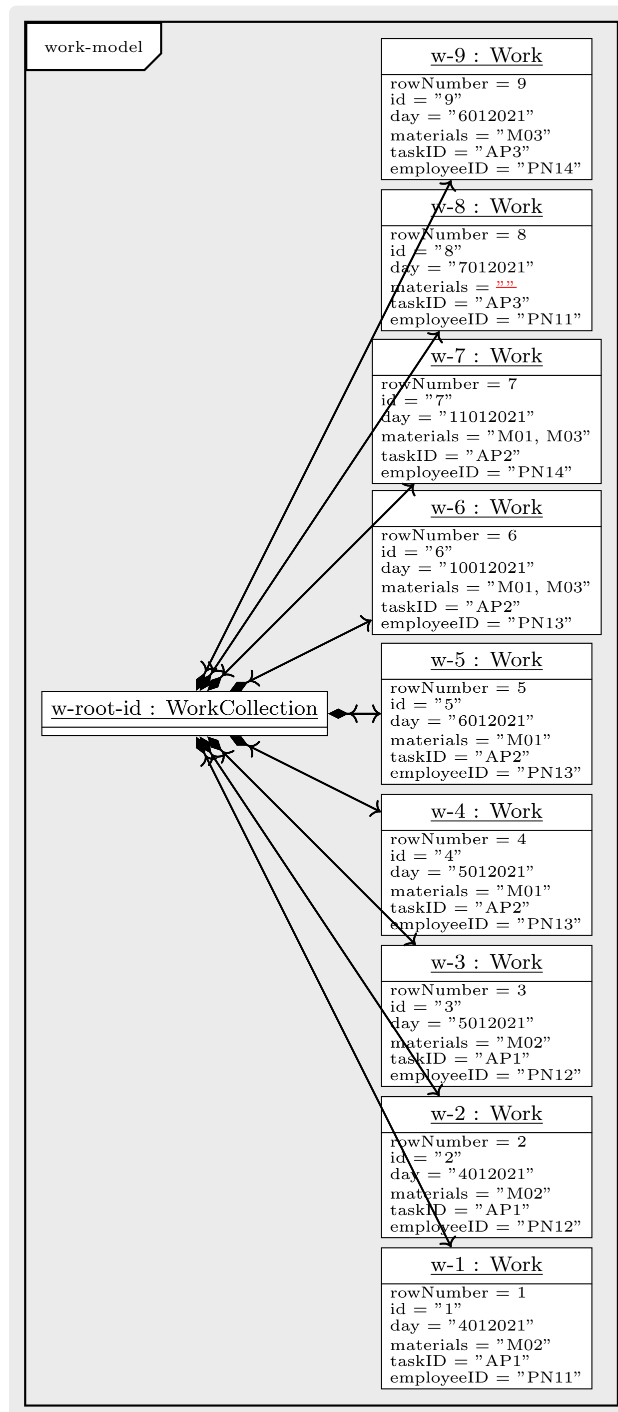
This is the description of the first execution run for the initialization: Starting with the initial data sources, the SU(M) and the new view(point)s are created, while possible inconsistencies in the data sources are fixed. The resulting models serve as starting point for the following test case scenarios.

As result after completing the synchronization, the following changes are expected:

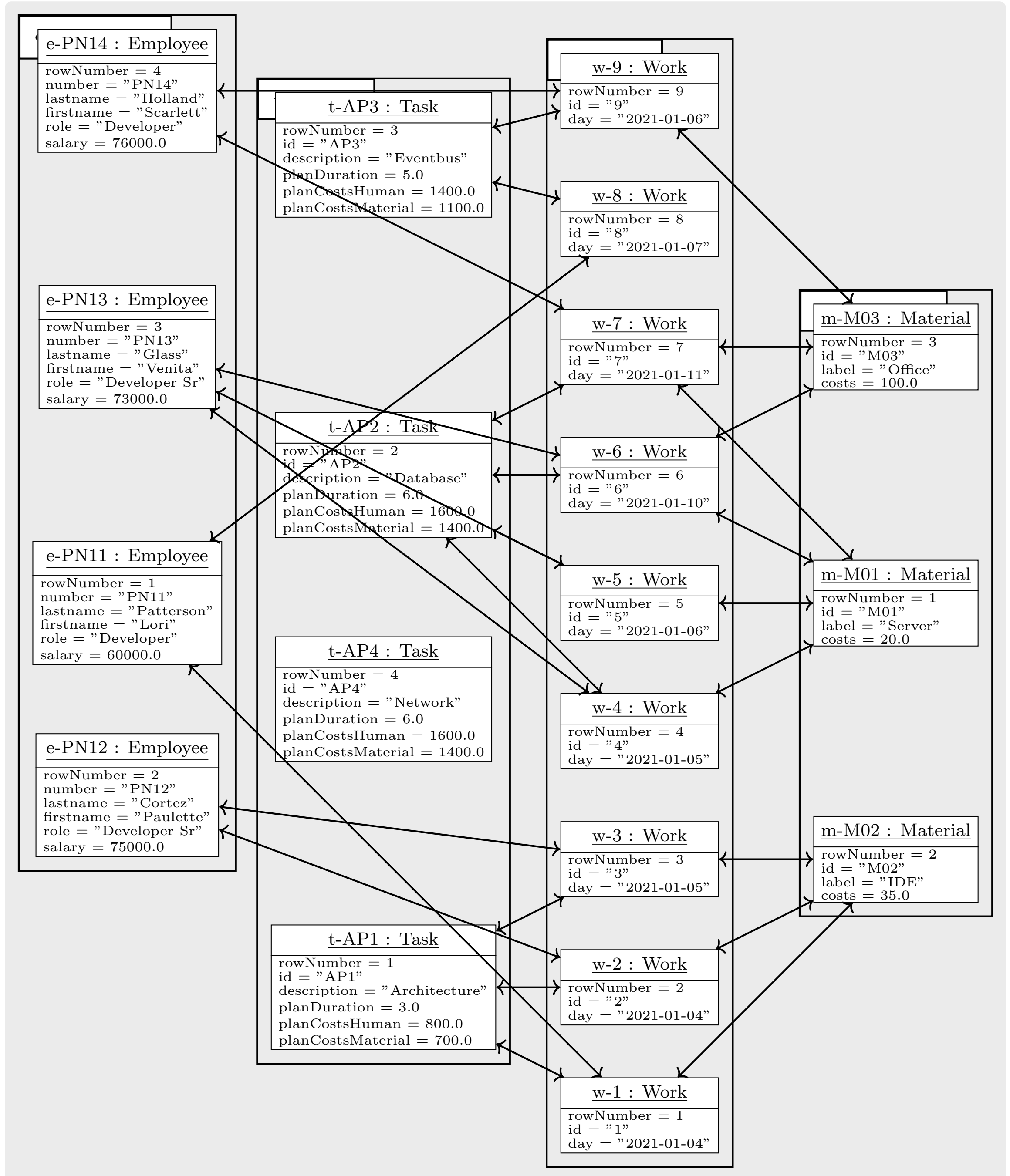
- In Work, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

#	ID	MID	APID	Day	Materials
1	1	PN11	AP1	4012021	M02
2	2	PN12	AP1	4012021	M02
3	3	PN12	AP1	5012021	M02
4	4	PN13	AP2	5012021	M01
5	5	PN13	AP2	6012021	M01
6	6	PN13	AP2	10012021	M01, M03
7	7	PN14	AP2	11012021	M01, M03
8	8	PN11	AP3	7012021	
9	9	PN14	AP3	6012021	M03

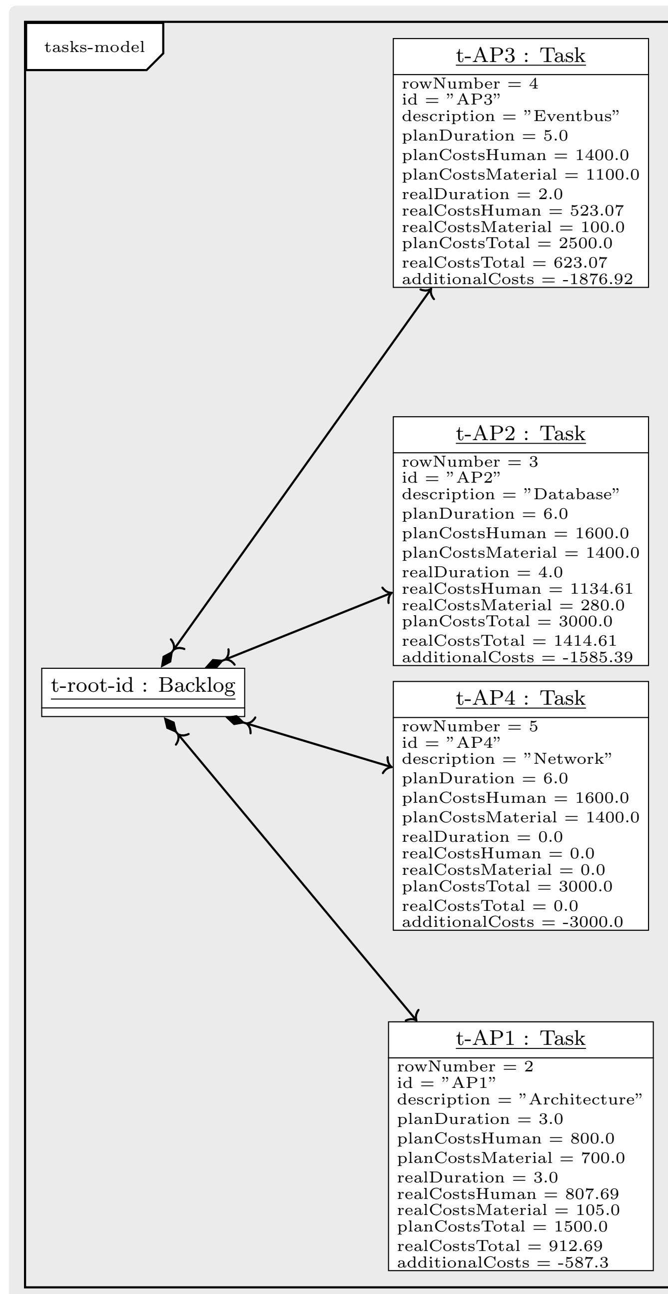
  The model with highlighted changes is represented graphically:



- In Employees, no changes are expected in the model.
- In Tasks, no changes are expected in the model.
- In Materials, no changes are expected in the model.
- Since no model existed for SUM before this execution, it is conceptually not possible to specify expected changes. Only the model after the execution can be defined:



- Since no model existed for Costs before this execution, it is conceptually not possible to specify expected changes. Only the model after the execution can be defined:



The real changes after execution of the configured operators correspond to the expected changes in the models. Therefore, the scenario is successfully fulfilled by this test case.

11.4.2 Scenario: Create new Work

Creating a new work entry of an employee for a task increases the real costs of this task, even when no materials are used, since the work of the employee adds another day of human costs of the task. This test case tests mainly C 1, C 2 and C 3.

The *User* applies the desired changes to Work by changing its internal EMF model. Therefore, the external representation is not yet updated. The model differences are represented in textual form:

```
1  w-10.createInstance()
2  w-10.changeType(null ⇒ work.Work)
3  w-10.add(rowNumber, 0, "10")
4  w-10.add(id, 0, "10")
5  w-10.add(employeeID, 0, "PN14")
6  w-10.add(taskID, 0, "AP3")
7  w-10.add(day, 0, "9012021")
8  w-10.changeNamespace(null ⇒ work-model)
9  w-root-id.add(work, 9, w-10)
10 w-10.add(container, 0, w-root-id)
```

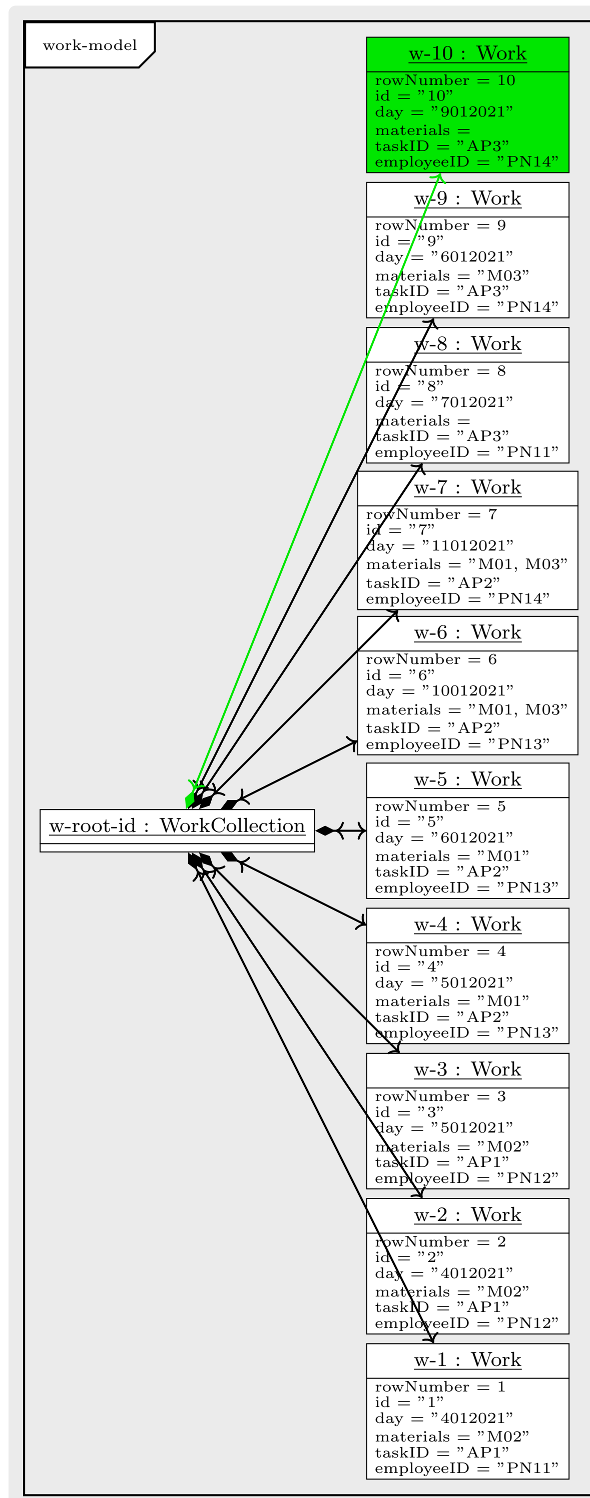
${}^{\text{User}}\Delta^{14}_{\text{Work}}$

As result after completing the synchronization, the following changes are expected:

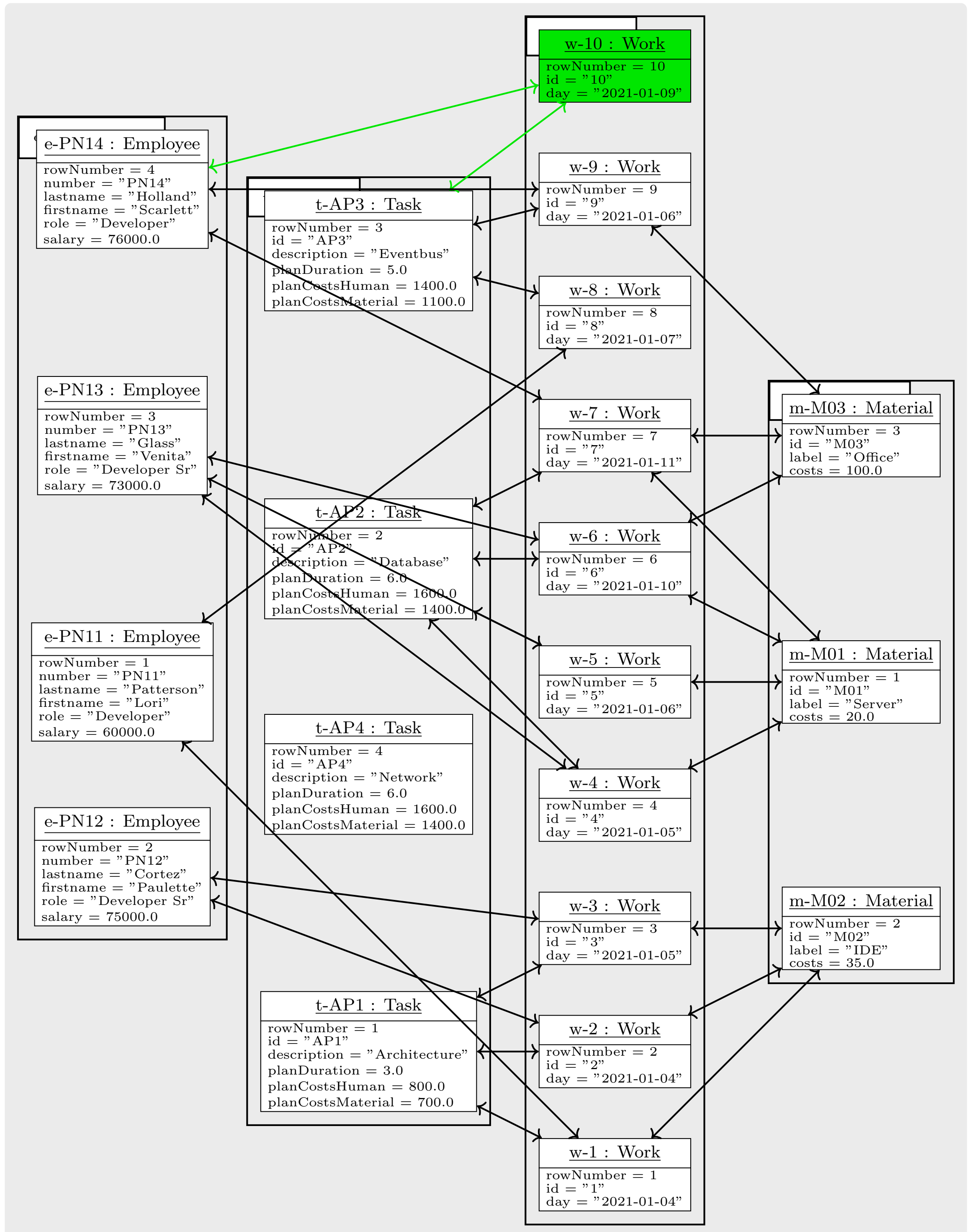
- In Work, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Work}}$). The model with highlighted changes is represented with its concrete rendering:

#	ID	MID	APID	Day	Materials
1	1	PN11	AP1	4012021	M02
2	2	PN12	AP1	4012021	M02
3	3	PN12	AP1	5012021	M02
4	4	PN13	AP2	5012021	M01
5	5	PN13	AP2	6012021	M01
6	6	PN13	AP2	10012021	M01, M03
7	7	PN14	AP2	11012021	M01, M03
8	8	PN11	AP3	7012021	
9	9	PN14	AP3	6012021	M03
10	10	PN14	AP3	9012021	

The model with highlighted changes is represented graphically:



- In Employees, no changes are expected in the model.
- In Tasks, no changes are expected in the model.
- In Materials, no changes are expected in the model.
- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically:

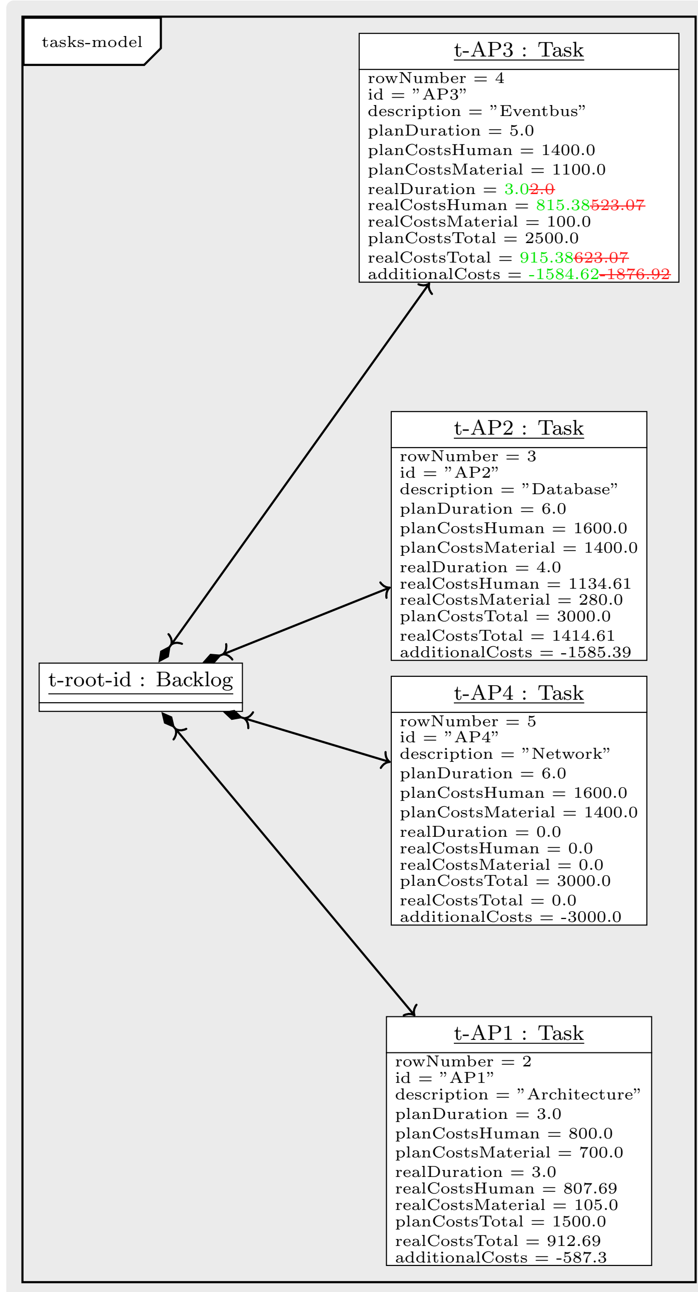


- In Costs, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

A1 fx id

	A	B	C	D	E	F	G	H	I	J	K
1	id	description	planDuratioı	planCostsHu	planCostsMa	realDuration	realCostsHur	realCostsMat	planCostsTot	realCostsTot:	additionalCosts
2	AP1	Architecture	3	800	700	3	807,69	105	1500	912,69	-587,3
3	AP2	Database	6	1600	1400	4	1134,61	280	3000	1414,61	-1585,39
4	AP3	Eventbus	5	1400	1100	3	815,38	100	2500	915,38	-1584,62
5	AP4	Network	6	1600	1400	0	0	0	3000	0	-3000
6											
7											
8											

The model with highlighted changes is represented graphically:



The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Work}}$ applied to $\boxed{\text{Work}}$ are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

11.4.3 Scenario: Change real Human Costs

Changed real-only values in the new view are automatically reverted to the previous value, since these values are recreated during the execution. Here, the real human costs of the task AP2 are manually reduced, but they are automatically recalculated to the previous value. Therefore, all views remain unchanged in the end.

The *User* applies the desired changes to Costs by changing its internal EMF model. Therefore, the external representation is not yet updated. The model differences are represented in textual form:

```
1 t-AP2.remove(realCostsHuman, 0, "1134.61")
2 t-AP2.add(realCostsHuman, 0, "800.0")
```

$^{\text{User}}\Delta^{14}_{\text{Costs}}$

As result after completing the synchronization, the following changes are expected:

- In Work, no changes are expected in the model.
- In Employees, no changes are expected in the model.
- In Tasks, no changes are expected in the model.
- In Materials, no changes are expected in the model.
- In SUM, no changes are expected in the model.
- In Costs, no changes are expected in the model (including the user changes $^{\text{User}}\Delta^{14}_{\text{Costs}}$, which are reverted due to the viewpoint definition).

The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes $^{\text{User}}\Delta^{14}_{\text{Costs}}$ applied to Costs are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

11.4.4 Scenario: Delete existing Work

Deleting an existing work entry of an employee for a task decreases the real costs of this task in general. Here, not only the real costs for employees are reduced for one task, but also its real costs for materials, since the removed work entry used one material. This test case tests mainly C 1, C 2 and C 3.

The *User* applies the desired changes to Work by changing its internal EMF model. Therefore, the external representation is not yet updated. The model differences are represented in textual form:

```
1  w-root-id.remove(work, 8, w-9)
2  w-9.changeNamespace(work-model ⇒ null)
3  w-9.remove(container, 0, w-root-id)
4  w-9.remove(id, 0, "9")
5  w-9.remove(rowNumber, 0, "9")
6  w-9.remove(materials, 0, "M03")
7  w-9.remove(taskID, 0, "AP3")
8  w-9.remove(employeeID, 0, "PN14")
9  w-9.remove(day, 0, "6012021")
10 w-9.changeType(work.Work ⇒ null)
11 w-9.deleteInstance()
```

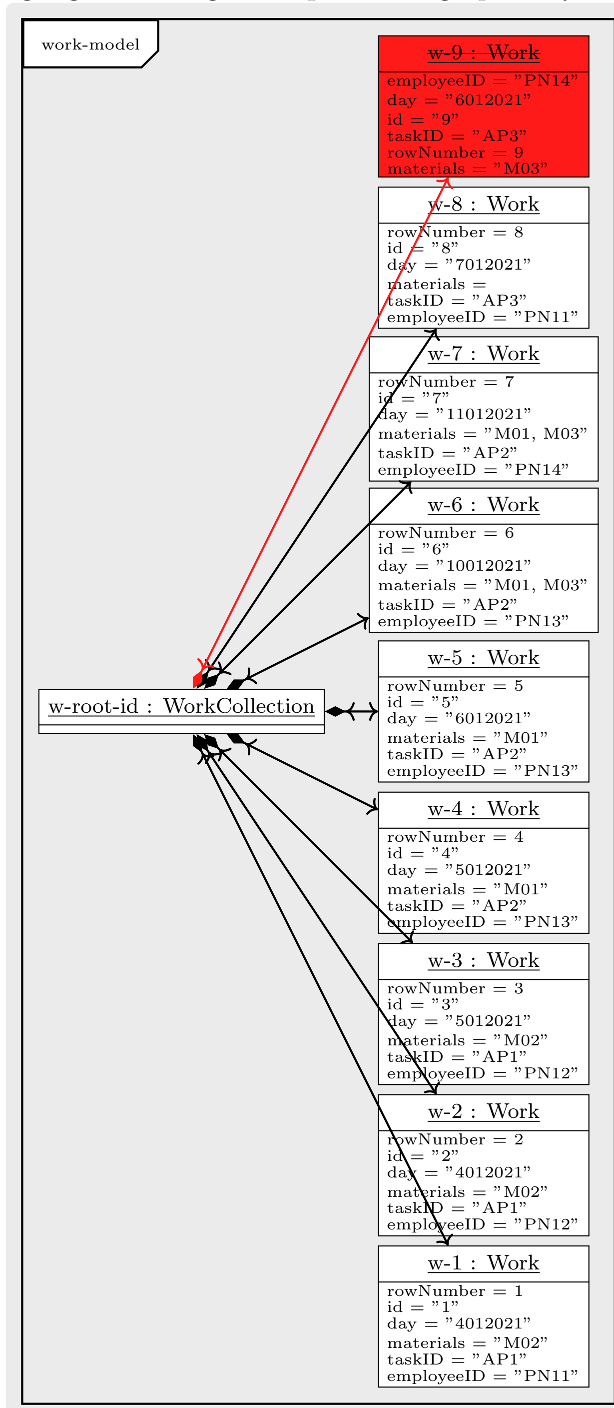
$^{\text{User}}\Delta^{14}_{\text{Work}}$

As result after completing the synchronization, the following changes are expected:

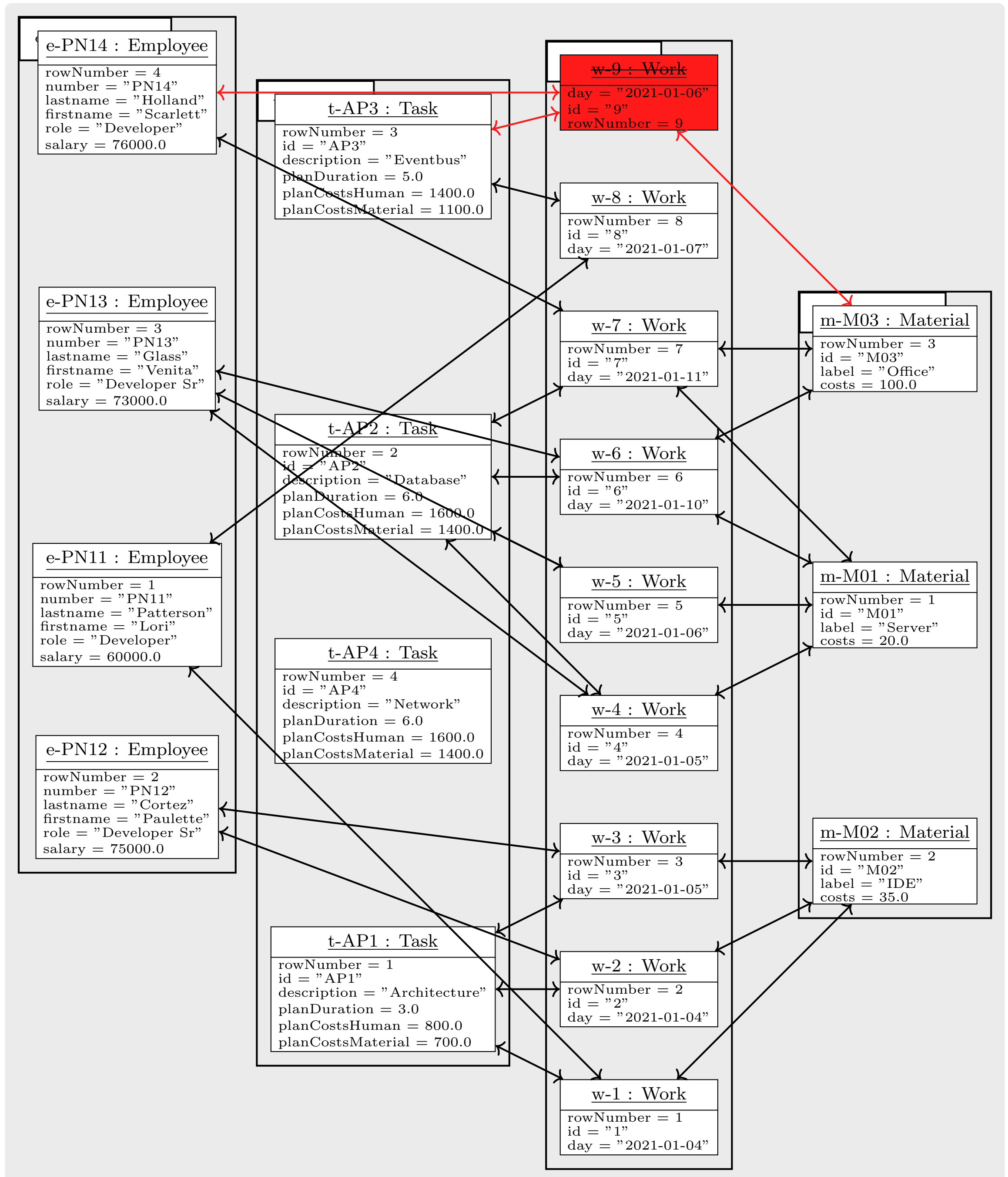
- In Work, some changes are expected in the model (including the user changes $^{\text{User}}\Delta^{14}_{\text{Work}}$). The model with highlighted changes is represented with its concrete rendering:

#	ID	MID	APID	Day	Materials
1	1	PN11	AP1	4012021	M02
2	2	PN12	AP1	4012021	M02
3	3	PN12	AP1	5012021	M02
4	4	PN13	AP2	5012021	M01
5	5	PN13	AP2	6012021	M01
6	6	PN13	AP2	10012021	M01, M03
7	7	PN14	AP2	11012021	M01, M03
8	8	PN11	AP3	7012021	

The model with highlighted changes is represented graphically:



- In [Employees], no changes are expected in the model.
- In [Tasks], no changes are expected in the model.
- In [Materials], no changes are expected in the model.
- In [SUM], some changes are expected in the model. The model with highlighted changes is represented graphically:

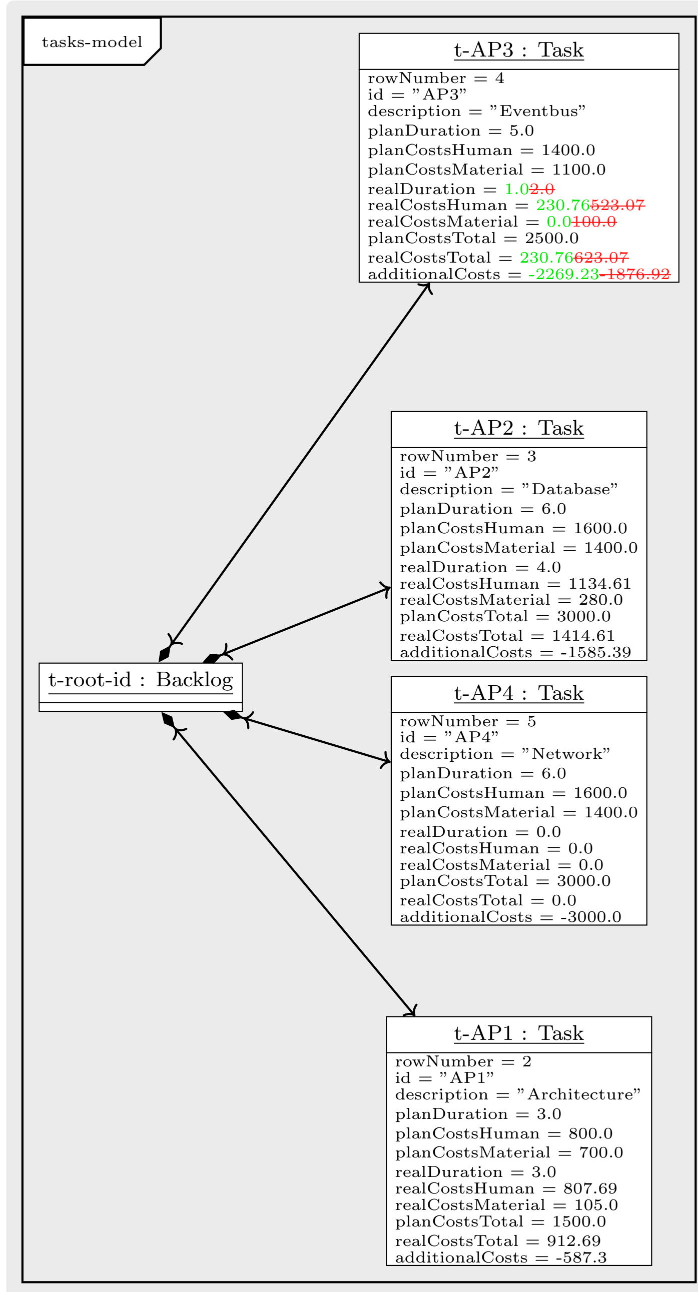


- In [Costs], some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

A1 | fx | id

	A	B	C	D	E	F	G	H	I	J	K
1	id	description	planDuratioi	planCostsHu	planCostsMa	realDuration	realCostsHur	realCostsMat	planCostsTot	realCostsTot	additionalCosts
2	AP1	Architecture	3	800	700	3	807,69	105	1500	912,69	-587,3
3	AP2	Database	6	1600	1400	4	1134,61	280	3000	1414,61	-1585,39
4	AP3	Eventbus	5	1400	1100	1	230,76	0	2500	230,76	-2269,23
5	AP4	Network	6	1600	1400	0	0	0	3000	0	-3000
6											
7											
8											

The model with highlighted changes is represented graphically:



The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Work}}$ applied to $\boxed{\text{Work}}$ are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

11.4.5 Scenario: Renamed Task in Costs View

This test case shows, that changing information which is directly contained in the SUM is possible in new views. Here, the description of the task AP4 is changed in the new view and this renaming is propagated into the data source for tasks. This test case tests mainly C 2.

The *User* applies the desired changes to Costs by changing its internal EMF model. Therefore, the external representation is not yet updated. The model differences are represented in textual form:

```
1  t-AP4.remove(description, 0, "Network")
2  t-AP4.add(description, 0, "Communication")
```

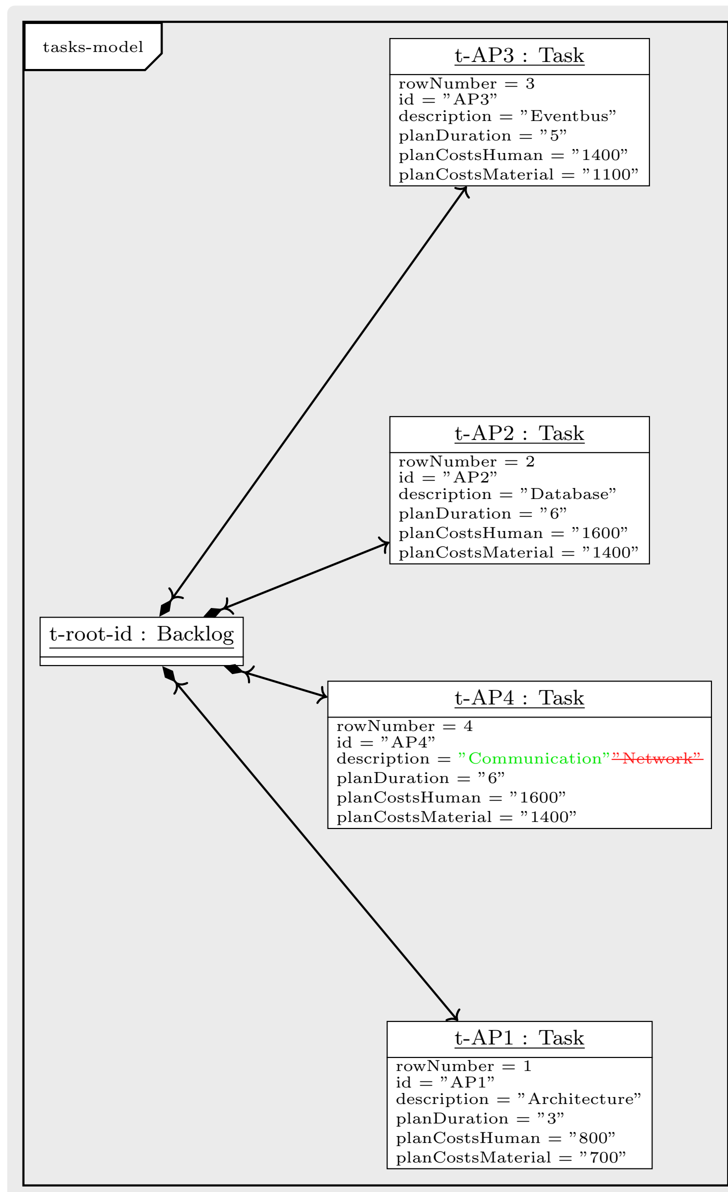
User $\Delta^{14}_{\text{Costs}}$

As result after completing the synchronization, the following changes are expected:

- In Work, no changes are expected in the model.
- In Employees, no changes are expected in the model.
- In Tasks, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

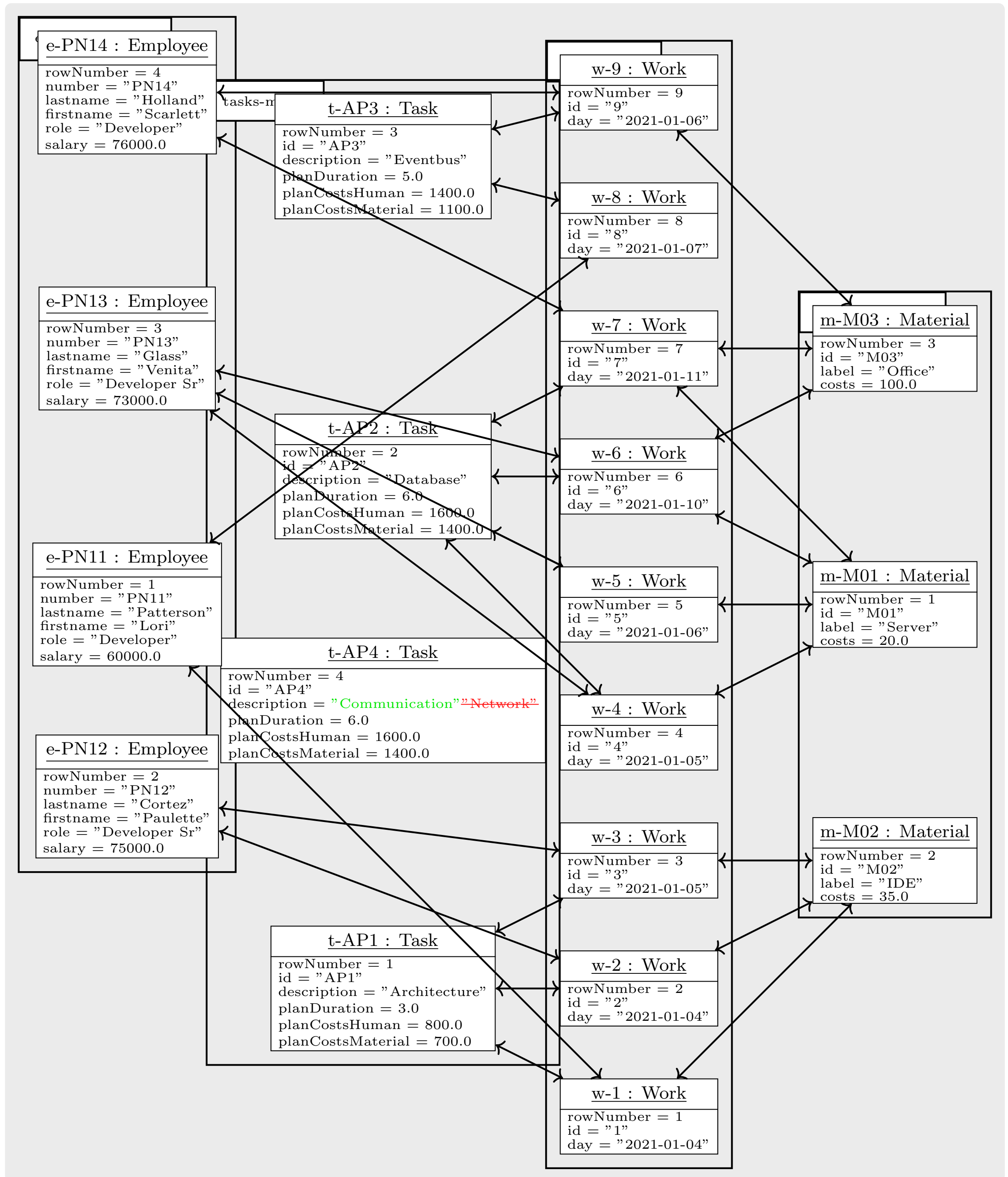
#	ID	Description	Plan Duration	Plan People Costs	Plan Material Costs
1	AP1	Architecture	3	800	700
2	AP2	Database	6	1600	1400
3	AP3	Eventbus	5	1400	1100
4	AP4	Communication	6	1600	1400

The model with highlighted changes is represented graphically:



- In Materials, no changes are expected in the model.

- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically:

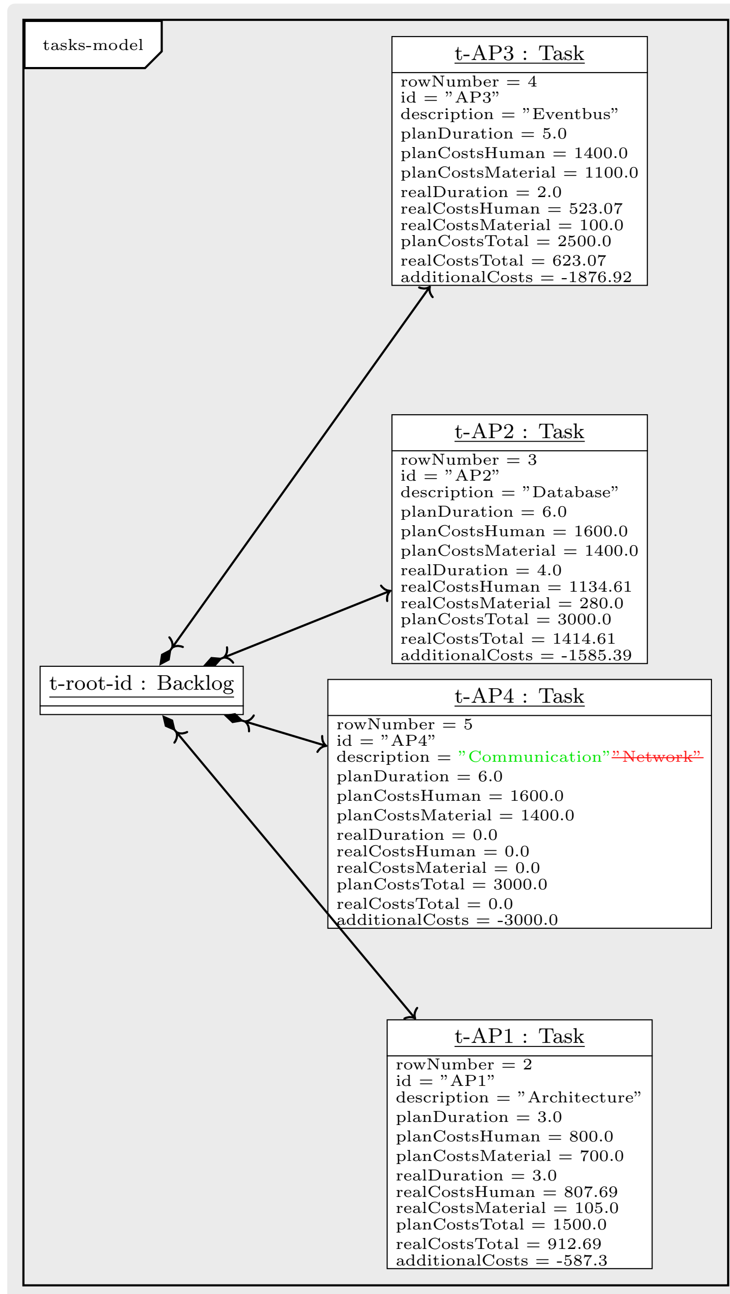


- In Costs, some changes are expected in the model (including the user changes $^{\text{User}}\Delta^{14}_{\text{Costs}}$). The model with highlighted changes is represented with its concrete rendering:

A1 | id

	A	B	C	D	E	F	G	H	I	J	K
1	id	description	planDuratio	planCostsHu	planCostsMa	realDuration	realCostsHur	realCostsMat	planCostsTot	realCostsTot	additionalCosts
2	AP1	Architecture	3	800	700	3	807,69	105	1500	912,69	-587,3
3	AP2	Database	6	1600	1400	4	1134,61	280	3000	1414,61	-1585,39
4	AP3	Eventbus	5	1400	1100	2	523,07	100	2500	623,07	-1876,92
5	AP4	Communicat	6	1600	1400	0	0	0	3000	0	-3000
6											
7											
8											

The model with highlighted changes is represented graphically:



The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Costs}}$ applied to Costs are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

11.4.6 Scenario: Change Salary

Changing the salary of employees leads to updated real costs for the tasks they worked on. Here, the salary of Lori Patterson is increased, which leads to increased costs for the two tasks she worked on. This test case tests mainly C 1, C 2 and C 3.

The *User* applies the desired changes to Employees by changing its internal EMF model. Therefore, the external representation is not yet updated. The model differences are represented in textual form:

```
1  e-PN11.remove(salary, 0, "60000")
2  e-PN11.add(salary, 0, "80000")
```

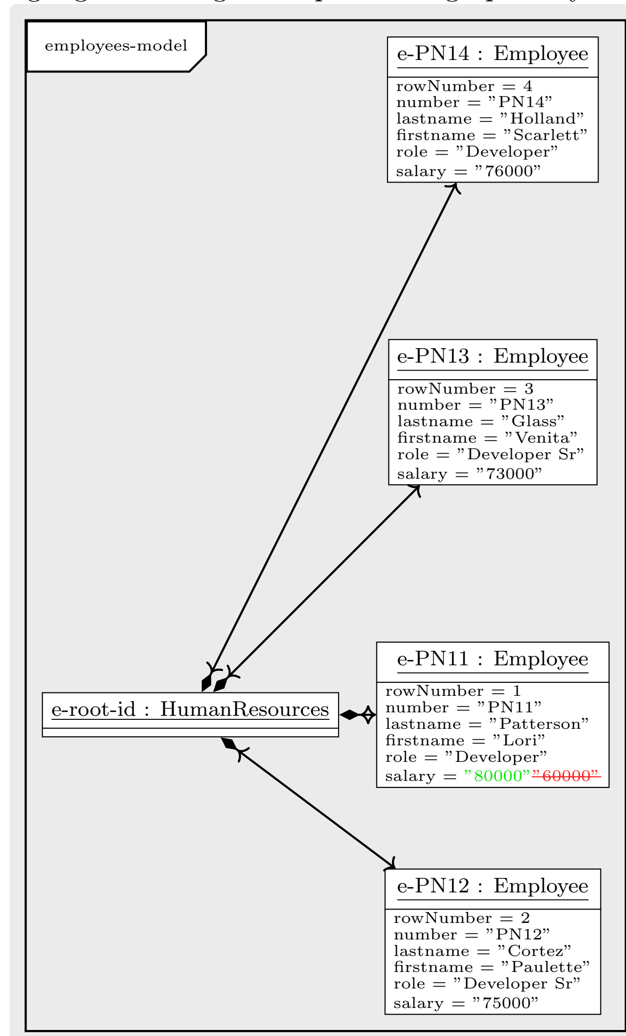
${}^{\text{User}}\Delta^{14}_{\text{Employees}}$

As result after completing the synchronization, the following changes are expected:

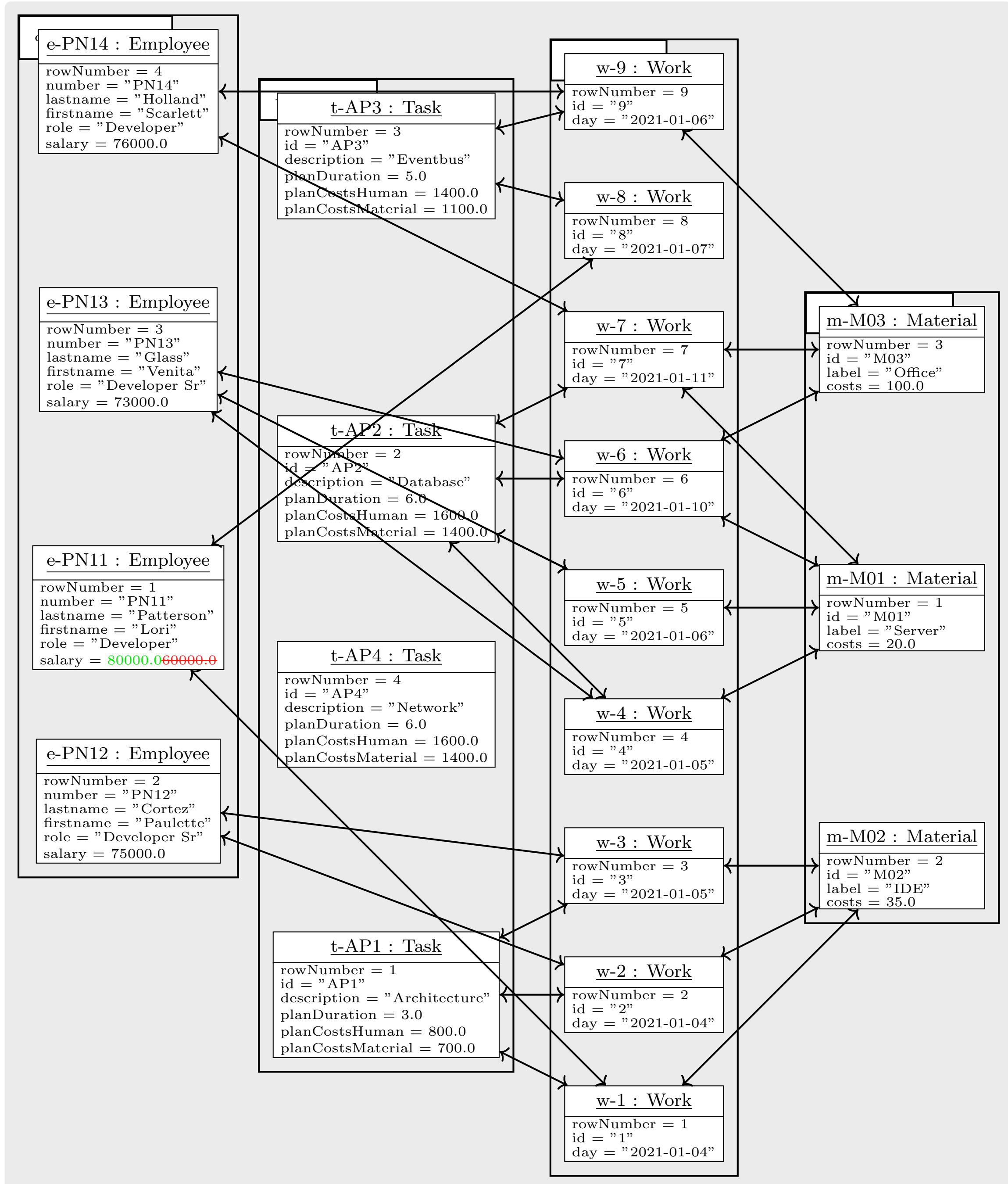
- In Work, no changes are expected in the model.
- In Employees, some changes are expected in the model (including the user changes ${}^{\text{User}}\Delta^{14}_{\text{Employees}}$). The model with highlighted changes is represented with its concrete rendering:

#	Personnel Number	Lastname	Firstname	Role	Salary
1	PN11	Patterson	Lori	Developer	80000
2	PN12	Cortez	Paulette	Developer Sr	75000
3	PN13	Glass	Venita	Developer Sr	73000
4	PN14	Holland	Scarlett	Developer	76000

The model with highlighted changes is represented graphically:



- In Tasks, no changes are expected in the model.
- In Materials, no changes are expected in the model.
- In SUM, some changes are expected in the model. The model with highlighted changes is represented graphically:

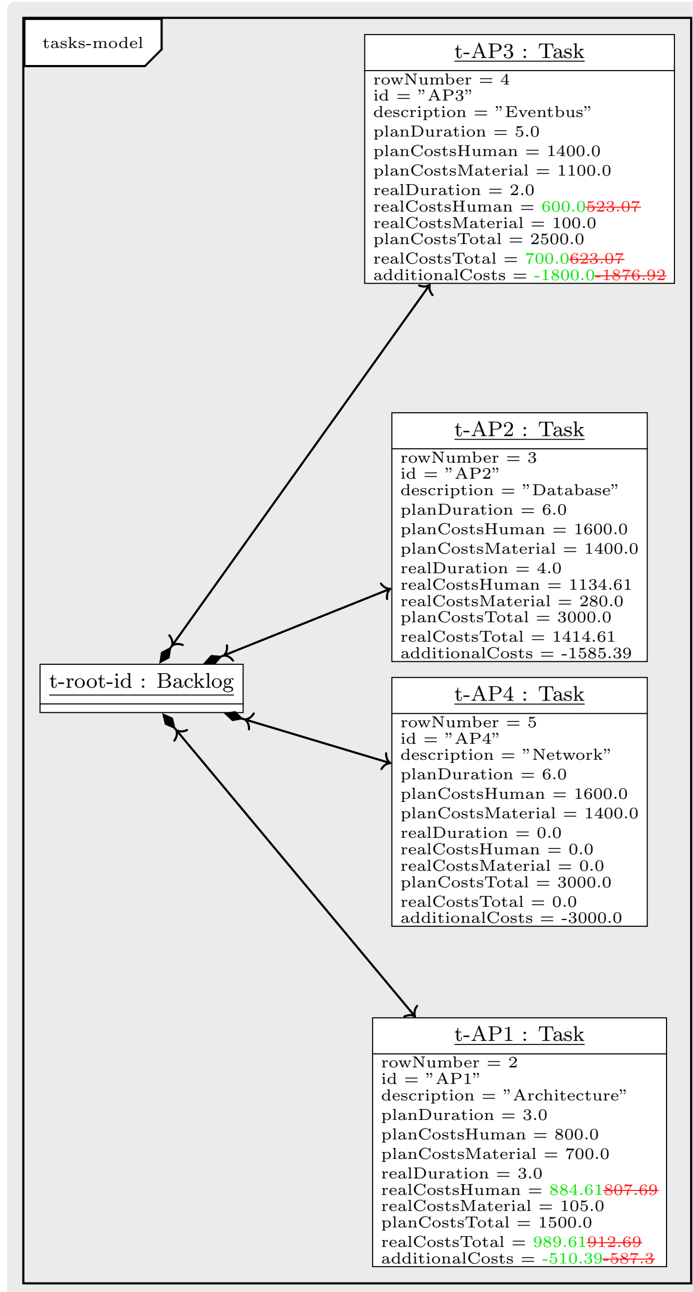


- In Costs, some changes are expected in the model. The model with highlighted changes is represented with its concrete rendering:

A1 | id

	A	B	C	D	E	F	G	H	I	J	K
1	id	description	planDuratioı	planCostsHu	planCostsMa	realDuration	realCostsHur	realCostsMat	planCostsTot	realCostsTot	additionalCosts
2	AP1	Architecture	3	800	700	3	884,61	105	1500	989,61	-510,39
3	AP2	Database	6	1600	1400	4	1134,61	280	3000	1414,61	-1585,39
4	AP3	Eventbus	5	1400	1100	2	600	100	2500	700	-1800
5	AP4	Network	6	1600	1400	0	0	0	3000	0	-3000
6											
7											
8											

The model with highlighted changes is represented graphically:



The real changes after execution of the configured operators correspond to the expected changes in the models. In particular, all user changes ${}^{\text{User}}\Delta^{14}_{\text{Employees}}$ applied to Employees are propagated into all related views. Therefore, the scenario is successfully fulfilled by this test case.

11.5 Summary: Contributions

Consistency Goals for Integration

Since knowledge management focuses more on the integration of existing information in order to provide derived information, the consistency goals guide the possibilities for integration here. All consistency goals and their rules of this application are listed below:

Collected Consistency Goals and their Rules

C 1 Each work entry refers to its employee.

C 2 Each work entry refers to a task.

C 3 Each work entry refers to all used materials.

The consistency goals C 1, C 2 and C 3 target two data sources and therefore represent *inter-model consistency* issues.

These consistency goals and their consistency rules are successfully tested by test cases documented in Section 11.4 ☞ 434. The mapping of test cases and their explicitly targeted consistency goals and rules are summarized in Table 11.10: The first column lists all consistency goals and their consistency rules. The second refers to the test cases which test explicitly the consistency goal or their consistency rule inside the same row in the first column.

Table 11.10: Mapping of Consistency Goals and their Consistency Rules tested in Sections for Knowledge Integration

Consistency	**Test Cases**
C 1	11.4.2, 11.4.4, 11.4.6
C 2	11.4.2, 11.4.4, 11.4.5, 11.4.6
C 3	11.4.2, 11.4.4, 11.4.6

Table 11.10 shows, that all consistency goals and their consistency rules are successfully tested.

successful Test Cases for ensuring Consistency

The documented test cases reflect some typical change scenarios in knowledge management and demonstrate, that MoConseMI is able to keep all views up-to-date: Three test cases explicitly show, that the derived values for real costs of tasks are updated in the new Costs view, i. e. Section 11.4.2 ☞ 437 adds another work entry leading to increased real costs for employees, Section 11.4.4 ☞ 442 deletes an existing work entry leading to decreased real costs for employees and materials, and Section 11.4.6 ☞ 449 increases the human costs more indirectly by increasing the salary of an employee, leading to increased costs for work for tasks. Section 11.4.3 ☞ 442 demonstrates, that changes of derived read-only knowledge are automatically reverted, while Section 11.4.5 ☞ 446 demonstrates, that information which directly stems from the SUM respectively data sources can be modified in new views. These test cases show, that MoConseMI supports knowledge management with keeping knowledge up-to-date. In general, these test cases show, that MoConseMI ensures inter-model consistency.

The details of the test cases are summarized in Table 11.11 ☞ 454: Each scenario is represented by one row. The “Source” column indicates the view in the orchestration, at which the user applied the wanted changes. The “Kind” column indicates, if the user changed the external representation (E) or the internal EMF model (I). The “#” column contains the number of changes, made by the user. The following columns with the names of the data sources, SUM and new view contain the number of resulting changes, after finishing the synchronization. The last column “Testing” refers to the consistency goals and consistency rules, which are explicitly evaluated by the current test case.

The scenario 11.4.1 is described with more details above, but is not summarized in the table to keep it short.

Table 11.11: Summary of Test Cases for Knowledge Integration

	Description	**Source**	**Kind**	**#**	**Work**	**Employees**	**Tasks**	**Materials**	**SUM**	**Costs**	**Testing**
11.4.2	Create new Work	Work	I	10	10	0	0	0	12	8	C 1, C 2, C 3
11.4.3	Change real Human Costs	Costs	I	2	0	0	0	0	0	0	
11.4.4	Delete existing Work	Work	I	11	11	0	0	0	14	10	C 1, C 2, C 3
11.4.5	Renamed Task in Costs View	Costs	I	2	0	0	2	0	2	2	C 2
11.4.6	Change Salary	Employees	I	2	0	2	0	0	2	12	C 1, C 2, C 3

In the test case 11.4.3, the user changes are reverted due to the consistency goals and consistency rules or the definitions of new view(point)s, like read-only information. This shows an extreme strategy to ensure consistency, namely by reverting the wanted changes. The test case 11.4.5 shows one of the main benefits of the approach: The user can change a new view and the changes are propagated back into the initial data sources. For the fast and easy definition of test cases, 11.4.2, 11.4.3, 11.4.4, 11.4.5 and 11.4.6 applied the wanted user changes to the internal EMF model, not to the external representation.

possible Extensions

While this example depicts a restricted and artificial knowledge management scenario, it could be extended with additional data sources, e. g. with knowledge about projects in order to group tasks, and with additional new view(point)s, e. g. with visualizations for used materials in order to improve resource planning. Additional consistency goals could be defined and realized with additional operators, e. g. to enforce, that employees, materials and tasks are never deleted, since they are required for documenting work and managing costs of tasks. This could be realized with →ChangeModel which reverts such deletions. These extensions and some more are possible in principle, since the design of MoConseMI is not restricted regarding the number of possible data sources, new view(point)s, and consistency goals.

Contributions

Summarizing, the contributions of this application are the finding, that MoConseMI is applicable for knowledge management, since MoConseMI is able to integrate existing knowledge and to provide integrated and derived knowledge in new view(point)s. Additionally, MoConseMI supports knowledge management by keeping the integrated data sources up-to-date. In detail, this application shows, that MoConseMI can reuse non-normalized data like the concatenation of the used materials for work and is able to improve them with the help of the ⇄ChangeMultiplicity operator (Work↔01). In general, knowledge management is another example which shows, that MoConseMI works in practice, i. e. it is successfully evaluated, that MoConseMI is able to integrate existing data sources into an explicit SU(M)M, to derive a new view(point) from the SU(M)M, and to automatically keep all these views consistent to each other.

Chapter 12

Application in general

Part III☞163 discussed the design and implementation of MoConseMI made by the *platform specialist*, while this Part IV☞283 up to now presented the application of MoConseMI in several application scenarios and showed the success of MoConseMI to fulfill consistency challenges. This Chapter 12 summarizes some findings of these applications and provides some guidelines for the application of MoConseMI *in general*. The application of MoConseMI is done by *methodologists* and *users*, meaning that methodologists apply MoConseMI once for each project to configure the desired consistency rules, while users apply the "configured instance of MoConseMI" by methodologists to be often supported with automated fixes for inconsistencies. Therefore, Section 12.1 supports methodologists with general hints, how to apply MoConseMI in order to create orchestrations for projects, while Section 12.3☞462 supports users with general hints, how to apply the configurations with MoConseMI in order to ensure consistency within a particular project. While these two sections focus more on technical guidelines for the MoConseMI framework (Chapter 8☞263), Section 12.2☞458 supports methodologists with some guidelines, how to solve recurring "consistency patterns" with particular operators (Chapter 7☞241).

12.1 Process of Configuration

This section supports methodologists during their application of MoConseMI in order to automate ensuring consistency within a particular project with a one-time configuration. This configuration is done during the use case "specify consistency", as designed in Section 5.2.1☞171, while the general design of configurations is discussed in Section 6.4☞203. In this section, the general *process* for the configuration of MoConseMI for a particular project is presented first. Then the *Java API* of MoConseMI is sketched, which is used by methodologists for the technical realization and demonstrated along the ongoing example. These parts together serve as a guideline for methodologists.

Process for specifying Consistency

The *process* for specifying consistency by methodologists consists of the following four steps:

1. elicitate Consistency Goals and Consistency Rules

1. Before using the MoConseMI framework, the desired consistency in the particular project must be elicited by methodologists: Since users expect automated fixes for occurred inconsistencies, methodologists should work together with users and other stakeholders of the particular project in order to identify the desired consistency. Feldmann, Herzig et al. (2015b) provide some examples for possible consistency goals in the domain of cyber-physical systems (CPSs), including best-practice, guidelines and domain-specific standards like conversation of units and data formats or possible violations of physical laws. The consistency is documented in terms of *consistency*

goals and consistency rules, as designed in Section 2.3☞71. These consistency goals and consistency rules are used for discussions and agreement.

2. check legal Conditions

2. After collecting consistency goals and consistency rules as "requirements for consistency", the legal aspects of them must be checked, since there might be some laws or other restrictions for data processing including data integration. These legal conditions are out of the scope of this thesis in general (Section 1.3.2☞43).

3. configure Orchestration with Java API

3. The collected, documented and checked consistency goals and consistency rules are realized by methodologists with the means provided by MoConseMI: On conceptual level, existing operators are selected, configured and combined into the orchestration, as designed in Section 6.4☞203. On technical level, this is done with the Java API provided by MoConseMI and sketched in the following.

4. initialize the SU(M)M

4. After finishing the orchestration, the methodologist starts the use case "initialize SU(M)M" (Section 5.2.3☞176), which executes the orchestration once in order to create the initial SU(M)M and the new view(point)s as well as to fix inconsistencies in the initial data sources.

After completing this process, users can use MoConseMI to be supported with automated fixes for inconsistencies, as described in Section 12.3☞462. The Java API used in step 3 is directly demonstrated along the ongoing example:

Ongoing Example, Part 25: **Configuration by the Methodologist** ← List →

The methodologist elicited the desired consistency in the ongoing project together with the users and documented the consistency with consistency goals and consistency rules, as documented in Part 11☞76 of the ongoing example (step 1). Additionally, it is assumed that all legal conditions are met (step 2). Therefore, the methodologist configured the consistency goals and consistency rules for the ongoing example with the Java API, resulting in Listing 12.1 (step 3), which demonstrates the main calls of the Java API in strongly simplified and slightly renamed way.

```
1  @Override
2  protected void collectOperators() {
3      startDataSource("Requirements", new CsvAdapter(...) {...});
4      startDataSource("Java", new EmfAdapter(...));
5      combineTwoDataSources();

7      // for Consistency Goal 1
8      addReference(...);

10     startDataSource("ClassDiagram", new EmfAdapter(...));
11     combineTwoDataSources();

13     // for Consistency Goal 2
14     mergeTwoClasses(...);
15     mergeTwoAttributes(...);

17     // for Consistency Goal 3
18     addReference(...);

20     finishedSumm();
```

```
22      startNewViewPoint();

24      replaceReferenceByAttribute(...);
25      changeMultiplicity(...);

27      subSetFilter(...);
28      deleteNamespace(...);
29      deleteNamespace(...);

31      changeModel(...);
32      renameClassifier(...);
33      renameClassifier(...);

35      finishedNewViewPoint("Traceability", new ExcelAdapter(...)
      {...});

37      finishedOrchestration();
38  }
```

Listing 12.1: Orchestration for the ongoing Example with the Java API of MoConseMI

As entry point, the methodologist creates a new Java class which extends `MetaModelIntegrationEnvironment` and implements the method `collectOperators` (lines 1–2, 38) with the configurations for the orchestrations. Since there is no SU(M)M initially, but MoConseMI is bottom-up in order to reuse existing data sources (Section 5.1.1☞164), the orchestration starts with the integration of existing data sources into the SU(M)M, according to Section 6.4.3☞205: An existing data source is reused with the API call `startDataSource(...)` (lines 3, 4, 10), which specifies the adapters (Section 6.6.5☞226) according to technical spaces of the data sources (Chapter 8☞263). After an optional chain of operators, which is used for none of the three data sources in Listing 12.1☞456, the (meta)models of the last two data sources are technically combined into one (meta)model with `combineTwoDataSources()` (lines 5, 11). As detailed in Part 21☞206 of the ongoing example, operator chains are added to integrate the (meta)models on semantic level in order to fulfill the consistency goals (lines 7–8, 13–15, 17–18). The integration of data sources into the SU(M)M is finished with `finishedSUMM()` (line 20).

integrate Data Sources with `startDataSource(...)` and `combineTwoDataSources()`

Chains of operators consist of at least one selected and configured bidirectional operator, according to Section 6.4.2☞204 and written with the Java API for bidirectional operators according to Section 8.3.3☞270. Section 12.2☞458 provides some recommendations which operators could be selected in particular situations for methodologists. Operator chains are configured in the lines 8, 14–15, 18 and 24–33, hiding the configurations for all decisions for brevity here.

Operator Chains

In order to define new view(point)s according to Section 6.4.4☞209, the Java API of MoConseMI allows to start the definition of a new view(point) at any point in the orchestration with `startNewViewPoint()` (line 22). As in Listing 12.1☞456, usually the definitions of new view(point)s start at the SU(M)M in order to reuse all available concepts and information. The definition of the new view(point) with a chain of operators as detailed in Part 22☞209 of the ongoing example (lines 24–33) is finished with `finishedNewViewPoint(...)` (line 35). After integrating all data sources and defining all new view(point)s, the orchestration is finished with `finishedOrchestration()` (line 37).

define new View(point)s with `startNewViewPoint()` and `finishedNewViewPoint(...)`

These configurations with the Java API result in the orchestration with *bidirectional* operators as already shown in Figure 6.11☞209. Note, that the Java API for operators

uses the names of *unidirectional* operators in order to emphasize the bottom-up integration of data sources into the SU(M)M and the derivation of (smaller) new view(point)s from other (bigger) (meta)models like the SU(M)M. Using names of unidirectional operators improves the reading flow in these cases, while the orchestration is build up with bidirectional operators.

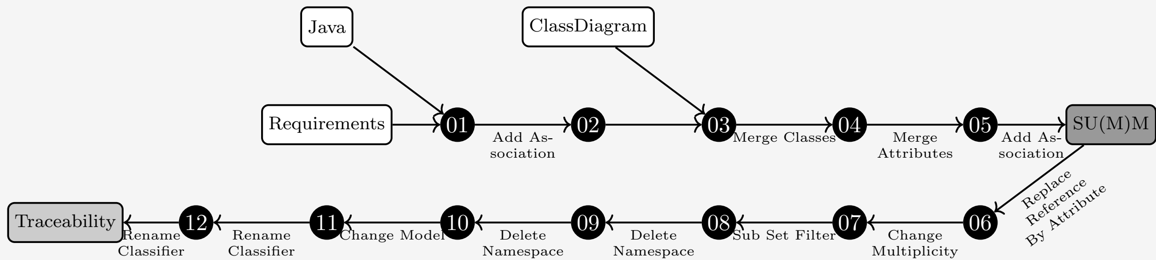


Figure 12.1: Unidirectional Operators of the whole Orchestration in the configured Direction

Figure 12.1 shows the unidirectional operators in the direction of the configuration, matching the names of operators in the Java API, while Figure 12.2 shows the inverse unidirectional operators.

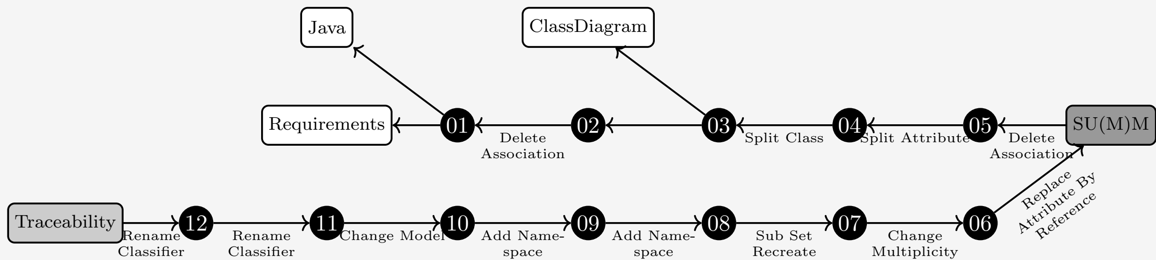


Figure 12.2: Unidirectional Operators of the whole Orchestration in the inverse Direction

Note, that the execution of operators does not care about these directions in the orchestration. For the execution, the directions import (from views to the SUM) and export (from the SUM to views) are relevant, as demonstrated in Part 23 ☞ 220 of the ongoing example, and the bidirectional operators provide the unidirectional operator depending on the current *direction of execution*. After the presented configuration of the orchestration it is executed once to initialize the SU(M)M, according to Section 6.5.4 ☞ 219 (step 4).

12.2 Recommendations for Orchestrations

This section provides some recommendations for methodologists, how to build orchestrations, and complements the technical manual for methodologists in Section 12.1 ☞ 455 with recommendations and best practices for semantic details of orchestrations. In order to integrate reused data sources into the SU(M)M, Section 12.2.1 ☞ 459 discusses, how to make relationships between depending elements in different data sources explicit. In order to keep redundant information in different data sources consistent to each other, Section 12.2.2 ☞ 460 recommends operators for reducing redundancies. Section 12.2.3 ☞ 460 discusses, how to manage remaining dependencies within the SUM. In order to support the definition of new view(point)s, Section 12.2.4 ☞ 461 provides according recommendations.

12.2.1 Explicit Links between Models

When reusing models from multiple data sources which represent parts of the same system under development, elements in these models often have some semantic relationships to elements in other models, which are not reflected by links within the models. More concrete, such relationships ...

- are always existing, but in most cases they are known only implicitly,
- are often manual documented and therefore need manually synchronization,
- are often less (or even outdated) documented due to the manual effort, or
- are sometimes hard coded in transformations for synchronization.

This section recommends to make such implicit relationships *explicit with links* during the integration into the SUM, since the explicit and persistent formulation of relationships and dependencies enables corresponding explicit links, that ...

- can be queried, managed and changed in a structured way,
- can be persisted and stored permanently,
- can be used for documentation and discussion, and
- can be reused as explicit information, e. g. for new views.

store Links between Views

Linking information of different views in form of explicit links is hard to realize, since each view contains only information of one link end, not of both link ends by definition. Again, another view could be introduced which stores the links with their connected information, as it is done by Persson, Torngren et al. (2013) with their "association views". This view would have lots of additional relations to the already existing views regarding the linked information. This leads again to the main problem of ensuring consistency between views.

implicit Links

Due to this difficulty, such links inbetween are often not made explicit, but are only remembered in the stakeholders heads resulting in follow-up problems like incomplete documentation, information asymmetries and information loss by fluctuation of developers. Bézivin (2006, p. 37) also proposes to use links to cover the challenge of fragmentation into multiple views.

Therefore, it is recommended to make implicit relationships explicit with links during the integration of data sources into the SUM. These links conform to associations in the SUMM. The strategy to define associations on metamodel level and conforming links on model level is done also in Romero, Jaén and Vallecillo (2009) with intensional (in the metamodel) and extensional (in the model) correspondences. The explicit links are not in the data sources in order to reuse them as they are. In order to enable users to manage links which require *free* interpretations of humans in contrast to *fixed* relationships which can be automated (Pfeiffer and Wasowski, 2013, p. 387, 392), without forcing users to use the SUM, dedicated new views (Section 12.2.4☞ 461) should be defined which allow to manage the links together with their connected elements stemming from different data sources.

dedicated Operators

MoConseMI provides operators for establishing explicit links: In particular, →AddAssociation can be used to introduce a new association between two depending classes in the metamodel, which enables conforming explicit links in the model. As an example, this strategy is used by the ongoing example to enable traceability between requirements and Java methods. If the elements are linked to each other via names or foreign keys, these values in attributes can be replaced by explicit links to the corresponding elements

with →REPLACEATTRIBUTEBYREFERENCE. As an example, this strategy is used for the knowledge management in Chapter 11☞387.

Traceability in general is a very prominent example, which manages explicit links and is supported by MOCONSEMI, as discussed in the following paper:

Related MoConseMI Publication

Johannes Meier and Andreas Winter (2018c): *Traceability enabled by Metamodel Integration.* In: Softwaretechnik-Trends, vol. 38(1), pp. 21–26.

This publication is cited as Meier and Winter (2018c) in this thesis.

12.2.2 Reduce Redundancies

Recommendation: reduce Redundancies in the SUM

During the integration of data sources into the SU(M)M, redundant information in overlapping data sources should be removed as much as possible, since redundancies are possible sources for inconsistencies. With other words, removing redundancies in the SUM is a strategy to keep redundant information in different views consistent to each other, since the views are automatically projected from the SUM and are up-to-date by design.

further Configurations benefit from reduced Redundancy

Additionally, reduced redundancies lead to less concepts in metamodels and to less information in models, which eases the integration of further data sources and the definition of new view(point)s, since methodologist need to know and understand less concepts and information in the (current) SU(M)M. In other words, the effort for reducing redundancies is worthwhile for all following configurations for data sources and new view(point)s. Having no redundancies directly leads to a single point-of-truth. Thomas and Nejmeh (1992) recommend to minimize redundant data as well, since redundancies require maintenance effort for their consistency.

Operators for reducing Redundancies vs Operators for ensuring Consistency directly

Keeping redundancies in the SUM (Section 12.2.3) requires additional configurations to keep them consistent to each other. Since these additional configurations are configured operators like →CHANGEMODEL again, these operators ensure consistency, e. g. in change translation-based way, while operators reduce redundancies otherwise. Since the means for both strategies are the same, i. e. configured operators, operators should be configured to reduce redundancies with the benefits discussed above. By removing redundancies, an optimal SUM as in OSM (Section 3.5.1☞124) could be reached in the best case.

dedicated Operators

MOCONSEMI provides operators for reducing redundancies: Completely redundant objects or values could be directly removed with operators like →DELETECLASS or →DELETEATTRIBUTE. The inverse unidirectional operators recreate the redundant information from the remaining information in the SUM, e. g. ←ADDATTRIBUTE calculates according default values for the “new” attribute slots. If there are objects representing partially overlapping information, e. g. classes in Java and in UML in the ongoing example with different additional information, they could be *merged* with →MERGECLASSES, since two objects representing the same element are unified into one object, while keeping all non-overlapping information of both objects. Afterwards, the merged objects might have redundant values for redundant attributes and redundant links for redundant associations, which could be merged with →MERGEATTRIBUTES and →MERGEREFERENCES. In particular the merging of attributes often occurs, when these attributes are used before to identify redundant objects in the configuration of the corresponding model decision of →MERGECLASSES.

12.2.3 Remaining Dependencies in the SUM

There are not only redundancies, but also explicit links and constraints as dependencies (Section 1.2.1☞31) within the SUM, whose consistency must be ensured as well. Addi-

tionally, there might be some reasons for keeping some redundancies, leading to possible inconsistencies, while redundancies should be minimized in general (Section 12.2.2 ☞ 460). In general, Vogel-Heuser, Fay et al. (2015, p. 66) say, that it is an open question, how much inconsistencies can remain. Therefore, this section provides some discussions for dependencies remaining in the SUM. More quality aspects of the SU(M)M are discussed in Section 13.3.3.1 ☞ 475.

Additionally to the finding, that "*[m]aintaining absolute consistency is not always possible*" (Finkelstein, Gabbay et al., 1993), there are some more reasons for keeping some dependencies within the SUM: Some dependencies might require too much effort for their automation or "*the completeness of such sets of rules can be varied, allowing for economic trade-off*" (Feldmann, Herzig et al., 2015a, p. 158). There might be consistency goals which cannot be automated at all (Section 14.2.2 ☞ 489). Additionally, iterative integration processes might manage some dependencies in early iterations and automate other dependencies in later iterations. The amount of remaining redundancies could be measured with model clones (Störrle, 2013). All remaining dependencies in the SUM might lead to inconsistencies, which must be managed explicitly by operators or manually.

Reasons for keeping Dependencies

Basing on these considerations, methodologists decide about the quality of the SU(M)M regarding internal dependencies and choose one of the following strategies to deal with each dependency:

Strategies to deal with Dependencies

1. Redundancies can be removed with configured operators (Section 12.2.2 ☞ 460).
2. All remaining dependencies can be maintained automatically with configured operators, if possible and reasonable: These operators ensure consistency directly, e. g. →ChangeModel can be configured in change translation-based way to react on occurred changes and to apply follow-up changes to fix an occurred inconsistency. But also some more advanced techniques could be used inside →ChangeModel, like automations supported by machine learning techniques using reinforcement learning (Barriga, Rutle and Heldal, 2019).
3. All dependencies without automation support must be managed manually: Maintaining consistency manually could be supported with dedicated new view(point)s (Section 12.2.4), which provide only the depending information which must be maintained by users. They could be supported by showing inconsistency warnings and some more hints (Balzer, 1991). Grundy, Hosking and Mugridge (1998) propose much more ideas for presenting, annotating and grouping inconsistencies to support users to manually fix inconsistencies, opening a wide field for user-friendly tool support for manual inconsistency management.

12.2.4 New View(point)s

New view(point)s are useful to support additional stakeholders with additional concerns (Section 1.2.3 ☞ 39), to manually maintain explicit links (Section 12.2.1 ☞ 459), or to manually manage remaining inconsistencies (Section 12.2.3 ☞ 460).

Reasons for new View(point)s

Usually, the operator chain for the configuration of the new view(point) starts at the SU(M)M, since it contains all information with minimized redundancies (Section 12.2.2 ☞ 460). If less information is required and available at other positions, the operator chain could start at these positions instead. After starting the chain of operators, non-related content should be filtered out, e. g. with →SubSetFilter, to make the current (meta)model smaller as early as possible, also suggested by König and Diskin (2017). The relevant information can be restructured and new values can be derived from existing information with various operators. Information which was required for these restructurings and calculations, but which should not be shown to users, should be filtered out again. Finally, adapters for

general Procedures in Operator Chains for new View(point)s

technical spaces can be reused (Section 8.4☞271) to support users with appropriate concrete renderings.

12.3 Process of Use

Users apply MoConseMI by using the project-specific orchestration, which is configured by methodologists with the means of MoConseMI (Section 12.1☞455). During the use case "fix inconsistencies automatically" (Section 5.2.2☞173), a user requests a view, changes this view and triggers MoConseMI to automatically propagate these changes into all other views in order to keep them consistent to the manually changed view. Usually, the views are changed by users, but the SUM or even any other intermediate model in the orchestration could be changed (Section 6.5.5☞220).

Java API for Use

In order to support users to use MoConseMI, corresponding Java API calls are provided, which are directly demonstrated along the ongoing example. This API is used also for the acceptance tests in the application examples.

Ongoing Example, Part 26: **Application by Users** ← List →

change external Concrete Rendering

Since users usually work not directly on the EMF model for the current view, but on the concrete rendering of the current view, the corresponding adapters determine the user changes, as designed in Section 6.6.5☞226. For the CSV representation of Requirements, the user gets the current requirements specification as CSV file, changes the CSV file and triggers MoConseMI with the Java API call in Listing 12.2 to execute the orchestration: Line 2 specifies the view by its name and line 3 allows to provide a user-defined description of changes for documentation, similar to a commit message. The `boolean` flag in line 3 indicates, that the identified user changes should be propagated to all views (`true`) and that are not only calculated (`false`).

```
1 framework.reloadChangedNode(
2     framework.getOrchestration().getNodeByLabel("Requirements"),
3     "description of my changes", true);
```

Listing 12.2: Trigger MoConseMI to propagate User Changes made in the concrete Rendering with CSV of the Requirements View

change internal Edapt Model

If the view to change is represented directly with EMF or if some fast test cases should be developed, user changes can be directly executed with the Edapt API (Section 8.2☞264) with the current model of the view, as demonstrated in Listing 12.3 for the test scenario presented in Part 18☞173 of the ongoing example. Line 2 specifies the view by its name and the lines 6–9 leave room to explicitly change the model of the view: Since the UUID for the class in ClassDiagram with the name "University" is known, it is retrieved directly in line 7. Line 8 is used to ensure, that the desired class is identified and shows the previous name of the class. Line 9 sets the name (encoded with the attribute "className") of the class to the new value "Institution". Line 11 allows to provide a user-defined description of changes for documentation, similar to a commit message.

```
1 framework.applyDiffToAnyNode(
2     framework.getOrchestration().getNodeByLabel("ClassDiagram"),
3     new AppyModelChanges() {
4         @Override
5         public void apply(MigrationInformation infos) {
6             // change the model directly, provided by "infos"
```

```
7               Instance cls = infos.getInstanceByUuid("cd1");
8               assertEquals("University", cls.get("className"));
9               cls.set("className", "Institution");
10          }},
11      "renamed University");
```

Listing 12.3: Change the Model of the View directly in the EDAPT Format

Both API calls `reloadChangedNode(...)` and `applyDiffToAnyNode(...)` determine the user changes and execute the execution loop. Afterwards, the execution differences ${}^{\mathrm{E}}\Delta$ are available for each view, as they are visualized in Part 18☞173 of the ongoing example.

12.4 Summary

This section demonstrated the application of the MoConseMI framework including its Java API by methodologists (Section 12.1☞455) and users (Section 12.3☞462). The process for realizing inter-model consistency by methodologists (Section 12.1☞455) is complemented with recommendations for the design of orchestrations, including to make implicit relationships explicit with links (Section 12.2.1☞459), to reduce redundancies as much as possible in order to ensure their consistency and to ease the orchestration (Section 12.2.2☞460), and to manage remaining dependencies with additional operators for automation (Section 12.2.3☞460) or with new view(point)s for manually maintenance (Section 12.2.4☞461).

Future Work: **Transaction Management**

For now, a user works on a single view and changes it with the following change propagation by MoConseMI, before another user can work on a view with following change propagation and so on, similar to the *lock-modify-unlock* principle (Altmanninger, Seidl and Wimmer, 2009). This works, but is only a first step towards a real transaction management for working with multiple users on multiple views. Atkinson, Stoll and Bostan (2009, p. 77) already propose a transactional versioning system for managing multiple views.

In particular, a transaction management should enable to revert together the user changes in one view and their follow-up changes by MoConseMI in a structured way. Here, a transaction would be the changes of a user in a view with the following execution loop (Section 6.5.3☞217).

Another extension for transaction management is to allow multiple users to change views at the same time: This research area is called "model integration" by Anjorin, Leblebici and Schürr (2016), "concurrent updates" by Diskin, Gholizadeh et al. (2016) and "bidirectional synchronization with reconciliation" by Antkiewicz and Czarnecki (2008), who provide also different realization strategies. For BX transformations specified with TGGs, Hermann, Ehrig et al. (2012) present an approach for semi-automatic conflict resolution for concurrent updates on source models and target models. The BX approach EVL+STRACE (Samimi-Dehkordi, Zamani and Kolahdouz-Rahimi, 2018) provides some support for concurrent updates. Another approach for concurrent updates is described in Xiong, Song et al. (2013), but there are still problems for non-bijective transformations. These related approaches show, that supporting concurrent updates of multiple users is important and needs to be solved also by MoConseMI as future work.

First ideas to extend MoConseMI with support for concurrent updates is to run the execution loop twice, one after the other, and to merge the calculated execution differences in the views. In case of conflicts, they must be solved manually by the users of the particular

views. For merging of models, there is also some support, including by Dam, Egyed et al. (2016) with some consistency support, by Sabetzadeh and Easterbrook (2006) with support for incompleteness and inconsistencies, and by Sabetzadeh and Easterbrook (2003) with traceability.

Part V

Achievements

This part evaluates MoConseMI regarding the fulfillment of requirements and discusses properties of operators, characteristics of orchestrations and some more conceptual aspects of MoConseMI. Additionally, this part summarizes the contributions of this thesis, discusses preconditions and limitations of MoConseMI, points to future work and concludes the thesis with a summary.

Chapter 13

Evaluation

Part IV☞283 applied MoConseMI to several application examples and demonstrated, that MoConseMI is able to ensure inter-model consistency in concrete applications. Chapter 12☞455 generalized these concrete applications and gave guidelines for successful applications of MoConseMI in general. In addition to the concrete applications before, this Chapter 13 evaluates further aspects of the design of MoConseMI in order to point to resulting characteristics of MoConseMI.

Outline

First of all, Section 13.1 validates that all requirements are fulfilled by MoConseMI. The following sections evaluate and discuss further aspects, which are grouped regarding the properties of operators (Section 13.2☞469), the characteristics of orchestrations (Section 13.3☞473) and general aspects of MoConseMI (Section 13.4☞478).

13.1 Fulfillment of Requirements

Section 5.3☞179 already discussed, that the main decisions for the design of MoConseMI fulfill all functional requirements. Therefore, this section summarizes and concretizes those arguments here, while focusing on practical arguments of the application of MoConseMI. Additionally, the technical requirements of Section 4.2☞157 are evaluated at the end of this section.

functional Requirements

Regarding *functional* requirements, Requirement R 1 (Model Consistency)☞154 is fulfilled by MoConseMI by using model synchronization techniques for change propagation (Section 5.1.4☞168), which is demonstrated with the test cases for consistency in the application examples in Part IV☞283.

Requirement R 1.1 (Generic Metamodels)☞154 is fulfilled by MoConseMI with the metamodel decisions of reusable operators, which allow methodologists to adapt the generic operators to the project-specific metamodels. The application examples in Part IV☞283 use different metamodels for different data sources in different projects.

Requirement R 1.2 (Generic Consistency Goals)☞155 is fulfilled by MoConseMI with the model decisions of reusable operators, which allow methodologists to adapt the model transformations of the generic operators according to the project-specific consistency goals and consistency rules. The application examples in Part IV☞283 fulfill different consistency goals and consistency rules in different projects.

Requirement R 2 (Reuse existing Artifacts)☞155 is fulfilled by MoConseMI's bottom-up design with existing artifacts as starting point (Section 5.1.1☞164) for their subsequent integration into the SU(M)M (Section 6.4.3☞205). The models and metamodels of the reused artifacts are addressed also by the following sub-requirements. The application examples in Part IV☞283 reuse different artifacts as data sources in different projects.

Requirement R 2.1 (Reuse existing Metamodels)☞155 is fulfilled by MoConseMI's bottom-up design with existing metamodels of the reused artifacts as starting point (Sec-

tion 5.1.1☞164) for their subsequent integration into the SU(M)M (Section 6.4.3☞205). The application examples in Part IV☞283 reuse different metamodels of data sources in different projects.

Requirement R 2.2 (Reuse existing Models)☞156 is fulfilled by MoConseMI's bottom-up design with existing models of the reused artifacts as starting point (Section 5.1.1☞164) for their subsequent integration into the SU(M)M (Section 6.4.3☞205). The application examples in Part IV☞283 reuse different models of data sources in different projects.

Requirement R 2.3 (Fix existing Models)☞156 is fulfilled by MoConseMI by reusing models according to Requirement R 2.2 (Reuse existing Models)☞156 and executing the execution loop with the initially reused (meta)models in order to fix initial inconsistencies in the reused models (Section 6.5.4☞219). With other words, the reused models are treated as possibly inconsistent models after users changed them. In contrast to the ongoing change propagation triggered by user changes (Section 6.5.5☞220), during the first execution of the execution loop, there are no historic data including empty history maps, which allows to fix only a subset of those inconsistencies which would be fixed in subsequent executions. Summarizing, MoConseMI is able to fix inconsistencies in initial models, if there is information to find these inconsistencies. Otherwise, users or methodologists have to fix those initial inconsistencies manually which require human knowledge.

Requirement R 3 (Define new View(point)s)☞156 is fulfilled by MoConseMI by using operators not only for integrating data sources into the SU(M)M, but also for defining new view(point)s (Section 6.4.4☞209). The application examples in Part IV☞283 define different new view(point)s in different projects.

Requirement R 3.1 (New Views reuse whole System Description)☞157 is fulfilled by MoConseMI by using the SUM, which contains all information of all data sources in integrated way, as starting point for the operator chain for the new view, which allows to reuse all information of the SUM for new views. The application examples in Part IV☞283 define different new views, which reuse information stemming from different data sources within the same project.

Requirement R 3.2 (New Viewpoints with arbitrary Metamodels)☞157 is fulfilled by MoConseMI by using operators, which jointly change metamodels and models, for the definition of new viewpoints. The application examples in Part IV☞283 define different new viewpoints, whose metamodels are different to the SUMM of the same project.

Requirement R 3.3 (Editable new Views)☞157 is fulfilled by MoConseMI by using the same operators for defining the transformations between all partial views and the SUM and by executing these operators in the same way (Section 6.5.3☞217), which allows to modify models of data sources as well as of new views in the same way. In particular, the bidirectional operators support both directions, from views (including new views) to the SUM and from the SUM to views. The application examples in Part IV☞283 demonstrate different test cases for user changes in new views in different projects.

technical Requirements

Regarding *technical* requirements, Requirement R 4 (Technical Spaces)☞158 is fulfilled by MoConseMI with the concept of adapters, which bridge views realized in different technical spaces to ECore as the technical space used within MoConseMI (Section 6.6.5☞226). Section 8.4☞271 presents several concrete adapters, which are successfully used by application examples in Part IV☞283. In particular, these applications contain some test cases, where users change views not by changing the EMF models, but by changing their representations in the technical spaces.

Requirement R 5 (Reusable Library)☞158 is fulfilled by MoConseMI, since MoConseMI is implemented as reusable Maven library without forcing users and methodologists to use a GUI (Section 8.1☞263). The reusability of the MoConseMI framework is shown by all application examples in Part IV☞283, since they are Maven projects which reuse MoConseMI as dependency.

Summarizing, all functional requirements are fulfilled by the design of MoConseMI, as

discussed in Chapter 5[☞163] and Chapter 6[☞185]. Additionally, the application examples in Part IV[☞283] demonstrate, that all functional requirements are fulfilled also in practice. The technical requirement are all fulfilled by the implementation of the MoConseMI framework in Chapter 8[☞263], supported by the design of the MoConseMI approach. Therefore, MoConseMI *fulfills all requirements* of Chapter 4[☞153].

13.2 Properties of Operators

Since MoConseMI is operator-based and the operators of Section 7.3[☞243] are an important part of the solution, this section discusses several properties of operators.

13.2.1 Formal Properties

no Proofs of formal Properties possible in general

The research for BX established several formal properties for BX transformations, which could be used to evaluate bidirectional operators of MoConseMI. According to *correctness*, which can be verified only, if the consistency is explicitly formalized (Anjorin, Buchmann et al., 2020, p. 661), most of these formal properties cannot be successfully proved for MoConseMI, since the consistency goals and consistency rules describe the desired consistency in natural language and without formalizations. Additionally, the behavior of unidirectional operators depends on the configurations for model decisions by methodologists as well, which makes general proves hard.

Nevertheless, some statements can be given for some formal properties of BX approaches. These properties are selected and taken from BX research (Anjorin, Buchmann et al., 2020; Hidaka, Tisi et al., 2016; Stevens, 2010):

- As already mentioned, *correctness* cannot be proved, since consistency is not formalized in MoConseMI.
- *Hippocraticness* requires not to change models which are already consistent to each other: Hippocraticness is ensured by MoConseMI after the execution loop stopped, since no branch differences occur anymore, i. e. the models are consistent to each other according to the configured orchestration. Note, that the termination of the execution loop cannot be guaranteed, as discussed in Section 14.3.1.1[☞491].
- Orchestrations in MoConseMI do not fulfill *least change* (Section 3.2.4[☞106]), since they fulfill *least surprise* by design according to the project-specific configurations of methodologists and according to the expectations of users for consistency.
- MoConseMI is *Turing-complete* (Hidaka, Tisi et al., 2016, p. 911), since →ChangeModel allows to transform models in arbitrary way with the Turing-complete Java programming language.
- Section 14.2.2[☞489] discusses, that consistency goals and consistency rules might be *non-deterministic* in MoConseMI under some circumstances.

MoConseMI trades formal Guarantees for Expressiveness

Summarizing, the design of MoConseMI trades formal guarantees for expressiveness. This trade-off is a general one in BX research: "*In general, the more expressive power the transformation has, the less guarantees on well-behavedness the system can provide. In the extreme case, Turing-complete transformation languages can express complex forward transformations, making it difficult to define a backward transformation that, when combined with the forward one, will form a well-behaved system*" (Hidaka, Tisi et al., 2016, p. 916).

13.2.2 Completeness of Operators

The list of operators in Section 7.3☞243 is *complete* in the sense, that the provided operators are sufficient to transform each (meta)model into each other (meta)model. *Operators are complete to realize any (Meta)Model Transformation* which is proved by applying the following operators:

1. Use the operators →AddNamespace, →AddClass, →AddAttribute and →AddAssociation to create the missing elements in the current metamodel, without changing the current model. Operators to create and delete data types, enums and enum literals are implemented, but not documented in Section 7.3☞243, since they are not used for the application scenarios.
2. Use the existing information in the current model to create model elements conforming to the new elements in the metamodel with the operator →ChangeModel.
3. Use the operators →DeleteNamespace, →DeleteClass, →DeleteAttribute, →DeleteAssociation to delete all elements in the current metamodel which are not used anymore. These operators delete conforming elements in the current model as well.

The steps 1 and 3 of this strategy allow to create any metamodel, while step 2 prevents undesired information loss in the model by creating model elements conforming to the new metamodel elements. Note, that it is important not to switch steps 1 and 3, as it is done in Herrmannsdoerfer, Vermolen and Wachsmuth (2011), since the deletion of metamodel elements (step 3) leads to the deletion of all conforming model elements as well, resulting in undesired information loss in the model.

Operators are complete regarding Kinds of Integration

Additionally, the list of operators is *complete* regarding the possible kinds of integration in general, which are merging of redundancies, linking related elements with each other and generalization of elements (Winter, 2000). This is justified in the following:

Merging of Redundancies can be done with operators to delete metamodel elements and conforming model elements like →DeleteClass or →DeleteAttribute. Additionally, objects representing partially overlapping information can be merged with →MergeClasses and their slots with →MergeAttributes and →MergeReferences, according to Section 12.2.2☞460.

Linking related Elements can be done with →AddAssociation in order to create a new association in the metamodel, which enables conforming links in the model, according to Section 12.2.1☞459.

Generalize Elements i. e. creating classes in the metamodel with a generalization hierarchy in terms of super classes and sub classes can be established with →AddClass for new classes, since the operator provides metamodel configurations for the desired sub classes and super classes for the new class, and with →ExtractSubClass, since the operator extracts a new sub class from an existing class with a generalization between them.

Instead off the above kinds of integration on the granularity of single (meta)model elements, Chechik, Nejati and Sabetzadeh (2012) classify the model integration with higher-level operators (Section 6.1.1☞186) for composing semantically related models, for weaving for integrating cross-cutting models into base models, and for merging of overlapping models. These objectives usually cannot be realized with single operators in MoConseMI, but require chains of operators.

Operators are not minimal

Since Section 7.3☞243 provides more operators than the here mentioned operators for completeness, the list of operators is *not minimal*, since they are designed with the objective

to ease configuration effort for methodologists by supporting recurring transformation patterns. As an example, the functionality of →ReplaceAttributeByReference could be reconstructed by chaining the operators →AddAssociation and →DeleteAttribute.

13.2.3 Complexity of Operators in O-Notation

Complexity of Metamodel Transformations is constant

This section discusses the complexity of operators in terms of the O-notation, where the number of changes in the (meta)model is the quantity of interest: Since the changes in the *metamodel* are the same for each execution of the same configured unidirectional operator, the effort on the metamodel can be seen as *constant*.

Complexity of Model Transformations depends on operators and their configurations

The effort of the transformations in the *model* depends on the operator *and* its configurations, which is demonstrated along some examples: →RenameClassifier does not change the model at all, leading to constant complexity. →MergeClasses searches for corresponding objects conforming to the two merged classes N, M in order to merge two matching objects, which results in the complexity $O(n \cdot m)$ with n as the number of objects conforming to N and m as the number of objects conforming to M, since each object conforming to N must be compared with each object conforming to M. This square amount directly stems from the design of the unidirectional operator without considering concrete configurations for its decisions. The impact of configurations for model decisions is very clear for →ChangeModel, since the whole model transformation is configured by methodologists, who could define model transformations with arbitrary complexity.

13.2.4 Reusability of Operators

The evaluation of the reusability of the designed and implemented operators is done by collecting experiences by applying them to various application examples, which is proposed also by other operator-based approaches (Herrmannsdoerfer, Vermolen and Wachsmuth, 2011). Some of such experiences regarding the reusability of operators are reported: Operators are used multiple times within the same application, leading to the reuse of operators, e. g. →ReplaceAttributeByReference is used to replace indirect relationships via IDs by explicit links in Chapter 11 ☞387. The same operators are used in multiple applications, leading to the reuse of operators, e. g. →AddAssociation is used in all applications including the ongoing example in order to enable additional explicit links between model elements. Additionally, the nesting of unidirectional operators is another way for reusing unidirectional operators, e. g. →SubSetFilter uses operators like →DeleteClass or →DeleteAttribute internally. Summarizing, these examples emphasize, that the designed and implemented operators are reusable.

13.2.5 Imperative vs Declarative Operators

This sections sums up the discussion, why the mostly imperative operators of MoConseMI are not designed to be completely declarative. Consistency goals according to Definition 15 ☞72 can be seen as declarative, since they describe the desired consistency, but not how to reach this consistency. Since there are multiple ways to fix inconsistencies to reach consistency again (Section 2.3 ☞71), consistency rules according to Definition 16 ☞75 concretize consistency goals by specifying strategies, how to fix inconsistency. Thus, consistency rules can be seen as imperative.

declarative Metamodel Decisions

Bidirectional operators can be seen as declarative from the point of methodologists, since they select bidirectional operators and configure them with simple values for metamodel decisions for one direction, while the configurations for the inverse indirection are derived automatically. Additionally, operators realize a static metamodel evolution scenario.

imperative Model Decisions

As discussed and decided in Section 8.3.1☞267, the configurations for the model decisions of unidirectional operators are implemented by methodologists with Java as imperative general-purpose programming language. Benefits are on pragmatic level with an easier use of history maps, branch differences and mixing transformations for models and metamodels. Additionally, managing consistency in change translation-based style derives follow-up changes from the current user change, which is required by some consistency goals and is imperative and not declarative.

Related Work

Related approaches provide some additional hints for this discussion as well: The VITRUVIUS approach (Section 3.5.2☞126) has one restricted declarative language and one more flexible imperative language for defining the desired consistency. "*Declarativeness had no positive impact on any of the case criteria*" (Rose, Herrmannsdoerfer et al., 2012, p. 351) of the Transformation Tool Contest 2010, comparing different tools for model co-evolution including classical model transformations, EPSILON FLOCK and EDAPT (Section 6.2.1☞193). This might be influenced by the compared declarative tools, since they were less mature than the compared imperative tools. In general, some BX approaches start to include imperative parts in order to improve the expressiveness (Chapter 3☞93).

Summary: imperative

Summarizing, the operators in MoConseMI are designed as imperative, since even declarative consistency goals need imperative consistency rules for project-specific realization and imperative configurations for model decisions increases the flexibility and expressiveness for methodologists.

13.2.6 Design of Operators revised

While the previous sections of Section 13.2☞469 evaluated details of the design for operators, this section discusses two general aspects of the operator design, i. e. the joint transformation of models and metamodels, and the advantages of MoConseMI over hand-written code.

joint Transformation of Models and Metamodels

Since operators usually change metamodels and conforming models together, structural refactorings of the metamodel to form the SUMM are somehow mashed with semantic fixes for inconsistencies in the model, which can be seen as violation of the principle for the separation of concerns: Since semantic heterogeneity comes often with structural heterogeneity (and vice versa), it is hard to clearly separate them from each other (Leser and Naumann, 2007, p. 69). With other words, it is necessary to overcome both semantic heterogeneity (in the models) and structural heterogeneity (in the metamodels), which is done by operators, which usually handle both meta-levels together and ensure, that models conform to their metamodels. *Changing metamodels* requires model co-evolution in any case and the degrees of freedom during model co-evolution can be exploited for semantic fixes. *Changing only models* can be done with →CHANGEMODEL in order to focus on semantic changes in the model only. Summarizing, since there are situations where structural and semantic changes occur together, since metamodel evolution requires model co-evolution, and since changing models only is possible with →CHANGEMODEL, the design of operators to jointly change models and their metamodels is reasonable and does not introduce restrictions.

hand-written Source Code vs Configurations with MoConseMI

Since MoConseMI guarantees no important formal properties (Section 13.2.1☞469), the question might arise, why not manually writing pure Java code without the MoConseMI framework for integrations and consistency preservation according to the statement "*Information integration is doable — write enough code and I will connect every software system anywhere*" (Halevy, Ashish et al., 2005, p. 785). While this strategy is possible in general, using the MoConseMI framework provides lots of benefits, in particular, since some model decisions are configurable with Java: The reusable operators guide the development of transformations for models and their metamodels with defined degrees of freedom. These operators ensure the conformance of (changed) models to their (changed) metamodels (except for model decisions that are actively misused by methodologists). Additionally and most importantly, reused operators provide bigger parts of transformations which can

be reused as they are. As an example, the operators →MergeClasses and →SplitClass are implemented in the MoConseMI framework with more than 1000 lines of code (including documentation), covering lots of special cases in ECore, which do not need to be understood by methodologists in all details when (re)using these operators. Additionally, the MoConseMI framework provides the generic execution loop and the calculation of the execution differences, which can be reused directly without writing any source code. Finally, the MoConseMI framework and their providers reduce accidental complexity with managing and bridging other technical spaces into EMF. Summarizing, MoConseMI comes with less formal guarantees which are interesting for platform specialists and researchers, but provides great support for methodologists in order to ensure inter-model consistency in practical application scenarios. Since bigger parts of (meta)model transformations can be reused with operators, using the MoConseMI framework is beneficial compared to manually writing the whole transformation including its execution.

13.3 Characteristics of Orchestrations

Since orchestrations are the result of methodologists, who apply MoConseMI in order to ensure consistency within a particular project, this section discusses some aspects of orchestrations and their configured and chained bidirectional operators. Note, that orchestrations are created and (if necessary) adapted by methodologists only and never by users. Additionally, methodologists can freely chose the order of configured operators, but after arranging operators in chains, their order is stable, since the input of operators depends on the output of their previous operators, as designed in Section 6.4.2☞204.

more Operators for new View(point)s than for integrating Data Sources

When investigating the orchestrations for application examples in Part IV☞283, a first finding is, that operator chains for new view(point)s are often longer than the operator chains for integrating a small number of data sources into the SU(M)M. One explanation is, that the structure of the SU(M)M depends on the structure of the reused data sources and the integration for depending information, while the structure for new view(point)s usually is quite different to the SUMM, which requires more operators for restructuring. Additionally, information which is contained only in one data source is (often unchanged) transformed into the SUM and kept there, while at least one →SubSetFilter is required to hide this information in new views.

13.3.1 Language Evolvability

Metamodels of integrated Data Sources evolve

Language evolvability discusses the scenario, when the metamodels of integrated data sources change. This aspect is motivated by Broy, Feilkas et al. (2010), by France and Rumpe (2007), by Persson, Torngren et al. (2013) (subsumed under extendability) and by Meier, Werner et al. (2020) as selection criterion E4 for SUM approaches, who emphasize the need to discuss this aspect here for MoConseMI as well. Since configured operators and therefore the whole orchestration depend on the metamodel of integrated data sources, the orchestration must be updated, if necessary.

Related Work

Since chains of operators represent (meta)model transformations, this problem is related to transformation co-evolution (Section 6.1.1☞186). This problem is also analyzed in the context of data ware houses (Section 3.6.3☞139), when schemata of their data sources changed: Arora and Gosain (2011) survey various operators to handle different evolution scenarios by various authors. Oueslati and Akaichi (2010) complemented the schema evolution of data sources with the corresponding maintenance i. e. update of materialized views.

The impacts of the *evolution of single metamodels* which are integrated as part of the SUMM depend on the particular metamodel changes. In general, if a single metamodel changes, the integration with all other metamodels into the SUMM has to be checked and

Impact of Metamodel Evolution on the Orchestration depends on the particular Changes

the corresponding operators might need to be updated:

- If the changes of the metamodel effect only parts which are not interesting for the integration, i. e. these parts are neither reused of nor linked by other metamodels, the SUMM can be updated accordingly and automatically, since all metamodel configurations for operators are still valid. This counts in particular for added concepts in evolved metamodels, since the in-place transformations of operators transfer the new concepts directly into the SUMM by design. If these concepts are also provided by other data sources, this leads to some more consistency goals, which can be added easily, as discussed in Section 13.3.4☞477.

- If the changes in the metamodel effect parts which are interesting for the integration, the impact of the changes have to be handled and all subsequent operators need to be updated. In the worst case, all parts of the orchestration after the integration of the data source with the changed metamodel have to be recreated. Therefore, methodologists might integrate data sources with higher expected evolution pressure later than data sources with less frequent evolution. If concepts in the metamodel are only reworked or refactored without loss, these refactorings can be reconstructed by additional operators.

Since the evolvability of languages, i. e. the metamodels of data sources, comes with noticeable effort in MoConseMI, other approaches for ensuring inter-model consistency might be more suited, if the metamodels of integrated data sources change very often. A possible alternative are synthetic approaches like enterprise information integration (Halevy, Ashish et al., 2005), since the impact of metamodel evolution can be solved directly at the data source and its relationships to other viewpoints. On the other hand, the history of integrated models can be tracked much easier in projectional approaches with explicit SUMs, as mentioned by Halevy, Ashish et al. (2005) for more projectional data ware houses in contrast to more synthetic federated data bases.

13.3.2 MoConseMI without reusing Data Sources

Usually, existing data sources are reused with their metamodels and conforming models in order to integrate them into the SU(M)M, according to the design of MoConseMI (Section 5.1.1☞164). This section discusses some alternative scenarios without the strong reuse of data sources.

reuse Metamodel of a Data Source, but no Model

The first variation is to *reuse only the metamodel* of a data source without a conforming model, as for a just started project. In MoConseMI, this is possible in general with the usual configuration of operators into an orchestration, since it mainly depends on the reused metamodels. At the same time, the configurations for model decisions of selected bidirectional operators must ensure, that the reused model of the data source is empty, i. e. configurations for model decisions must no expect the existence of some objects in the (not) reused model. Alternatively, expected objects could be created by additional operators before using them in subsequent operators.

reuse exactly one Data Source

Another case is, that exactly one data source is reused: In that case, this data source is transformed to the SU(M)M without the combination of other data sources, i. e. the special operator ⇄CombineSeparateDataSources is not used in the orchestration.

top-down without reusing any Data Source

The last case is to start top-down with an already existing SU(M)M and without reusing any data source, as in MoConseMI (Section 3.5.1☞124): In that case, the SU(M)M is used as starting point and data sources are neither reused nor combined. Instead, chains of operators are configured only to define new views with the SU(M)M as starting point. Note, that the implementation of MoConseMI (not the approach itself) requires at least one data source to reuse as technical limitation (Section 14.3.2☞495). This limitation of

the implementation can be overcome by using the SU(M)M as data source and applying a single operator with less impact like ⇄RenameClassifier in order to reach the SU(M)M afterwards.

13.3.3 Characteristics of the SU(M)M

Since the SU(M)M is the central part of projectional approaches for ensuring inter-model consistency, some characteristics of the SU(M)M in MoConseMI are discussed. This discussion is done here related to orchestrations, since the *SU(M)M depends on the particular orchestration* and is initially created during the first execution of the orchestration.

SU(M)M depends on Orchestration

13.3.3.1 Quality of the SU(M)M

The quality of the SU(M)M can be controlled by configuring more or less operators by the methodologist. The quality of the SU(M)M in terms of the number of internal dependencies is already discussed in Section 12.2.3☞460 with the recommendation to reduce dependencies as much as possible, since removed dependencies do not need to be kept consistent anymore.

This section focuses on the quality of the SU(M)M regarding simplicity, understandability and best practices for (meta)modeling, since the quality of the SUMM can influence transformations and analyses on it (Hinkel and Burger, 2018). Since the initial quality of the SU(M)M is mainly inherited from the reused and integrated data sources, operators might be configured to refactor the SU(M)M in order to ease the understanding of the SU(M)M and to ease definitions of new view(point)s. This eases the integration process, since previous refactorings of the (meta)models of data sources help preparing the subsequent integration, and since refactorings after the main integration focusing on reducing redundancies for consistency result in an optimized SU(M)M. For these refactoring, all operators might be used which prevent information loss in the model.

improve Understandability of the SU(M)M with additional Operators

These theoretical considerations are made clear along the ongoing example:

Ongoing Example, Part 27: **Quality of the SU(M)M** ← List →

The integration of data sources as depicted in Part 21☞206 of the ongoing example realizes all consistency goals and results in a SU(M)M, as depicted in Part 19☞176 of the ongoing example, but there is still potential for simplifications in the SU(M)M:

- While the representations for classes in Java and UML are merged into `ClassType`, there are still the two root classes `ClassDiagram` and `JavaASG`, which could be merged in order to simplify the SU(M)M, e. g. with ⇄MergeSplitClasses. Note, that the associations `ClassDiagram.classes` and `JavaASG.classes` must not be merged, e. g. with ⇄MergeSplitReferences, since they have different semantics, i. e. `JavaASG.classes` contains all available `ClassTypes`, while `ClassDiagram.classes` indicates the subset of `ClassTypes` which is depicted in the class diagram.
- An additional improvement might be to have all classes in the metamodel within a single namespace. This might ease the SUMM, since there is only one namespace instead of three namespaces, while the three namespaces provide some hints for the origins of the contained classes in terms of their initial data sources. For the definition of the new view(point) in Part 22☞209 of the ongoing example, ⇄SubSet must be configured with all classifiers in the two namespaces `umlclasses` and `asgjava` to hide, but not with only these two namespaces anymore, since they are "optimized" i. e. removed in the SUMM now.

Summarizing, the quality of the SU(M)M depends also on the impressions of methodologists and possible definitions for further new view(point)s.

13.3.3.2 Content of the SU(M)M

This section discusses the content of the SU(M)M, i. e. the concepts in the SUMM and the information stored in the SUM. In particular, this section continues the discussion, whether the SUM should contain all information of all data sources (as designed for MoConseMI in Section 5.1.2☞165), or whether the SUM should contain only the depending information which is relevant for the consistency goals, according to Figure 3.2☞100. This includes information which is not redundant, but is used in configurations for model decisions for ensuring consistency.

store Information which is not necessary for ensuring Consistency in the SUM?

Related approaches disagree on which option should be used and argue differently: The SUM idea strongly advocates to store all available information in the SUM (Section 3.4☞120). Egyed, Zeman et al. (2018) store only information which is required for consistency in order to ease the development of adapters. Baumgart (2010) proposes in his vision paper to store only "common concepts" in the SUMM. Following France and Rumpe (2007), interrelated views should be integrated with the help of a metamodel, but with describing (only) the interrelationships between the involved view(point)s. The Vitruvius approach stores all information in a modular SUM *and additionally* makes naturally redundant concepts explicit as commonalities (Klare and Gleitze, 2019).

Related Approaches disagree

MoConseMI follows the SUM idea and stores all information in the SUM, due to the following reasons: More information, which is superfluous in the sense, that they are not needed for ensuring consistency, does not hurt, *if* there are mechanisms for filtering visualizations and building new tailored viewpoints in order to manage large (meta)models, as discussed in Section 13.3.3.3. While the data sources to reuse and the consistency goals to fulfill might be completely known and would allow to determine a border between necessary and unnecessary information for the SUM, this is not always clear for the information which is required to derive new views from the SUM. If an additional new view is derived later which requires information which is in a data source, but not in the SUM, it is hard to reuse this information for the new view. Having all information in the SUM allows to reuse all information in flexible way for new views.

MoConseMI: all Information in the SUM eases to derive new Views

In MoConseMI, the SUM contains *additional information* which is not part of any data source. In particular, explicit links between information stemming from different data sources (Section 12.2.1☞459) are stored in the SUM. This counts also for traceability links, since they usually cannot be stored within the data sources, since the information of the other end of traceability links is not available.

store explicit Links in the SUM

Finally, the SUM in MoConseMI might store also *information about concrete renderings* like layout information. As an example, the `rowNumber` from the adapters for Excel CSV represent layout information. Such information is included into the SUM, since it might encode semantics like an implicit prioritization (which is not the case here), it is required for the backward transformation (which is not the case here), or it should be used for new views (which is the case here, since the order of requirements is the same in Requirements and in Traceability).

store Layout Information in the SUM

13.3.3.3 Large (Meta)Models

Depending on the content of the SU(M)M (Section 13.3.3.2), the (meta)models for the SU(M)M might become large. This challenge depends not on the integration in MoConseMI, but on the current projects, in particular, on the number and size of reused (meta)models. According to France and Rumpe (2007), who mention the size of metamodels as challenge and request better tool support for this challenge, large SU(M)Ms are a problem of lacking tool support, not a limitation of MoConseMI in the general, since for ensuring the consistency between large (meta)models, they must be related to each other in any case, even in synthetic approaches. Since "*Big Metamodels Are Evil*" (Fondement, Muller et al., 2013), they introduce operators to unmerge packages for reduction.

large SU(M)M depending on the particular Project

In order to manage large (meta)models, MoConseMI provides several concepts: First of all, the large SU(M)M is not given to users, only the smaller data sources or new views which are tailored to the concerns of users are usually provided for users. Therefore, large SU(M)Ms are only a challenge for methodologists, which are supported with namespaces for metamodels (Section 6.6.2 ☞222) and models (Section 6.6.3 ☞223). Since the models and metamodels of each data source come with their own namespaces, these namespaces are transformed into the SU(M)M by default and indicate the origin of their contained elements. These namespaces are explicitly supported with operators and can be used also for filtering with ⇄SubSet. Thanks to redundancy reduction (Section 12.2.2 ☞460), the number of elements in the SU(M)M might be smaller than the number of all elements in all data sources together. On technical level, most of the visualizations provided by the MoConseMI framework (Section 8.5 ☞279) support filters to visualize only elements of interest.

Users use (partial) Views, not the complete the SUM

Support for Methodologists

13.3.4 Reusability of Orchestrations

The reusability of orchestrations covers the scenario, that parts of a complete orchestration of a particular project are reused for another project. This scenario is motivated by Meier, Werner et al. (2020) and called "SUMM Reusability" as selection criterion E5 for SUM approaches, while the reusability of *orchestrations* emphasizes, that also the mechanisms for ensuring consistency within a SU(M)M should be reused as well. This scenario is done by methodologists, never by users, since users use data sources and update their models, but do not configure orchestrations. This scenario includes to add and remove data sources, new view(point)s as well as consistency goals and consistency rules for an existing orchestration.

adapt and reuse Parts of Orchestrations

Similar to the evolution of single metamodels in Section 13.3.1 ☞473, the effort for reusing parts of orchestrations depends on the details what to reuse, since chains of operators are fixed after configuration, since the output of one operator is the input for the subsequent operator. Some of the possible cases during the reuse of orchestrations and their SUMMs are sketched as evolution scenarios in the following publication:

Related MoConseMI Publication

Johannes Meier and Andreas Winter (2018b): *Towards Evolution Scenarios of Integrated Software Artifacts.* In: Softwaretechnik-Trends, vol. 38(2), pp. 63–64.

This publication is cited as Meier and Winter (2018b) in this thesis.

- *Adding another data source* is easy, since the tree of operators is extended with additional operators in order to integrate the current SU(M)M with the additional data source.

add Data Source

- *Removing an already integrated data source* is harder, since the integrated parts of its metamodel usually influence the configured operators, the SUMM and sometimes also new viewpoints. In the worst case, all parts of the orchestration after the integration of the removed data source have to be reworked. More consistency goals, in particular n-ary consistency goals, increase the integration and coupling of data sources into the SUMM and make the deletion of single data sources harder. Removing the last integrated data source is the easiest case, since only its chain operators to the SUMM must be updated or removed. An alternative might be to keep the data source and its metamodel, which allows to keep all depending operators, but to use this data source with an empty model, as discussed in Section 13.3.2 ☞474. Only the configurations of some model decisions might need an update in order to work with an empty model of the "inactivated" data source.

remove Data Source

add new Viewpoint

- *Adding new viewpoints* is easy, since only an additional chain of operators must be configured.

remove new Viewpoint

- *Removing new viewpoints* is also easy, since its chain of operators to (usually) the SUMM can be simply removed, since new viewpoints neither introduce new consistency goals nor provide new concepts, since they are completely derived from the SUMM.

add Consistency Goal

- *Adding consistency goals* results in additional operators, which improve the SU(M)M in form of an operator chain between the old SU(M)M and the new SU(M)M. Therefore, additional consistency goals can be supported easily.

remove Consistency Goal

- *Removing realized consistency goals* is harder, since operators which realize these consistency goals become unnecessary. If they are removed and they changed the metamodel, subsequent operators have to check and to adapt accordingly, if necessary. Therefore, an alternative could be keep the metamodel changes and to configure another strategy for model co-evolution, which keeps all information, but does not realize the consistency goal anymore.

Summarizing, *adding* additional data sources, new viewpoints and consistency goals is easy, since the existing orchestration is extended with another chain of operators, starting at the old SUMM and resulting in a new SUMM. *Removing* data sources and consistency goals is much harder, since the subsequent parts of the orchestration must be reworked. Due to their low modularity, the reusability of orchestrations and their SUMMs is limited in MoConseMI. The modularity of orchestrations is better in approaches using synthetic techniques, since the metamodel and its direct relationships to other metamodels can be directly removed, like in Vitruvius, whose main design goals include high modularity (Klare and Gleitze, 2019).

13.4 Conceptual Discussions on MoConseMI

This section discusses some conceptual characteristics of the whole MoConseMI approach. In particular, the inter-play of existing techniques and applications with MoConseMI is discussed.

13.4.1 Reuse existing Modeling Techniques

Modeling Techniques use explicit (Meta)Models

This section discusses, how existing modeling techniques can be applied for models managed by MoConseMI. In general, reusing other modeling techniques is easy with MoConseMI, sine the (meta)models for all data sources, new view(point)s and the SU(M)M are explicitly available and can be used as input for other modeling techniques. If information of models of multiple views should be used by other modeling techniques, these modeling techniques have to support multiple (meta)models as input, which is not true for each modeling technique. In these cases, the explicit SU(M)M of MoConseMI is beneficial, since it is a single (meta)model, which contains all concepts and all information of all data sources in integrated way and can be used as input for modeling techniques supporting only one (meta)model. These general ideas count for most of the existing modeling techniques, as detailed for some examples:

Modeling Techniques with one (Meta)Model as input use the SU(M)M

Cross-View Analyses Analyses of information which is spread over multiple views is hard to realize with query languages which support only one metamodel. The same counts for visualizations targeting information spread over multiple views. Here, using the SU(M)M is beneficial, since it contains all concepts and all information of all data sources in integrated way.

Constraints between different (Meta-)Models Constraints for relations of elements stemming from metamodels of different data sources can be expressed easier with a SUMM, since the constraint developer has to take into account only the SUMM as one metamodel, which contains the concepts of all data sources in integrated form. Additionally from a technical point of view, most of the constraint languages work only for one metamodel, which requires to use the SUMM.

Model Versioning For the combined versioning of models for different views, approaches for model versioning usually require one metamodel which describes the complete model, like the DL approach (Kuryazov, 2019). Therefore, the SUMM is beneficial and can be used for versioning the whole SUM, from which the models for the views can be derived.

Model Transformation Lots of model transformation languages support only one source metamodel, while there are also some model transformation languages supporting multiple source metamodels (Kahani, Bagherzadeh et al., 2019, p. 2374f). For transformations requiring multiple models from different views, model transformation languages supporting only one source metamodel are not usable, since there are multiple source metamodels. Here the SUMM allows to use even model transformation languages which support only one source metamodel in order to transform information stemming from multiple data sources.

Model Refactoring Misbhauddin and Alshayeb (2019) evaluate for UML, that refactoring on an integrated (meta)model is beneficial compared to refactorings on the single (meta)models. While they had to do the integration as preparation for the refactoring, using MoConseMI provides the SU(M)M directly. Therefore, (other) refactoring techniques could be reused and could be done directly on the SU(M)M.

13.4.2 Integrate Data Sources with different Abstraction Levels

A big advantage of MoConseMI is, that it enables to integrate (meta)models of data sources with heterogeneous levels of abstraction, e. g. in the ongoing example, requirements at a high abstraction level are integrated with Java source code at a low abstraction level with lots of details. In general, supporting different levels of abstraction provides benefits for companies (Mohagheghi, Gilani et al., 2013a, p. 107), while integration of models with heterogeneous abstraction levels is a very difficult problem (Hailpern and Tarr, 2006). For UML, Lucas, Molina and Toval (2009) found, that there are only few approaches for consistency covering different levels of abstraction. Programmers tend to integrate on code level (Burden, 2014), which is not sufficient, since there are also development projects without source code.

supporting different Abstraction Levels is important

MoConseMI supports the integration of data sources with different abstraction levels in general by providing operators for reuse, which are independent from the concrete metamodels thanks to their metamodel decisions and which are independent from the abstraction levels of these metamodels as well. Operators like →AddAssociation and →ReplaceAttributeByReference for linking elements, as discussed in Section 12.2.1 ☞459, can be used also for relating elements at different levels of abstraction with each other.

Support by MoConseMI

13.4.3 Integrate other Research into MoConseMI

The MoConseMI approach is open to other research and allows to integrate related research, as sketched for the following examples: Since the managed (meta)models are explicitly available, other modeling techniques can be reused, as discussed in Section 13.4.1 ☞478.

Approaches which manage consistency in manual or semi-automated way by *users* could be integrated into new view(point)s, provided by MoConseMI. *Methodologists* could be supported with results from metamodel matching (Lopes, Hammoudi et al., 2006) and with techniques from data integration in the data base research for selecting and configuring operators and configurations of their model decisions (Doan, Halevy and Ives, 2012). Contributions of MoConseMI for further research areas are discussed in Section 14.1.2☞486.

13.4.4 Integrate MoConseMI into other Applications

Since the implementation of MoConseMI is done as a reusable Java framework in form of a Maven library according to Requirement R 5 (Reusable Library)☞158, MoConseMI could be integrated into other applications in general.

Additionally, ensuring inter-model consistency with MoConseMI could be deployed on a server, since the MoConseMI framework does not force a GUI, and the adapters of the views exchange models with clients, which enables the spatial separation of users, while they are supported with automated fixes for inconsistencies. For future work, this idea could be extended with support for model versioning of all models, transaction management (Section 12.4☞463) and management of access rights. In general, there is a need for such central model repositories, as surveyed by Di Rocco, Di Ruscio et al. (2015). Model repositories might ease to manage a SU(M)M by embedding a configured MoConseMI instance together with support for collaboration.

13.4.5 Intra-Model Consistency

As defined in Definition 6☞42, *intra-model consistency* addresses the internal consistency within one model, while this thesis focuses on *inter-model consistency* addressing the consistency of depending elements in different models. For TGGs, intra-model consistency is called “domain correctness” and expressed by domain constraints to determine consistent domain models (Anjorin, Leblebici and Schürr, 2016). This section discusses, how MoConseMI can ensure also intra-model consistency.

Operators for Intra-Model Consistency

In MoConseMI, intra-model consistency goals are realized with the same operators as for inter-model consistency goals: Removing possible redundancies or unused information can be done with the same operators, which are also used for reducing redundancies for inter-model consistency, as discussed in Section 12.2.2☞460. Additionally, ⇄ChangeModel allows to realize other intra-model consistency goals. Since intra-model consistency goals should be fulfilled before realizing inter-model consistency goals (according to Definition 7☞42), operators for ensuring intra-model consistency are often configured after reusing a data source and before their technical combination with other (meta)models.

Inter-Model Consistency becomes Intra-Model Consistency in MoConseMI

By combing multiple data sources into one SU(M)M (Section 6.4.3☞205), all inter-model consistency goals are realized within one model like for intra-model consistency goals. Therefore, there is no difference between intra-model consistency and inter-model consistency on *technical* realization level in MoConseMI. In other words, by combining (meta)models, inter-model consistency becomes intra-model consistency in general, which is found by Egyed, Zeman et al. (2018, p. 32) as well.

Possible intra-model consistency goals are concretized along the ongoing example:

Ongoing Example, Part 28: **Intra-Model Consistency** ← List →

The ongoing example could be extended with consistency goals which address only one data source and therefore represent intra-model consistency:

In Requirements, the row number of requirements (`Requirement.rowNumber`) could be constrainted by consistency goals: In particular, a correct row number is required for newly

created requirements. Additionally, each row number should occur only once, and gaps in row numbers caused by removed requirements should be avoided. This could be realized with an additional ⇄ChangeModel for Requirements in the orchestration, which checks the row numbers of all requirements and fixes wrong row numbers.

In Java, all methods within the same class must have unique names.

In Java, each class must have at least one constructor, which is a method with the same name as the class. This could be realized with an additional →AddAssociation for Requirements in the orchestration, which introduces a new association which indicates the special method(s) used as constructors. This allows to rename classes and their constructors.

In ClassDiagram, it might be ensured, that each `Association` has always a *type*, even if the corresponding class was just removed. Usually, such a fix is not required, if the change is done in ClassDiagram, since the multiplicity of 1 for the `type` ensures, that there is always a class as type. But if the change is made inside Java (or in the SUM or a new view), such a fix is required, if the `Associations` are hidden and therefore cannot be fixed by users. The fix could be realized with an additional →ChangeModel for ClassDiagram in the orchestration, which randomly adds a available class as new type. There is always at least one such class i. e. the class owning the association. This example indicates, that the clear separation between intra- and inter-model consistency becomes blurred during the integration of data sources and the definition of new views.

13.5 Summary of the Evaluation

Summarizing the evaluation, the *application examples* in Part IV☞283 show, that the MoConseMI approach as well as its implementing MoConseMI framework work and ensure inter-model consistency in practice. Section 13.1☞467 evaluated, that all *requirements* for MoConseMI are fulfilled. Since the three *challenges* for ensuring inter-model consistency in Section 1.2☞31 are directly addressed by the three functional requirements Requirement R 1 (Model Consistency)☞154, Requirement R 2 (Reuse existing Artifacts)☞155 and Requirement R 3 (Define new View(point)s)☞156, the corresponding challenges in Section 1.2.1☞31, Section 1.2.2☞36 and Section 1.2.3☞39 are fulfilled as well. Since the fulfillment of these three challenges forms the objective of this thesis in Section 1.3.1☞42, MoConseMI fulfills the *objective* of this thesis. Additionally, the proposed *deliverables* of Section 1.4☞47 are provided with the MoConseMI approach (Chapter 5☞163, Chapter 6☞185, Chapter 7☞241), its implementing MoConseMI framework (Chapter 8☞263) and the applications examples (Part IV☞283).

Furthermore, this section evaluates the characteristics of operators (Section 13.2☞469) and orchestrations (Section 13.3☞473) in MoConseMI and justifies the applicability of MoConseMI in a broader context (Section 13.4☞478). Possible limitations of this evaluation are discussed in Section 14.3.3☞495. Remaining investigations which are possible in the future are discussed in Section 14.4.1☞496.

Chapter 14

Conclusion

This section summarizes the thesis in terms of its contributions in Section 14.1, the preconditions for MoConseMI in Section 14.2[☞489], its limitations in Section 14.3[☞490], and possible extensions as future work in Section 14.4[☞496]. Section 14.5[☞499] summarizes the results of this thesis.

14.1 Contributions

This section summarizes the contributions of this thesis and in particular of the developed MoConseMI approach for ensuring model consistency (Section 14.1.1), for other research areas within software engineering (Section 14.1.2[☞486]), and for some application domains (Section 14.1.3[☞488]).

14.1.1 Contributions to Model Consistency

Since ensuring the consistency of models is the main motivation and challenge to overcome for this thesis (Section 1.2.1[☞31]), this section summarizes the contributions of this thesis for model consistency. In particular, MoConseMI as new projectional SUM approach for ensuring inter-model consistency is summarized in Section 14.1.1.1, whose main characteristics are discussed in comparison with related approaches in Section 14.1.1.2[☞484].

14.1.1.1 New SUM Approach

With MoConseMI, this thesis contributes a new SUM approach (Meier, Klare et al., 2019) for ensuring inter-model consistency. MoConseMI follows the SUM idea (Section 3.4[☞120]) and propagates changes between views and the explicit SUM in order to keep all views consistent to each other. *Users* use views and manually change them, while MoConseMI propagates these user changes automatically to all other views, according to the consistency goals of the particular project. The configuration of MoConseMI according to the project-specific metamodels and consistency goals is done once by *methodologists*. The MoConseMI approach is implemented in the MoConseMI framework, which is applied to several application examples, showing the applicability of MoConseMI for projects with demand for ensuring inter-model consistency in practice. Additionally, MoConseMI supports to integrate and to keep models with different levels of abstraction consistent to each other (Section 13.4.2[☞479]). Finally, MoConseMI ensures intra-model consistency as well (Section 13.4.5[☞480]).

MoConseMI is a new SUM Approach for ensuring Inter-Model Consistency

MoConseMI is model synchronization-based and operator-based in order to support methodologists during their configurations of MoConseMI for application in particular projects. MoConseMI provides lots of reusable bidirectional operators (Chapter 7[☞241]),

MoConseMI provides reusable Operators for Methodologists

which are reused and configured by methodologists in order to define the transformations between all view(point)s and the SU(M)M, which are automatically executed by MoConseMI in order to propagate changed model parts between all views.

14.1.1.2 Characteristics of MoConseMI revised

Since MoConseMI is neither the first approach for ensuring inter-model consistency nor the first SUM approach, this section discusses the main characteristics of MoConseMI in comparison with some related approaches.

conceptual Characteristics of SUM Approaches

Two main characteristics of MoConseMI are the reuse of existing (meta)models by starting bottom-up *and* the use of an explicit SU(M)M, which makes MoConseMI unique compared with the other three SUM approaches, as discussed along the ongoing example:

Ongoing Example, Part 29: **Related SUM approaches** ← List →

The ongoing example can be realized also with other SUM approaches, as investigated by Meier, Werner et al. (2020). Figure 14.1 shows the applications of all four SUM approaches for the ongoing example by in graphical way. Additionally, Figure 14.1 visualizes the classification of the projectional SUM approaches regarding the features of Figure 3.8☞124 for conceptual design choices.

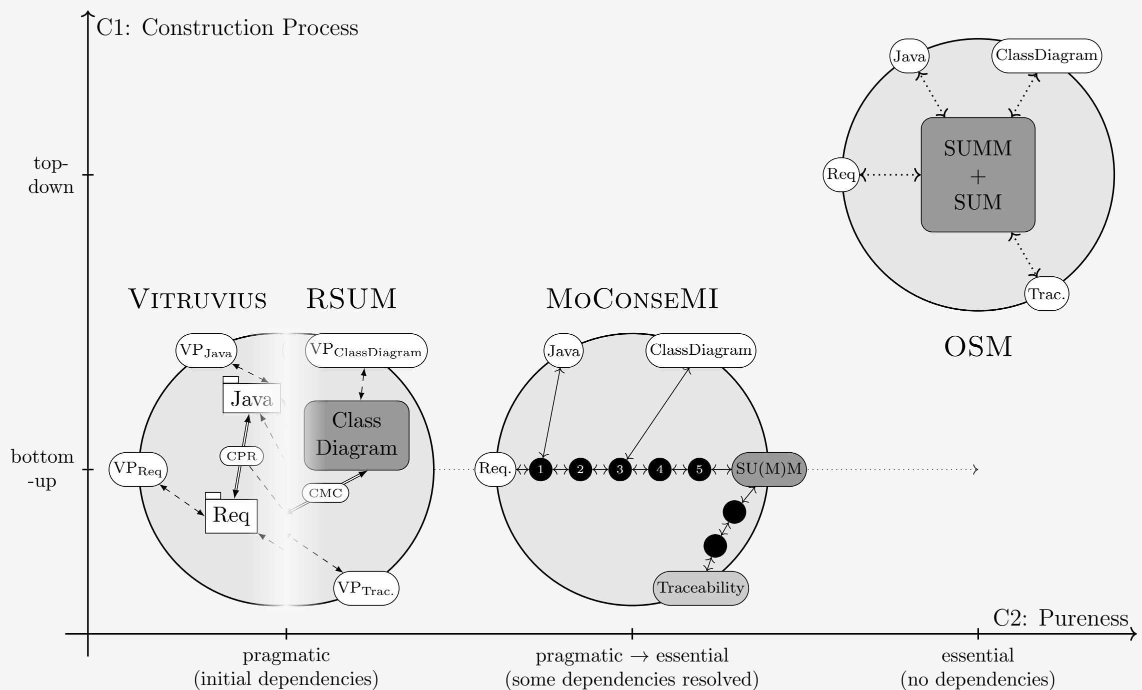


Figure 14.1: Conceptual Classification (taken from Meier, Werner et al. (2020))

Vitruvius and RSUM are both bottom-up in order to reuse existing (meta)models as they are, and they are pragmatic with keeping all dependencies in a modular SU(M)M. Dedicated mechanisms are provided to keep the depending models consistent to each other. The complete visualizations of these two SUM approaches can be found in Figure 3.11☞127 respectively in Figure 3.13☞130. In contrast, OSM starts top-down with a predefined, explicit SU(M)M without any internal dependencies, which makes additional means for ensuring consistency superfluous, while a strategy to create such a SU(M)M with reuse of existing (meta)models is missing.

This motivates to develop MoConseMI as the fourth projectional SUM approach (visualized in Figure 5.1☞172). MoConseMI starts bottom-up in order to reuse (meta)models

as they are, but integrates them into an explicit SU(M)M by explicitly linking depending elements and reducing an adjustable number of redundancies in order to ensure consistency (Section 12.2 ☞458). Summarizing, MoConseMI combines ideas of the current SUM approaches Vitruvius and RSUM for reusing (meta)models on the one side, and of OSM for an explicit SU(M)M without any internal dependencies on the other side. In particular, MoConseMI provides a pragmatic strategy to create a SU(M)M in bottom-up way with reusing exising (meta)models and therefore answers the question, how to create a SU(M)M with reuse of existing view(point)s (Atkinson, Stoll et al., 2013).

Some more characteristics of SUM approaches are compared by Meier, Werner et al. (2020) and are summarized in Table 14.1: The reusability of metamodels (E1) refers to Requirement R 2.1 (Reuse existing Metamodels) ☞155 and is easily supported by all bottom-up approaches including MoConseMI, since they use existing metamodels as starting point. The reusability of models (E2) refers to Requirement R 2.2 (Reuse existing Models) ☞156 which is supported by all bottom-up approaches, while MoConseMI outperforms due to its ability to fix some initial inconsistencies within the reused models, according to Requirement R 2.3 (Fix existing Models) ☞156.

MoConseMI reuses and fixes existing (Meta)Models

Criterion		OSM	Vitruvius	RSUM	MoConseMI
C1	Construction Process	top-down	bottom-up	bottom-up	bottom-up
C2	Pureness	essential	pragmatic	pragmatic	pragmatic → essential
E1	Metamodel Reusability	hard	easy	easy	easy
E2	Model Reusability	hard	middle	middle	easy
E3	Viewtype Definability	easy	hard	middle	middle
E4	Language Evolvability	middle	easy	middle	middle
E5	SUMM Reusability	middle	easy	easy	middle
T1	Configuration Languages	ECore, DeepATL	Mappings/React., ModelJoin	RCs, ModelJoin	Bidirectional Operators
T2	Meta-Metamodel	PLM	ECore	CROM	ECore

Table 14.1: Classification of projectional SUM approaches (taken from Meier, Werner et al. (2020))

Defining new view(point)s (E3 "Viewtype Definability" in Meier, Werner et al. (2020)) is possible for all SUM approaches, while approaches with one explicit SUM including MoConseMI do not need to collect information from multiple models within a modular SUM. Since the SUM of OSM contains no internal dependencies by design, the SUM of MoConseMI might be completely optimized in theory, but often some dependencies of the reused models remain (Section 12.2.3 ☞460). When defining new view(point)s with MoConseMI, any operations on the SU(M)M can be realized to derive the new view(point), not only typical ones like selection, joins and renames (Bruneliere, Burger et al., 2019). In particular, while the other three SUM approaches have restricted concepts for new view(point)s in order to support modifiability, MoConseMI could support arbitrary new view(point)s in general, but requires to specify also the inverse direction with configurations for model decisions of the inverse unidirectional operators. Additionally, defining new view(point)s is eased for methodologists by using the same operators as for the integration of data sources into the SU(M)M for ensuring consistency.

MoConseMI enables easy and flexible Definitions for new View(point)s

The evolvability of languages i. e. the evolution of reused metamodels (E4) is lower in approaches with some integrations of the reused metamodels including MoConseMI compared to approaches with a modular SUMM, since not only evolved metamodel must be handled but also the integrated SUMM. Depending on the particular changes in the metamodel, the adaptation effort might be small, e. g. only some operators need to be added, or large, if large portions of the orchestration need to be reworked for MoConseMI, as

Evolution of Metamodels and Orchestrations is limited in MoConseMI for some Cases

discussed in Section 13.3.1☞473. The reusability of SUMMs (E5) includes also the reusability of the mechanisms for ensuring consistency within the SU(M)M. Approaches with a modular SUMM allow to easily add and remove parts in comparison to approaches with an explicit and optimized SUMM including MoConseMI. While integrating additional data sources, defining new viewpoints and fulfilling additional consistency goals is easy, removing data sources is hard, since they are integrated with other data sources into the explicit SUMM, as discussed in Section 13.3.4☞477.

Furthermore, MoConseMI is projectional, but without restricted projectional editing in contrast to some DSL workbenches like Jetbrains MPS, since views are loosely coupled with the SUM by model transformations (Section 3.6.2☞137) and therefore allow any editing mode.

MoConseMI is model synchronization-based and calculates ${}^{E}\Delta$

MoConseMI behaves like change translation-based approaches from an outside perspective, but realizes it like model synchronization-based approaches from an internally perspective. In particular, the configured operators synchronize models by chains of model transformations, and the desired execution differences ${}^{E}\Delta$ are derived from these model transformations in generic way, while change translation-based approaches create ${}^{E}\Delta$ directly. As an example, such a direct translation of changes, as requested by (Hidaka, Tisi et al., 2016, p. 922), is supported by Vitruvius (Section 3.5.2☞126). MoConseMI might change the model with the user changes and even might amend the user changes, therefore, there are no authoritative models (Stevens, 2017) in MoConseMI.

MoConseMI's SUM is adjustable, supports n-ary Consistency and reduces Integration Effort

Regarding the discussion synthetic vs projectional approaches, Karsai (2014) proposes, that defining a unifying semantics (in form of a SUM) for all models is unrealistic in practice. Instead, the integration should be tailored to the current purpose and needs. Since Karsai (2014) proposes integration models for such an integration, that integration tends to become synthetic. MoConseMI is suited here, since it allows methodologists to integrate as much as desired *and* this integration results in an explicit SUM, integrated at least regarding these concepts. By the explicit integration of data sources into the SUM in MoConseMI, n-ary consistency relations between data sources which cannot be realized by pairs of binary relations (Macedo, Cunha and Pacheco, 2014) in synthetic approaches can be realized on the explicit SUM. Additionally, by reducing redundancies during the integration of data sources into the SU(M)M, redundancies are treated only once for each data source with these redundant information (linear effort) in contrast to synthetic approaches, where redundancies need to be treated for each pair of data sources with these redundant information (square effort).

MoConseMI supports Methodologists and Users

MoConseMI handles the "*balance between automation and personalization of repair*" (Barriga, Rutle and Heldal, 2019, p. 175) by introducing the methodologist as additional stakeholder, while BX approaches often discuss only users and platform specialists explicitly. The *methodologist* realizes the personalization of generic operators for the particular project once and enables the ongoing and automated support for *users* for fixing inconsistencies. Additionally, the manual handling for some dependencies by users is still possible via new views (Section 12.2.3☞460).

14.1.2 Contributions to other Research Areas

The contributions of MoConseMI for model consistency (Section 14.1.1☞483) come with some further contributions for other research areas within software engineering, which are summarized in the following sections.

14.1.2.1 Traceability

Traceability (Seibel, Neumann and Giese, 2010) interconnects related artifacts of development projects and is motivated by several related works (Kuhn, Murphy and Thompson,

2012; Briand, Falessi et al., 2012). Vogel-Heuser, Fay et al. (2015) identify, that traceability between different artifacts and the automation of trace management require additional support and future research. MoConseMI provides support for managing traceability, as discussed in the following paper and shortly summarized here:

Related MoConseMI Publication

Johannes Meier and Andreas Winter (2018c): *Traceability enabled by Metamodel Integration.* In: Softwaretechnik-Trends, vol. 38(1), pp. 21–26.

This publication is cited as Meier and Winter (2018c) in this thesis.

Pair-wise trace links between arbitrary elements of arbitrary data sources can be stored directly and explicitly in the SUM, which is supported by operators like →AddAssociation in order to make implicit relationships between models explicit, as detailed in Section 12.2.1☞459. Additionally, operators can react on changes and maintain trace links, e. g. by removing trace links of deleted elements or by adding candidates for additional trace links based on heuristics. In the end, traceability demands can be seen as special consistency goals and managing traceability becomes a natural part within overall consistency management with MoConseMI.

store Trace Links in the SUM

manage Traceability like Consistency Goals

14.1.2.2 Round-trip Engineering

Round-trip engineering, as introduced in Section 3.3.1☞108 and motivated by several related works (Hailpern and Tarr, 2006; Lettner, Tschernuth and Mayrhofer, 2011; Vogel-Heuser, Fay et al., 2015), can be fulfilled by transformations of MoConseMI, since the bidirectional operators support backward transformations. Additionally, operators in MoConseMI prevent information loss (Section 6.5.2☞214) even when models with different levels of abstraction are integrated (Section 13.4.2☞479).

MoConseMI supports Round-trips with inverse Transformations and prevents Information Loss

Looking at the ongoing example, classes in Java and class diagrams are partially overlapping with associations only in class diagrams and methods only in Java. Nevertheless, renaming of a class in the class diagrams does not remove methods of the corresponding class in Java, but leads only to the corresponding renaming without information loss, as depicted in Part 18☞173 of the ongoing example.

14.1.2.3 Model Co-Evolution

Since unidirectional operators jointly change metamodels and conforming models, MoConseMI contributes also to model co-evolution research. The unidirectional operators realize recurring metamodel evolution scenarios and provide a generic strategy for corresponding model co-evolution, that makes degrees of freedom configurable with model decisions. With this design, MoConseMI reduces the effort for writing the whole model migration compared to *resolution strategy langauges* and improves the flexibility for project-specific model co-evolution compared to *predefined resolution strategies* (Section 6.2.1☞193). In particular, the model decisions of unidirectional operators in MoConseMI improve the flexibility for project-specific model co-evolution in comparison with Edapt, whose design was partially reused for unidirectional operators in MoConseMI (Section 6.2.2☞196), while Edapt provides only fixed model co-evolution. Therefore, MoConseMI is an approach for model co-evolution, which balances reusable metamodel evolution scenarios with customizable model co-evolution.

MoConseMI provides flexible Model Co-Evolution

14.1.2.4 Model Transformations

Model transformations can be orthogonally classified regarding the metamodels of source and target, where *exogeneous* model transformations have different metamodels for source and target and *endogeneous* model transformations have the same metamodel for source and target, and regarding the models of source and target, where *out-place* model transformations create a new model for the target and *in-place* model transformations update the source model and provide it as target model (Section 2.2.3☞67). This classifications are depicted in Table 14.2 with some examples.

	exogeneous	**endogeneous**
in-place	MoConseMI, Edapt	various approaches e. g. Henshin
out-place	various approaches e. g. ATL	refine / merge / copy models

Table 14.2: Orthogonal Classification of Model Transformations

MoConseMI enables in-place and exogeneous (Model) Transformations

While most model transformation approaches are either out-place and exogeneous like ATL or in-place and endogeneous like Henshin, examples for out-place and endogeneous model transformations include model refinements, model merging and copying models. Model synchronization approaches are usually exogeneous and out-place as well. MoConseMI is a rare example for an in-place and exogeneous transformation approach, since different metamodels for in-place (model) transformations require to change the metamodel as well, which is beyond traditional model transformations approaches. This counts also for Edapt, but MoConseMI with model decisions provides more expressiveness for transforming models (Section 14.1.2.3☞487).

14.1.2.5 Model Differences

MoConseMI provides an invertible and integrated Representation of Model and Metamodel Differences

MoConseMI provides several contributions for model differences: Section 6.7.2☞229 designs a new approach for the *integrated, model-based and (meta)metamodel-independent representation of differences in models and their metamodels*, which are *invertible* without additional information like complete (meta)models. This integrated representation supports some cases for *model difference co-evolution* in Section 6.7.3☞235. On technical level, model differences calculated with EMF Compare can be converted to this new difference representation and recorders are available for recording changes in EMF metamodels and Edapt models.

14.1.3 Contributions to Application Domains

MoConseMI provides also contributions for some application domains, as sketched in the following sections.

14.1.3.1 SEIS Architecture

MoConseMI integrates Architecture Viewpoints and Views into a SU(M)M with some Consistency Support

Chapter 10☞373 realizes the proposed integration of four viewpoints for architectures (Hofmeister, Nord and Soni, 2000) (and two additional viewpoints for SEIS architectures) explicitly into a SUMM. Additionally, conforming views are integrated in a conforming SUM. Both integrations are supported technically with the MoConseMI framework. Furthermore, the realization with MoConseMI provides also some support for ensuring inter-view consistency.

14.1.3.2 Table-oriented Data Management

MoConseMI supports the table-oriented data management in form of spreadsheets by providing model-based representations for such data managed in spreadsheets in order to apply modeling techniques on them like approaches for managing consistency. These model-based representations are enabled by adapters for CSV (Section 8.4.4☞275) and Excel (Section 8.4.3☞273), which are used by several application examples in Part IV☞283. These adapters transform content of CSV and Excel files into EMF-based metamodels and models.

MoConseMI provides Adapters for CSV and Excel

14.2 Preconditions

This section discusses preconditions, which must be fulfilled, before MoConseMI is applicable. These preconditions depend on the design of the MoConseMI approach.

14.2.1 Data as Model

Data which should be kept consistent by MoConseMI need to be available, e. g. data managed by existing tools need to be *exportable* and *importable* from that tool. Available data must conform to a structure, which can be expressed as metamodel. This structure might be implicit (Jin, Cordy and Dean, 2002) or might depend on conventions, as in spreadsheets with rows for the header i. e. the structure and rows for the content i. e. the "real data". Additionally, the data must be transformable into models conforming to this metamodel. These transformations are supported by MoConseMI with the design of reusable adapters (Section 6.6.5☞226) and with predefined adapters for concrete technical spaces (Section 8.4☞271), while adapter providers are able to develop additional adapters for additional technical spaces, as demonstrated in Chapter 9☞283. This clear separation into metamodels and conforming models is required to define consistency goals and consistency rules on metamodels in a generalized way in order to ensure the consistency of all conforming models (Section 2.2.2☞60, Section 2.3☞71). Summarizing, using data in form of models conforming to metamodels is a precondition for MoConseMI.

Precondition: available Data as Models conforming to Metamodels

Since the design of MoConseMI requires stable UUIDs for all model and metamodel elements (Section 6.6.4☞225), operators, their configurations and adapters need to ensure stable UUIDs as another precondition for MoConseMI, which hamper the work of platform specialists, methodologists and adapter providers. On the other hand, UUIDs are also beneficial for transparency in visualizations, since models can be visualized with various model differences within the same graphic, including operator differences and branch differences, with examples in Part 18☞173 of the ongoing example.

stable UUIDs

14.2.2 Supportable Consistency Goals

While Requirement R 1.2 (Generic Consistency Goals)☞155 requires MoConseMI to be generic and to support project-specific consistency goals in general, the following paragraphs discuss some preconditions for consistency goals and consistency rules in order to be supportable by MoConseMI. Complementing the following conceptual preconditions, consistency goals and consistency rules must comply with legal requirements.

Preconditions for Consistency Goals

MoConseMI is able to automate fixes for inconsistencies only for consistency goals, whose consistency rules can be automated in general. Calculating fixes must be supportable by algorithms without manual user interaction. These algorithms can be used as configurations for model decisions of unidirectional operators in order to realize transformations, which are fully automatic (Stevens, 2008). Non-automatable consistency goals cannot be automated with operators, but might be supported with new view(point)s, as discussed in

automatable Consistency Goals and Consistency Rules

Section 12.2.3☞460. This includes also other manual or collaborative modeling techniques like matching of models (Bennani, El Hamlaoui et al., 2018). In the ongoing example, determining traceability links between Java methods and requirements cannot be automated in general, since algorithms do not know the intentions of developers and can provide only candidates for traceability links with heuristics. Regarding the classification for levels of possible automation for consistency goals by Kramer (2017, p. 90), consistency goals in level 1 must be ensured completely manually by users, consistency goals in levels 2–5 can be supported with dedicated hints and suggestions in new views for users, and consistency goals in level 6 can be automated by configured operators. Demuth, Lopez-Herrejon and Egyed (2015, p. 582) suggest, that users should decide, if a consistency goal is ensured automatically or manually. Here, the methodologist decides about the automation, since only methodologists can realize the automation, not users. Of course, methodologists should realize the needs of users, in particular, they should communicate with each other regarding the desired degree of automation (Section 12.1☞455).

Consistency Goals must have a Fix-Point

Consistency goals and their *consistency rules must not contradict each other*, since fixing the first inconsistency would lead to the second inconsistency whose fix introduces the first inconsistency again. For example, when formalizing the consistency goals with OCL, the resulting OCL constraints must not have conflicts, which could be checked, for example, with MaxUSE (Wu and Farrell, 2021). While consistency rules are often deterministic, *non-deterministic consistency rules* are possible in some cases: As an example, →AddAttribute might initialize empty slots for a new attribute with random values for the *first* time, if this random values is kept unchanged for the *following* executions. If different values are used in different executions, the execution loops (correctly) detects branch differences and executes the operator again and again. Therefore, non-determinism is allowed for configurations of methodologists, as long as a *fix-point* for the execution is available. Here, MoConseMI is more pragmatic than other rather formal BX approaches like Hermann, Ehrig et al. (2011), who can guarantee consistency of TGGs for model synchronization only, if the execution of the particular TGGs is deterministic in both directions.

incomplete Consistency Goals

Missing consistency goals due to incomplete knowledge of the domain or the particular project (Klare, Syma et al., 2019) do not break the approach, i. e. MoConseMI assumes the final state to be consistent, but results in inconsistent results in the eyes of users. The same counts, if too much i. e. unnecessary consistency goals are defined, which restrict the set of consistent models too much. In other words, MoConseMI ensures no total consistency, but ensures consistency regarding the realized consistency goals only, as discussed by Abilov, Mahmoud et al. (2015) as well.

14.3 Limitations

This section about limitations shows disadvantages or open parts of the work documented in this thesis. It informs about the reasons for limitations and points to possible solutions. The limitations cover the approach (Section 14.3.1), the implementation of the approach (Section 14.3.2☞495) or their validation (Section 14.3.3☞495).

14.3.1 Limitations of the Approach

This section discusses the limitations of the design of the MoConseMI approach, since "*semantic integration remains an extremely difficult problem*" (Doan and Halevy, 2005, p. 85).

14.3.1.1 Termination

MoConseMI cannot guarantee Termination

MoConseMI cannot guarantee the termination of the execution of an orchestration: This is a general interoperability problem in networks of transformation (Klare, Syma et al., 2019), as they occur in synthetic settings, e. g. it can occur also in Vitruvius (Klare, 2018), due to alternating and diverging loops between alternative transformation paths (Klare, Syma et al., 2019). MoConseMI in contrast avoids this source for non-terminating executions by design, since the tree of configured operators prevents alternating and possibly conflicting transformation paths (Section 6.4.5☞213), as in dense graphs of transformations.

Nevertheless, in MoConseMI there are other reasons for missing termination: In general, multiple executions of operators are required, as shown in Section 6.5☞213. One possible reason are consistency goals and consistency rules which contradict each other or provide no fix-point, as discussed in Section 14.2.2☞489. Another possible reason are inaccurate realizations of consistency goals and consistency rules in detail, with an example in the following box:

Ongoing Example, Part 30: **Termination** ← List →

Since methodologists have a great flexibility for configurations of model decisions with Java, MoConseMI cannot guarantee the termination of the execution of any orchestration. In other words, methodologists have to ensure termination themselves. An example for a missing termination, caused by inaccurate configurations by accident, is demonstrated along Listing 14.1 (which terminates in the shown form) for ❿↔⓫.

```
1  changeModel(new ChangeModelDecision() {
2      @Override
3      public void changeModel(MigrationInformation infos, ...) {
4          // CSV ==> Excel: increment row numbers
5          infos.getAllInstances("data.Requirement").forEach(req ->
       req.set("rowNumber", req.<Integer>get("rowNumber") + 1));
6      }
7  }, new ChangeModelDecision() {
8      @Override
9      public void changeModel(MigrationInformation infos, ...) {
10         // Excel ==> CSV: decrement row numbers
11         infos.getAllInstances("data.Requirement").forEach(req ->
       req.set("rowNumber", req.<Integer>get("rowNumber") - 1));
12     }
13 });
```

Listing 14.1: Configuration of ⇄ChangeModel for ❿↔⓫ in the ongoing Example

Since the row numbers of requirements start with 0 due to its initial CSV format in Requirements, the row numbers are incremented by one by →ChangeModel, since Excel is used for Traceability where row numbers start with 1 (line 5). In contrast, ←ChangeModel decrements each row number by one (line 11). If line 11 for ←ChangeModel would be empty by accident, the execution would not terminate, since →ChangeModel increments the row numbers for each execution, leading to new branch differences, which lead to another execution of →ChangeModel with further increments and so on. Line 11 is important, since it reverts the changes of →ChangeModel, which are required only for Traceability, but should not change the SUM.

A possibility to guarantee termination would be to formalize consistency goals and consistency rules in order to proof the termination before execution or to restrict expressions to guarantee termination by design. Another possibility would be to restrict possible configurations for model decisions, since Java is TURING-complete, e. g. with a dedicated DSL or a dedicated transformation language. Since a DSL might need to be extended for new operators (Section 6.3.2 ☞ 200) and transformation languages have limitations regarding model and metamodel changes, history maps and branch differences (Section 8.3.1 ☞ 267), MoConseMI follows a very pragmatic strategy and allows Java for configuring model decisions, with the disadvantage of missing guarantees for termination.

14.3.1.2 Performance

Decreased *performance* in terms of time in comparison to other approaches for model consistency is a drawback of the MoConseMI approach, since real incrementality is not supported by the operators. In general, the performance and scalability is important for modeling (Kolovos, Tisi et al., 2013) including incrementality for model queries, transformations and multi-view languages. In particular, change-driven techniques with a performance depending on the occurred model changes is required for scalability (Kulkarni, Reddy and Rajbhoj, 2010), which is fulfilled by change translation-based approaches for ensuring model consistency by design. A benefit of MoConseMI compared to change translation-based approaches is its ability to create the initial SUM and to recreate views.

Since the operators of MoConseMI are implemented to be in P and not in NP, MoConseMI outperforms some proof-theory-based approaches which are in NP (Chapter 3 ☞ 93). This holds as long as the configurations for model decisions in MoConseMI are in P. But compared with change translation-based approaches, MoConseMI is model synchronization-based without support for source-incrementality (Section 2.2.3 ☞ 67), i. e. the transformation effort depends on the size of models and not on the size of model changes, while target-incrementality is supported and target changes are preserved by updating models without information loss and providing execution differences for the external adapters.

Execution depends on Size of Models, not on Size of Model Changes

The actual reason, why transformations of operators in MoConseMI cannot support real incrementality by design, depends on the decision to execute operators *in-place* and not out-place, since incrementality is possible only for out-place model transformations with trace links between source model and target model. In-place transformations work only with *one* model, which prevents to create trace links and to transfer some changes from the source model to the target model, since the "target" model does not exist and must be created by updating the complete "source" model by executing the complete in-place transformation.

in-place vs out-place Transformations

Executing operators out-place with incrementality would be very inefficient for longer chains of small transformations, since a vast amount of copy-operations of unchanged model elements is required for each operator leading to poor performance (but only for the first execution) and since *all* intermediate (meta)models must be kept in memory leading to huge memory consumption (for all executions). Here, MoConseMI trades higher execution time for less memory consumption.

Execution Time vs Memory Consumption

When comparing MoConseMI with out-place model transformations without incrementality, MoConseMI outperforms, since the performance of in-place transformations is better, since only changed elements need to be transformed and unchanged elements remain the same without the need to copy them.

in-place vs out-place without Incrementality

Instead of using multiple small transformations in form of chained operators, MoConseMI could ask methodologists to write a single, compact transformation, which could be executed out-place with incrementality, but that would prevent the reuse of predefined operators with intermediate steps in transformations in order to ease the work for metho-

Execution Time vs Reusability for Methodologists

dologists. Here, MoConseMI trades higher execution time for reusability of operators to support methodologists.

Future Work: Towards out-place and incremental Transformations

An idea to improve the execution nevertheless and to overcome these conceptual problems would be to transform a chain of in-place operators into a single out-place model transformation which supports incrementality once before execution. Probably, this requires to restrict operators and their model decisions to enable incrementality, in particular, the handling of UUIDs probably needs an improved design. This idea is future work, since it is a general research question for model transformations. The current in-place execution should be kept as debugging mode for methodologists in order to provide intermediate models.

14.3.1.3 Skills of Stakeholders

Another drawback of MoConseMI is, that high skills are required for some stakeholders. Therefore, the required skills for all four groups of stakeholders are discussed. Additionally, the scalability of MoConseMI regarding its applicability for each stakeholder is discussed.

less Skills required for Users: their Views

The required *skills for users are very low*, since they work with their known views as they are and work directly on the provided concrete renderings in particular technical spaces. Therefore, users do not need to learn modeling or EMF. Users request consistent views, submit changes in commit-based way and get updates for their views to be consistent with other views automatically. The existing data sources with their particular technical spaces are supported by adapters, while the consistency is ensured automatically by the MoConseMI framework according to the configuration of methodologists. Since additional new views can be provided by MoConseMI to support additional concerns of users, the scalability of MoConseMI regarding views for users is very good, while the performance of the consistency preservation at runtime could be improved, as discussed in Section 14.3.1.2☞492.

high Skills required for Methodologists: Domain Knowledge

The required *skills for methodologists are high*, since they need to know the artifacts of the particular development project including the needs for their consistency. On the other hand, they could be supported by users, when discussing and defining the particular consistency goals and consistency rules.

high Skills required for Methodologists: Modeling, EMF

high Skills required for Methodologists: Configurations with MoConseMI

For configuring orchestrations according to project-specific (meta)models and consistency goals, methodologists need (meta)modeling skills in general and knowledge about EMF in particular, since (meta)models are the central artifacts for change propagation. Compared with formal approaches like TGGs, the MoConseMI approach is very pragmatic by guaranteeing less properties in general and providing more flexibility for configurations with known imperative languages like Java. Using Java is easier for configurations than formal specifications, as discussed in Section 8.3.1☞267. Additionally, methodologists are strongly supported by MoConseMI with reusable adapters for other technical spaces, reusable operators, and some reusable predefined configurations for their model decisions. In particular, the operator-based integration of data sources into the SU(M)M and the operator-based definition of new view(point)s benefits from transparency in transformations, since intermediate (meta)models are available for visualizations and for debugging during execution, since debugging is a challenge (Kramer, 2017; Mannadiar and Vangheluwe, 2011). Additionally, operators allow to integrate and extend step-wise, supporting iterative development.

related Ideas supporting Methodologists

Furthermore, related approaches provide several ideas to ease the work of methodologists: Bennani, El Hamlaoui et al. (2018) propose ideas and tool support for multiple methodologists working together with domain experts to distribute the high effort. Sánchez-Cuadrado, de Lara and Guerra (2012) propose ideas to enable modeling for do-

main experts, easing the discussions between methodologists and domain experts. Configuration by-example could be used to reduce the level of abstraction and to enable people with lower modeling skills being methodologists, in analogy to model transformation by-example (Kappel, Langer et al., 2012), DSL by-example (López-Fernández, Garmendia et al., 2019), metamodel by-example (López-Fernández, Cuadrado et al., 2015) and definition of view(point)s by-example (Werner, Wimmer and Aßmann, 2019). In order to find corresponding elements in the metamodels of data sources to integrate, model matching techniques (Somogyi and Asztalos, 2020) could be used. Since their results provide only possible candidates for matches, methodologists have to manually check them before realization.

Scalability of MoConseMI for Methodologists

The *scalability* of MoConseMI regarding the configuration tasks of methodologists at development time is good, since operators ensure scalability with short operator chains for few consistency goals and metamodels with less dependencies, and with longer operator chains for many consistency goals and metamodels with lots of dependencies. The Transformation Tool Contest 2010 (Rose, Herrmannsdoerfer et al., 2012, p. 351), comparing different tools for model co-evolution including classical model transformations, found that in-place transformations provide benefits for methodologists regarding conciseness, since only the differences between old and new versions of (meta)models must be explicitly managed. Reducing redundancies during the integration of data sources reduces the complexity for methodologists, since only the current data source to integrate and the current "intermediate SU(M)M" need to be understood. In particular, all already integrated data sources do not need to be known anymore in detail, since they are completely integrated in the current intermediate SU(M)M. This is the reason for the complexity reduction of projectional approaches (linear complexity) compared to synthetic approaches (square complexity).

very high Skills required for Platform Specialists: Modeling, Consistency, reusable Design, Technical Spaces

The required *skills for platform specialists are very high*, since deep knowlege about modeling for models and metamodels as well as for their evolution and difference representation are required. Additionally, model transformations and their use for ensuring model consistency must be known. In order to ease the work of methodologists, knowledge about reusable design for operators, their decisions and adapters for other technical spaces is required. Nevertheless, the scalability is good, since this effort is spent once for developing MoConseMI in generic way and can be reused and configured by methodologists for multiple applications in practice.

high Skills required for Adapter Providers: Technical Spaces, EMF, Edapt

The required *skills for adapter providers are high*, since they need to know all details of the new technical space and of EMF and Edapt. Additionally, they have to ensure stable UUIDs and to deal with model differences (Section 8.4 ☞271). Experience from practice indicates, that building adapters is challenging (Demuth, Kretschmer et al., 2016, p. 537). Nevertheless, the scalability is good, since this effort is spend once for developing a new adapter, which can be reused by methodologists for multiple applications in practice.

Stakeholders revised

Summarizing, the design of MoConseMI with the resulting skills for the groups of stakeholders follows the principles of separation of concerns and to solve recurring aspects only once in generic way for reuse. Without support, users have to ensure automatable consistency goals manually and themselves. Therefore, MoConseMI shifts the general automation of change propagation to platform specialists who provide mechanisms for ensuring consistency once, while users use and change only their views as manual and ongoing work. Users usually cannot manage all consistency goals within a project, since users usually know only their views according to the idea of multiple views for system development. Therefore, methodologists are introduced as experts for the particular project and for ensuring model consistency in general. Additionally, modeling skills cannot be expected for each user, in contrast to methodologists, who configure the desired consistency goals *once* for each project. Therefore, the methodologist is required only once for the configuration, not for each execution of the operator tree, in contrast to the approach of El Hamlaoui, Bennani et al. (2019). The distinction of the tasks of users and methodologists follows the

principle of separation of concerns, since users concentrate on their development tasks for the particular project, while methodologists manage consistency.

14.3.2 Limitations of the Implementation

There are also some limitations in the MoConseMI framework as the implementation of the MoConseMI approach, which are discussed here. These limitations depend on open or incomplete implementations, while they are fulfilled by the design of MoConseMI in general.

As described in Section 6.5.3☞217, the execution loop for orchestrations requires further conceptual investigations for optimization. For this, the implementation of the execution loop provides some first possibilities to adapt the order to change propagation, which need to be extended for future investigations.

configure Execution Loop

Currently, the Java API for configuring orchestrations (Section 12.1☞455) does not completely support trees of new view(point)s, only independent chains of operators for each new view(point) are supported. Additionally, starting directly with the SU(M)M without any data source is not possible in the present implementation, but can be easily overcome (Section 13.3.2☞474). Nevertheless, both aspects are supported by the MoConseMI design in general.

restricted Java API for configuring Orchestrations

Currently, the implementation does not allow to shutdown the framework after handling user changes and to restart before the next changes of users. In general, the MoConseMI approach supports the offline mode, since all required artifacts for the execution can be persisted and loaded, including models, metamodels, differences (either model-based as EMF models or text-based, see Section 6.7.2☞229) and history maps (Section 6.5.2☞214).

shutdown and restart the Framework

While the performance of the MoConseMI design is discussed in Section 14.3.1.2☞492, the performance of the MoConseMI framework could be improved as well. Supporting huge models could be improved via the proxy-concept or partial loading, as mentioned in Section 2.5.3☞87. The selection of EMF as internal technical space of MoConseMI might be reconsidered, since there are more performant alternatives for EMF and its implementation, e. g. KMF (Fouquet, Nain et al., 2012). Creating visualizations requires additional computation effort and increases performance, but the desired visualizations can be configured. In general, no explicit performance tuning is done for the Java source code of the developed MoConseMI framework.

Performance of the Implementation

14.3.3 Limitations of the Evaluation

While Part IV☞283 shows the general applicability of MoConseMI in practice and Chapter 13☞467 evaluates further aspects of MoConseMI, there might be some limitations of the evaluation itself.

There is no direct comparison with other synthetic or projectional approaches for ensuring inter-model consistency. Such comparisons are hard due to conceptual problems. For example, it must be ensured that the same degree of knowledge about the consistency problem to solve is available and that the experience with the applied approaches is similar. Furthermore, technical problems are challenging, e. g. different technical spaces of the tool support. Regarding projectional SUM approaches, MoConseMI is most similar to Vitruvius, since both use EMF as technical space (Meier, Werner et al., 2020). Additionally, different degrees of maturity of approaches and their tool support might influence the evaluations, e. g. the Vitruvius approach is developed mainly with two PhD projects by Burger (2014) and Kramer (2017) with another just completed PhD project by Klare (2018), complemented with related PhD projects by Goldschmidt (2010) and Langhammer (2017). Nevertheless, there is a higher-level conceptual comparison of SUM approaches

direct Comparisons with other Approaches

done by Meier, Werner et al. (2020) and the contained discussions are already taken up in Section 14.1.1.2 ☞ 484.

Scalability regarding Number of Data Sources

The scalability of MoConseMI regarding the number of data sources to integrate into the SU(M)M is not evaluated in practice, since Wiederhold (1999) mentions enterprise systems with hundreds of data sources and proposes, that SUMM-based approaches will not work in such settings. In theory, redundancies could be reduced during the integration of data sources into the SU(M)M, which enables linear integration effort, as discussed in Section 14.3.1.3 ☞ 493. The same counts for new views, which need to be directly synchronized only with the SUM and not with all other views directly.

same Author of Approach and Evaluation

Finally, the evaluation might be limited, since the development and the evaluation of MoConseMI are done by the same author. Nevertheless, MoConseMI was successfully applied by two Bachelor theses by Michel (2019) and Wegner (2021), supervised by the author of MoConseMI.

14.4 Outlook

This section discusses, how the ideas and works described in this thesis can be extended in future work. Possibilities for future work cover the approach (Section 14.4.1), its implementation (Section 14.4.2 ☞ 497) and additional application domains (Section 14.4.3 ☞ 498).

14.4.1 Outlook for the Approach

Summary of already discussed Future Work

Some possible extensions for the MoConseMI approach are already discussed during its design and implementation. They are collected in the next box as reference and can be summarized as follows: An improved execution order might improve the performance for the execution of orchestrations. Mapping UUIDs, which are internally used by MoConseMI, with IDs in other technical spaces of data sources would support the stability of UUIDs. Adapters for additional technical spaces would ease the application of MoConseMI with the reuse of further existing data sources in other application areas (Section 14.4.3 ☞ 498). An extended transaction management could support the roll-back of user changes and automated follow-up changes as well as concurrent changes of multiple users. Executing operators out-place and incrementally would improve the performance. In addition to these already discussed aspects, further possible extensions are presented in the following.

Future Work: **Summary**

1 Outlook to Future Work 20
2 Improved Execution Order 219
3 UUID Mapping in Adapters 227
4 More Adapters 278
5 Transaction Management 463
6 Towards out-place and incremental Transformations 493
7 Summary 496

dynamic Orchestrations

While the orchestration is configured by methodologists at development time, it is treated as stable after this configuration, and is executed during runtime in order to ensure inter-model consistency, future work could investigate, how "dynamic" orchestrations could be realized, i. e. the orchestration changes at runtime, e. g. with additional operators for additional consistency goals, additional data sources or additional new view(point)s. Dynamic orchestrations are similar to the evolution and reuse of (stable) orchestrations at

development time with similar restrictions, as discussed in Section 13.3.4☞477. In 2014, Mussbacher, Amyot et al. (2014) defined flexible model integration on the fly for current and individual needs as grand challenge for model-driven engineering for the next 30 years.

Inter-Model Consistency of Multi-Level Models

While MoConseMI is restricted to ensure consistency of models, extensions to ensure the consistency of multi-level models might be possible. As preparation, ECore must be replaced by a technical space which supports multi-level modeling. Additionally, this would require to rework the current implementation of MoConseMI in order to support multi-level modeling (Atkinson, Gerbig and Tunjic, 2013a), in particular for the model transformations in the operators (Atkinson, Gerbig and Tunjic, 2013b), the links and associations on different levels (Atkinson, Gerbig and Kuhne, 2015), and even for the identification of elements inside the multiple levels for the configuration of operators (Atkinson and Gerbig, 2014). Finally, model co-evolution could be extended to multi-level model co-evolution, while the difference representation of Section 6.7.2☞229 is a first step towards a multi-level model difference representation.

Integration with Reference Metamodels

While MoConseMI integrates concrete (meta)models into a concrete project-specific SU(M)M, this integration could be done on *reference level*, as motivated by Kurpjuweit and Winter (2007) in their outlook. Motivation for integrations on reference level is the *missing exchangeability of single metamodels* in concrete SUMMs (similar to Section 13.3.1☞473): If Java should be exchanged by C++ in the SUMM for the ongoing example, then all parts of the Java metamodel have to be removed from the SUMM and the C++ metamodel has to be integrated in the SUMM. It might be easier, if only recurring parts of object-oriented programming languages (like classes and methods) are collected as *reference metamodel* for programming languages and this reference metamodel is integrated into a reference SUMM (RSUMM). This might allow to exchange the reference metamodel with concrete metamodels and to reuse the integration for the RSUMM, which is concretized with the concrete metamodel for Java or C++, coming with some language dependent information (like pointer handling). In this way, recurring integration aspects could be realized once in more generic way as well, e. g. for traceability, for which Schwarz, Ebert and Winter (2010) and Seibel, Neumann and Giese (2010) present reference metamodels (Section 14.1.2.1☞486). This approach might ease also the evolution of concrete metamodels (Section 13.3.1☞473), since concrete metamodels could be replaced with the help of the corresponding reference metamodel without affecting the integration of reference metamodels into the RSUMM. This might also improve the modularity of SUMMs, while Klare (2018) follows a similar strategy by aggregating redundant concepts in metamodels similar to reference metamodels here, but with another purpose, i. e. in order to prevent alternating execution paths. First ideas for integrations on reference level are proposed in the following publication:

Related MoConseMI Publication

Johannes Meier and Andreas Winter (2016): *Towards Metamodel Integration Using Reference Metamodels.* In: Proceedings of the 4th Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO 2016), pp. 19–22.

This publication is cited as Meier and Winter (2016) in this thesis.

14.4.2 Outlook for the Implementation

The implementation could be extended with providing a graphical DSL for methodologists for developing orchestrations, as visualized in Figure 6.11☞209, while the current Java API could be seen as internal textual DSL with Java as host language. Additional tool support could be provided for methodologists including better visualizations of intermediate models in model transformation chains (Von Pilgrim, Vanhooff et al., 2008). In order to improve

the performance of the implementation, the internal representations for models could be replaced by other, more efficient implementations (Section 14.3.2 ☞495).

14.4.3 Outlook for further Applications

Introduction

Part IV ☞283 already showed, that MoConseMI is reusable and applicable to different application areas. Therefore, this section sketches some additional application areas for MoConseMI in the future. Before that, some characteristics of MoConseMI are summarized, which indicate suitable application areas, where the benefits of MoConseMI are exploited.

typical Application Areas for MoConseMI

multiple Views on depending and heterogeneous Data

Application areas strongly benefit from MoConseMI, if they jointly combine, manage or change depending and heterogeneous information, since MoConseMI overcomes technical heterogeneity with adapters and structural and semantic heterogeneity with operator-based transformations, which explicitly integrate information into a SU(M)M. An important benefit of MoConseMI is, that its bidirectional operators support transformations in both directions, which enables to keep the original data sources up-to-date. As an example, data ware houses (Section 3.6.3 ☞139) usually combine heterogeneous information from multiple data sources, but do not support to propagate changes back into the original data sources. Therefore, MoConseMI might act as data ware house, but does not provide any benefit in that read-only scenario.

Repository-style vs Transformation Chains

Since MoConseMI uses an explicit SU(M)M, application scenarios in repository-style, where different users, tools or actors work on overlapping information, benefit from MoConseMI, since the data is managed by the explicit SU(M)M in integrated work, while the users, tools and actors are supported with dedicated view(point)s. In contrast, unidirectional chains of transformations, where single transformations provide their output as input for the next transformation, benefit less from MoConseMI, since the different input and output data depend on each other. Even though they could be stored within the SU(M)M with possible trace links, usually no support for consistency is required, since the data are kept consistent by the transformations. In contrast, MoConseMI supports iterative refinements of depending data with support for automated inter-model consistency, which counts in particular for various development projects, where different users and tools work together with heterogeneous and depending data which together form the system under development.

After having identified key aspects where MoConseMI can support most, some possible future application scenarios for MoConseMI are sketched in the following paragraphs.

Software Reengineering

Software reengineering projects might benefit from MoConseMI, as proposed in (Meier, Werner et al., 2020), since MoConseMI enables to reuse legacy artifacts as data sources, to fix initial inconsistencies due to their legacy state, and to update legacy artifacts and write them back into their original technical space. Additionally, new view(point)s could be defined for collecting information from all legacy artifacts for overviews for understanding and for supporting new stakeholders, special analysis tools, and so on. Here, the explicit SU(M)M is beneficial, since software reengineering projects often follow the repository-style, e. g. Kullbach, Winter et al. (1998) and Fuhr, Winter et al. (2012), where using the same technical space is beneficial according to the experience of Bruneliere, Cabot et al. (2015). Mens, Wermelinger et al. (2005) mention the integration of all data involved in software development as challenge in software evolution, since not only the evolution of sourcecode but also the evolution of higher-level artifacts is relevant. Therefore, all involved artifacts can be kept consistent to each other, called "co-evolution" by Mens, Wermelinger et al. (2005). Additionally, manual corrections can be supported by MoConseMI side-by-side with automated corrections by evolution tools, both enabled by new view(point)s.

Enterprise Systems

Since enterprise systems (Section 3.6.5 ☞144) have lots of systems and data bases with depending information, Chen, Doumeingts and Vernadat (2008, p. 656) expect, that mod-

eling techniques will help for enterprise integration. In particular, in enterprises, there are multiple perspectives by various stakeholders, while only simple integrations are realized (Frank, 2014). Therefore, MoConseMI could help to ensure the consistency of data, which are used by different tools and stakeholders as views and which are stored in various data bases as data sources.

Data Interoperability

Data interoperability, i. e. supplying the same data in different formats, is a typical BX scenario (Abou-Saleh, Cheney et al., 2018), since data in one representation is transformed into another representation and vice versa. MoConseMI is beneficial here, since its projectional SUM prevents direct transformations between each combination of two data formats. Instead, only one transformation is required between each data format and a SUM. In the ongoing example, requirements are (slightly adapted) visualized with CSV for Requirements and with Excel for the new Traceability.

Hardware-Software-Systems

The development of hardware-software-systems complement software development projects with additional information and views for the hardware and its combination with the software. Therefore, MoConseMI could be applied also to development projects for hardware-software-systems, in particular, to cyber-physical systems (Persson, Torngren et al., 2013). Following Bucchiarone, Cabot et al. (2020), model-based development for IoT and smart systems is a grand challenge, since models must be integrated with software components and hardware components (Wortmann, Barais et al., 2020). In the embedded systems domain, checking of consistency is one important purpose of models (Liebel, Marko et al., 2018, p. 101), i. e. ensuring consistency is not only a precondition for using depending models, but also a direct benefit. First ideas for the integration of IoT languages are sketched in the following publication:

Related MoConseMI Publication

Muzaffar Artikov, Johannes Meier and Andreas Winter (2019): *Towards Integrated IoT-Languages.* In: 2019 International Conference on Information Science and Communications Technologies (ICISCT).

This publication is cited as Artikov, Meier and Winter (2019) in this thesis.

solve MDA

MoConseMI is suited to realize MDA projects, since the bidirectional operators of MoConseMI support also backward transformations and prevent undesired information loss, which supports the integration and consistency preservation of models with arbitrary levels of abstraction (Section 13.4.2 ☞479). In particular, chains of refinement transformations from abstract models to models representing code could be recreated with operators in MoConseMI in order to support all modeling artifacts in MDA as views (Selic, 2011) on a SU(M)M. For the case of UML, Lucas, Molina and Toval (2009) claimed, that support for views on different levels of abstraction is important, but rarely supported. Compared with the discussions of Zheng and Taylor (2013) regarding model-driven development (MDD), which can be transferred to MDA, MoConseMI realizes *domain-specific* MDD extended with *project*-specific adaptations and with support for round-tripping (since they defined MDD to be unidirectional), and MoConseMI can ensure consistency between models and code.

14.5 Summary of the Thesis

Increasing size, complexity and heterogeneity of software-intensive systems make it nearly impossible that single persons develop a whole system. Therefore, different stakeholders with different concerns are involved and are supported with tailored views on the system. These views conform to viewpoints and enable multi-view modeling of the system under development. Since these views are realized with models and jointly represent the whole

system under development, the models semantically depend on each other in terms of redundant information, explicit links and further constraints, and therefore must be consistent to each other regarding these dependencies. Since the manual ensuring of consistency between models is error-prone, time-consuming and restricted by limited knowledge of users about models of other views, this thesis aims to develop an approach to automatically ensure consistency between multiple models.

MoConseMI is the newly designed and implemented approach of this thesis for automatically ensuring inter-model consistency. Its main and unique characteristic is the reuse of existing metamodels and conforming models as data sources, which are integrated into an explicit Single Underlying (Meta)Model (SU(M)M). This enables to propagate changes between data sources and the SUM in order to re-establish the consistency after changes in any of the models. By this means, MoConseMI supports users of views with automated fixes for inconsistencies, while the desired project-specific consistency goals are configured only once for each project with reusable operators. MoConseMI does not require a formalization of the desired consistency, but provides a pragmatic strategy to initially create a SU(M)M in bottom-up way from existing (meta)models, which is automatically realized by operators that are manually configured to realize consistency. With this strategy, MoConseMI fills a gap in related work by answering the question, how to create a SU(M)M with the reuse of existing view(point)s (Atkinson, Stoll et al., 2013). In MoConseMI, existing data sources are complemented with new view(point)s which can be derived from the SU(M)M and are kept consistent directly with the SUM as well.

The change propagation is realized by a tree of reusable bidirectional operators, which are selected and configured by methodologists. Methodologists configure the tree of bidirectional operators manually and once in a project so that the consistency is automatically fulfilled that is required by the users. The technical heterogeneity of models is overcome by reusable adapters for different technical spaces.

MoConseMI is successfully applied to several application examples including managing distributed access rights, knowledge management and viewpoints for architectures of sensor-based information systems as well as to a strongly simplified software development project as ongoing example. These examples cover applications within and outside of software engineering. This emphasizes that MoConseMI is reusable for and transferable to a broad range of projects, allowing for even more than the presented applications (Section 14.4.3 ☞ 498). Additionally, the evaluation of MoConseMI shows among others, that the designed operators are reusable and reduce the configuration effort for methodologists, that MoConseMI is combinable with other research and into other applications, that MoConseMI can even fulfill intra-model consistency, and that all requirements of this thesis are fulfilled by MoConseMI.

The motivation, design, implementation and evaluation of MoConseMI are supported by an ongoing example, which depicts a software development project in strongly simplified way with depending requirements, class diagrams and Java source code.

Limitations of MoConseMI (Section 14.3 ☞ 490) cover mainly the performance of the in-place change propagation, which depends on the size of models and not on the size of the model changes to propagate, while the design of operators allows the initial creation of the SU(M)M complementing the change propagation.

The possible future works (Section 14.4 ☞ 496) show, that this thesis forms a sound base for keeping diverse models consistent to each other, while the integration of reference level might ease the application of MoConseMI and the evolution of configured operators in similar projects.

MoConseMI complements its main contribution for ensuring inter-model consistency with further contributions for traceability, model co-evolution and difference representations for models and their metamodels.

Final Result

Finally, this thesis developed the MoConseMI *approach* to ensure the consistency between interrelated models automatically. MoConseMI is implemented in a supporting *framework* and is successfully evaluated by several *applications*, showing the reuse and transferability of MoConseMI to different application areas.

This thesis shows, that MoConseMI ensures the consistency between interrelated models automatically. Existing models and metamodels can be reused by MoConseMI transforming them into a Single Underlying (Meta) Model (SU(M)M). New view(point)s can be derived from the SU(M)M and are kept consistent as well.

MoConseMI supports users with automated fixes for inconsistencies, while methodologists need to configure the desired project-specific consistency goals only once for each project with reusable operators. The technical heterogeneity of models is overcome by reusable adapters for different technical spaces.

Part VI

Appendix

This part provides additional supporting information including collected lists for definitions, figures and tables as well as the bibliography with all used literature entries.

Appendix A

Collected Lists

A.1 Parts of the Ongoing Example

1 Typesetting Conventions 20
2 Introduction 25
3 Consistency Challenges 34
4 Alternative Consistency Challenges 35
5 Data Sources 37
6 New View(Point)s 40
7 Applied Concepts as Megamodel 51
8 Modeling 60
9 Used Metamodels and Models 64
10 Multiple possible Fixes 73
11 Consistency Goals and Rules 76
12 Stakeholders 83
13 Concepts of Modeling 90
14 Levels of Heterogeneity 95
15 Which Stakeholders decide? 98
16 Synthetic vs Projectional 101
17 Overview of MoConseMI 172
18 Exemplary Inconsistency Fix 173
19 Initial SUMM and SUM 176
20 Overview of Adapters 178
21 Integration of Data Sources 206
22 Definition of the new View(point) 209
23 Exemplary Execution Order 220
24 CSV Adapter for Requirements 276
25 Configuration by the Methodologist 456
26 Application by Users 462
27 Quality of the SU(M)M 475
28 Intra-Model Consistency 480
29 Related SUM approaches 484
30 Termination 491

A.2 List of Definitions

1 Definition of Definitions 20
2 Consistency 32

3 Artifact . . . 36
4 Data Source . . . 37
5 New View(Point) . . . 40
6 Intra-Model Consistency . . . 42
7 Inter-Model Consistency . . . 42
8 Stakeholder . . . 55
9 Concern . . . 55
10 Viewpoint . . . 55
11 View . . . 56
12 Model . . . 59
13 Metamodel . . . 61
14 Model Transformation . . . 67
15 Consistency Goal . . . 72
16 Consistency Rule . . . 75
17 User (Stakeholder) . . . 80
18 Methodologist (Stakeholder) . . . 81
19 Platform Specialist (Stakeholder) . . . 82
20 Adapter Provider (Stakeholder) . . . 83
21 Technical Space . . . 84
22 Unidirectional Operator . . . 190
23 Bidirectional Operator . . . 192
24 Metamodel Decision . . . 198
25 Model Decision . . . 201
26 Orchestration . . . 213
27 UUID . . . 225

A.3 List of Figures

1.1 Java source code of the ongoing example . . . 39
1.2 Class diagram of the ongoing example . . . 39
1.3 The final concrete syntax of Traceability in Excel format . . . 41

2.1 Concepts as Megamodel, applied to the ongoing Example . . . 52
2.2 Concepts for Stakeholders, Systems and Views . . . 54
2.3 Concepts for Views and their Concrete Syntaxes . . . 57
2.4 Kinds of Views in Multi-Perspective Modeling . . . 57
2.5 Concepts for Views and their Models . . . 58
2.6 The OMG Model Stack (left) applied to represent the ongoing requirements (right) . . . 61
2.7 Multi-level modeling (MLM) applied to represent the ongoing requirements 63
2.8 Metamodel for the data source Requirements . . . 65
2.9 Model for the data source Requirements . . . 65
2.10 Metamodel for the data source Java . . . 65
2.11 Model for the data source Java . . . 66
2.12 Metamodel for the data source UML . . . 66
2.13 Model for the data source UML . . . 66
2.14 Feature Model for Model Transformations (adapted and extended from Czarnecki and Helsen (2006)) . . . 68
2.15 Terminology for Consistency . . . 71
2.16 Concepts for Views and their Consistency . . . 73
2.17 Concepts for Consistency . . . 73

2.18 Exemplary Consistency Relation between source and target models . . . 74
2.19 Overview of Consistency Goals in the ongoing Example 78
2.20 Use Cases of Consistency Management 79
2.21 Relevant Concepts of ECore 88
2.22 Concepts for Stakeholders, Views, Models and Concrete Syntaxes 90
2.23 Modeling concepts of MoConseMI applied to represent the ongoing requirements 91

3.1 Feature Model for classifying functional Objectives of related Approaches 94
3.2 Design Choices for technical Realization 100
3.3 Classification of synthetic vs projectional approaches for the ongoing example 101
3.4 Conceptual designs of different approaches for consistency management . 103
3.5 Kinds of information overlaps between source models (S, always right) and target models (T, always left) as Venn diagrams 104
3.6 Design Choices for selecting one of multiple possible Fixes 106
3.7 Round-Trip Engineering of Model Transformations 116
3.8 Design Choices for conceptual Realization (derived from Meier, Werner et al. (2020)) 124
3.9 SUM approach OSM (taken and slightly adapted from Meier, Werner et al. (2020)) 125
3.10 Exemplary Metamodel for SUMM in OSM (taken from Meier, Werner et al. (2020)) 126
3.11 SUM approach Vitruvius (taken and slightly adapted from Meier, Werner et al. (2020)) 127
3.12 Exemplary Metamodel(s) for SUMM in Vitruvius (taken from Meier, Werner et al. (2020)) 127
3.13 Exemplary Metamodel for SUMM in RSUM (taken and slightly adapted from Meier, Werner et al. (2020)) 130
3.14 SUM approach RSUM (taken from Meier, Werner et al. (2020)) 130

5.1 SUM approach MoConseMI (taken and slightly adapted from Meier, Werner et al. (2020)) 172
5.2 Metamodel for the SUMM (left/top) and model for the SUM (right/bottom) 177
5.3 MoConseMI with Adapters 178
5.4 Design Choices with possible and decided (in gray) Choices fulfilling Requirements 180
5.5 Selected Features for classifying functional Objectives of MoConseMI . 181
5.6 Selected Design Choices for conceptual Realization of MoConseMI . . 182
5.7 Selected Design Choices for technical Realization of MoConseMI . . . 182

6.1 Feature Model for classifying Operator-based Approaches 186
6.2 Main Design of the Operator Signature 190
6.3 Selected Features (light gray) for unidirectional Operators in MoConseMI 191
6.4 Selected Features (light gray) for bidirectional Operators in MoConseMI 192
6.5 Metamodel Decisions for the Operator Signature 197
6.6 Model Decisions for the Operator Signature 201
6.7 Chaining Operators according to the Operator Signature 204
6.8 Operators for the Integration of Data Sources into the SU(M)M 207
6.9 Metamodel Changes from 01 to 02 207
6.10 Metamodel Changes from 03 to 05 (left/top) and 05 to SUMM (right/bottom) 208

6.11 All Operators: Integration and Definition of new View(point)s 209
6.12 Metamodel Changes from SUMM to 06 (left/top), from 06 to 07 (middle) and from 07 to 10 (right/bottom) 211
6.13 Metamodel Changes from 11 to Traceability 212
6.14 Metamodel for the new viewpoint Traceability 212
6.15 Model for the new view Traceability 212
6.16 Execution Information within the Operator Signature 215
6.17 Model Differences and Metamodel Differences within the Operator Signature 216
6.18 Execution Order to update all Views 221
6.19 Concepts of EDAPT to represent EMF Models, similar to Herrmannsdoerfer, Benz and Juergens (2009, Figure 3) 224
6.20 Combining EMF for representing Metamodels and EDAPT for representing Models 225
6.21 Feature Model for classifying functional Features of Difference Representation Approaches 229
6.22 Selected Features for the Difference Representation in MoConseMI . . 230
6.23 Overview about the Model-based Representation of Differences 232
6.24 `ChangeContainer` as central Point to manage Model-based Differences . 233
6.25 Conceptual Idea for the Calculation of Branch Differences 237
6.26 Selected Features of Model Transformations for unidirectional Operators 239
6.27 Selected Features of Model Transformations for the whole MoConseMI approach 239

8.1 MAVEN Projects and Dependencies 264
8.2 Metamodel for the possible Configurations of the EXCEL Adapter 274
8.3 Metamodel for the possible Configurations of the CSV Adapter 276
8.4 Metamodel for the data source Requirements 277
8.5 Model for the data source Requirements 278

9.1 Metamodel of Htpasswd 285
9.2 The initial input model of Htpasswd 285
9.3 Metamodel of Authz 287
9.4 The initial input model of Authz 288
9.5 Metamodel of Htaccess 289
9.6 The initial input model of Htaccess 290
9.7 Metamodel (left/top) and the final model after the initialization (right/bottom) of SUMM 291
9.8 The final concrete syntax of Overview in Excel format 292
9.9 Metamodel of Overview 293
9.10 The final model after the initialization of Overview 294
9.11 Overview about Consistency Goals in Access Data Management 295
9.12 Configured Tree of Operators for Access Data Management 298
9.13 Metamodel Changes from Htpasswd to 01 299
9.14 Model Changes from Htpasswd to 01 299
9.15 Metamodel Changes from Authz to 02 300
9.16 Model Changes from Authz to 02 301
9.17 Changes in Metamodel (left/top) and Model (right/bottom) from 03 to 04 303
9.18 Changes in Metamodel (left/top) and Model (right/bottom) from 04 to 05 305
9.19 Metamodel Changes from Htaccess to 06 306
9.20 Model Changes from Htaccess to 06 307
9.21 Changes in Metamodel (left/top) and Model (right/bottom) from 07 to 08 308

9.22 Changes in Metamodel (left/top) and Model (right/bottom) from 08 to 09 310
9.23 Changes in Metamodel (left/top) and Model (right/bottom) from 09 to SUMM . . . 312
9.24 Changes in Metamodel (left/top) and Model (right/bottom) from SUMM to 10 . . . 314
9.25 Changes in Metamodel (left/top) and Model (right/bottom) from 10 to 11 316
9.26 Changes in Metamodel (left/top) and Model (right/bottom) from 11 to 12 318
9.27 Changes in Metamodel (left/top) and Model (right/bottom) from 12 to 13 320
9.28 Metamodel Changes from 13 to 16 . . . 322
9.29 Metamodel Changes from 16 to 17 . . . 324
9.30 Metamodel Changes from 17 to 18 . . . 324
9.31 Model Changes from 17 to 18 . . . 325
9.32 Metamodel Changes from 18 to Overview . . . 326
9.33 Model Changes from 18 to Overview . . . 327

10.1 Metamodel for the Conceptual Viewpoint . . . 374
10.2 Metamodel for the Module Viewpoint . . . 375
10.3 Metamodel for the Execution Viewpoint . . . 376
10.4 Metamodel for the Code Viewpoint . . . 377
10.5 Metamodel for the Topology Viewpoint . . . 377
10.6 Metamodel for the Data Viewpoint . . . 378
10.7 Overview about the Viewpoints and their Integrations . . . 379
10.8 Operator Orchestration for the technical Integration . . . 382
10.9 Simplified Orchestration for the Definition of two new Viewpoints . . . 383
10.10 Metamodel for the new Viewpoint Intersections . . . 383
10.11 Metamodel for the new Viewpoint ModulesOnly . . . 384

11.1 Metamodel of Work . . . 389
11.2 The initial input model of Work . . . 390
11.3 Metamodel of Employees . . . 391
11.4 The initial input model of Employees . . . 392
11.5 Metamodel of Tasks . . . 393
11.6 The initial input model of Tasks . . . 394
11.7 Metamodel of Materials . . . 395
11.8 The initial input model of Materials . . . 396
11.9 Metamodel of SUMM . . . 397
11.10 The final model after the initialization of SUMM . . . 398
11.11 The final concrete syntax of Costs in Excel format . . . 399
11.12 Metamodel of Costs . . . 399
11.13 The final model after the initialization of Costs . . . 400
11.14 Overview about Consistency Goals in Knowledge Integration . . . 401
11.15 Configured Tree of Operators for Knowledge Integration . . . 403
11.16 Metamodel Changes from Work to 01 . . . 404
11.17 Model Changes from Work to 01 . . . 405
11.18 Model Changes from 01 to 02 . . . 407
11.19 Metamodel Changes from Employees to 03 . . . 408
11.20 Model Changes from Employees to 03 . . . 409
11.21 Metamodel Changes from 04 to 05 . . . 410
11.22 Model Changes from 04 to 05 . . . 411
11.23 Metamodel Changes from 05 to 06 . . . 413
11.24 Metamodel Changes from Tasks to 10 . . . 414

11.25 Model Changes from Tasks to 10 . . . 415
11.26 Metamodel Changes from 10 to 11 . . . 417
11.27 Model Changes from 10 to 11 . . . 418
11.28 Metamodel Changes from 11 to 12 . . . 420
11.29 Metamodel Changes from Materials to 13 . . . 421
11.30 Model Changes from Materials to 13 . . . 422
11.31 Metamodel Changes from 14 to SUMM . . . 423
11.32 Model Changes from 14 to SUMM . . . 424
11.33 Metamodel Changes from SUMM to 20 . . . 425
11.34 Model Changes from SUMM to 20 . . . 426
11.35 Metamodel Changes from 20 to 24 . . . 430
11.36 Model Changes from 20 to 24 . . . 431
11.37 Model Changes from 24 to Costs . . . 433

12.1 Unidirectional Operators of the whole Orchestration in the configured Direction . . . 458
12.2 Unidirectional Operators of the whole Orchestration in the inverse Direction 458

14.1 Conceptual Classification (taken from Meier, Werner et al. (2020)) . . . 484

A.4 List of Tables

1.1 The initial input of Requirements in CSV format . . . 38

3.1 Comparing BX Approaches . . . 112
3.2 Comparing Techniques for Change Propagation . . . 150

9.1 Mapping of Consistency Goals and their Consistency Rules tested in Sections for Access Data Management . . . 369
9.2 Summary of Test Cases for Access Data Management . . . 370

11.1 The initial input of Work in CSV format . . . 388
11.2 The initial input of Employees in CSV format . . . 391
11.3 The initial input of Tasks in CSV format . . . 393
11.4 The initial input of Materials in CSV format . . . 395
11.10 Mapping of Consistency Goals and their Consistency Rules tested in Sections for Knowledge Integration . . . 453
11.11 Summary of Test Cases for Knowledge Integration . . . 454

14.1 Classification of projectional SUM approaches (taken from Meier, Werner et al. (2020)) . . . 485
14.2 Orthogonal Classification of Model Transformations . . . 488

A.5 List of Code Listings

8.1 Examples for the Java API to configure bidirectional Operators . . . 271
9.1 The initial input of Htpasswd in Htpasswd format . . . 284
9.2 The initial input of Authz in Authz format . . . 286
9.3 The initial input of Htaccess in Htaccess format . . . 289
12.1 Orchestration for the ongoing Example with the Java API of MoConseMI 456

12.2 Trigger MoConseMI to propagate User Changes made in the concrete Rendering with CSV of the Requirements View 462
12.3 Change the Model of the View directly in the Edapt Format 462
14.1 Configuration of ⇄ChangeModel for ❿↔⓫ in the ongoing Example . . 491

Bibliography

[Abilov, Mahmoud et al., 2015] Marat Abilov, Tariq Mahmoud, Jorge Marx Gómez and Manuel Mora (2015): *Towards an incremental bidirectional partial model synchronization between organizational and functional requirements models.* In: 5th International Model-Driven Requirements Engineering Workshop, MoDRE 2015 - Proceedings, pp. 1–10. Cited on pages 114 and 490.

[Abou-Saleh, Cheney et al., 2018] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna and Perdita Stevens (2018): *Introduction to Bidirectional Transformations.* vol. 9715, Springer, pp. 1–28. Cited on pages 69, 97, 110, 114, 116, 141, 148 and 499.

[Abouzahra, Bézivin et al., 2005] Anas Abouzahra, Jean Bézivin, Marcos Didonet Del Fabro and Frédéric Jouault (2005): *A practical approach to bridging domain specific languages with UML profiles.* In: Proceedings of the Best. Cited on page 132.

[Acher, Collet et al., 2010] Mathieu Acher, Philippe Collet, Philippe Lahire and Robert France (2010): *Comparing Approaches to Implement Feature Model Composition.* In: LNCS, vol. LNCS 6138, pp. 3–19. Cited on page 187.

[Aivaloglou, Hoepelman and Hermans, 2017] Efthimia Aivaloglou, David Hoepelman and Felienne Hermans (2017): *Parsing Excel formulas: A grammar and its application on 4 large datasets.* In: Journal of Software: Evolution and Process, vol. 29(12), p. e1895. Cited on page 274.

[Altmanninger, Seidl and Wimmer, 2009] Kerstin Altmanninger, Martina Seidl and Manuel Wimmer (2009): *A survey on model versioning approaches.* In: International Journal of Web Information Systems, vol. 5(3), pp. 271–304. Cited on pages 228, 230, 237 and 463.

[Amalio, de Lara and Guerra, 2015] Nuno Amalio, Juan de Lara and Esther Guerra (2015): *Fragmenta: A theory of fragmentation for MDE.* In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 106–115. Cited on page 206.

[Ananieva, Burger and Stier, 2018] Sofia Ananieva, Erik Burger and Christian Stier (2018): *Model-Driven Consistency Preservation in AutomationML.* In: IEEE International Conference on Automation Science and Engineering, vol. 2018-Augus, pp. 1536–1541. Cited on page 129.

[Ananieva, Klare et al., 2018] Sofia Ananieva, Heiko Klare, Erik Burger and Ralf Reussner (2018): *Variants and Versions Management for Models with Integrated Consistency Preservation.* In: Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, pp. 3–10. Cited on page 129.

[Angyal, Lengyel and Charaf, 2008] László Angyal, László Lengyel and Hassan Charaf (2008): *A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering.* In: 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs 2008), IEEE, pp. 463–472. Cited on pages 84 and 117.

[Anjorin, Buchmann et al., 2020] Anthony Anjorin, Thomas Buchmann, Bernhard Westfechtel, Zinovy Diskin, Hsiang-Shang Ko, Romina Eramo, Georg Hinkel, Leila Samimi-Dehkordi and Albert Zündorf (2020): *Benchmarking bidirectional transformations: theory, implementation, application, and assessment.* In: Software and Systems Modeling, vol. 19(3), pp. 647–691. Cited on pages 70, 111, 115, 225 and 469.

[Anjorin and Cheney, 2019] Anthony Anjorin and James Cheney (2019): *Provenance Meets Bidirectional Transformations.* In: 11th International Workshop on the Theory and Practice of Provenance (TaPP 2019). Cited on page 141.

[Anjorin, Leblebici and Schürr, 2016] Anthony Anjorin, Erhan Leblebici and Andy Schürr (2016): *20 years of triple graph grammars: A roadmap for future research.* In: Electronic Communications of the EASST, vol. 73(1), pp. 1–20. Cited on pages 111, 463 and 480.

[Anjorin, Rose et al., 2014] Anthony Anjorin, Sebastian Rose, Frederik Deckwerth and Andy Schürr (2014): *Efficient model synchronization with view triple graph grammars.* In: LNCS, vol. LNCS 8569, pp. 1–17. Cited on page 135.

[Anjorin, Saller et al., 2013] Anthony Anjorin, Karsten Saller, Sebastian Rose and Andy Schürr (2013): *A Framework for Bidirectional Model-to-Platform Transformations.* In: LNCS, vol. LNCS 7745, pp. 124–143. Cited on page 226.

[Antkiewicz and Czarnecki, 2008] Michał Antkiewicz and Krzysztof Czarnecki (2008): *Design Space of Heterogeneous Synchronization.* In: LNCS, vol. LNCS 5235, pp. 3–46. Cited on pages 45, 106 and 463.

[Arcega, Font et al., 2019] Lorena Arcega, Jaime Font, Øystein Haugen and Carlos Cetina (2019): *An approach for bug localization in models using two levels: model and metamodel.* In: Software and Systems Modeling, vol. 18(6), pp. 3551–3576. Cited on page 46.

[Arendt, Biermann et al., 2010] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause and Gabriele Taentzer (2010): *Henshin: Advanced concepts and tools for in-place EMF model transformations.* In: LNCS, vol. LNCS 6394(PART 1), pp. 121–135. Cited on page 268.

[Arora and Gosain, 2011] Meenakshi Arora and Anjana Gosain (2011): *Schema Evolution for Data Warehouse: A Survey.* In: International Journal of Computer Applications, vol. 22(6), pp. 6–14. Cited on page 473.

[Artikov, Meier and Winter, 2019] Muzaffar Artikov, Johannes Meier and Andreas Winter (2019): *Towards Integrated IoT-Languages.* In: 2019 International Conference on Information Science and Communications Technologies (ICISCT). Cited on page 499.

[Asplund and Törngren, 2015] Fredrik Asplund and Martin Törngren (2015): *The discourse on tool integration beyond technology, a literature survey.* In: Journal of Systems and Software, vol. 106, pp. 117–131. Cited on page 44.

[Aßmann, Zschaler and Wagner, 2006] Uwe Aßmann, Steffen Zschaler and Gerd Wagner (2006): *Ontologies, Metamodels, and the Model-Driven Paradigm.* In: Ontologies for Software Engineering and Software Technology, pp. 249–273. Cited on page 142.

[Atkinson, Bostan et al., 2008] Colin Atkinson, Philipp Bostan, Daniel Brenner, Giovanni Falcone, Matthias Gutheil, Oliver Hummel, Monika Juhasz and Dietmar Stoll (2008): *Modeling components and component-based systems in KobrA*. In: LNCS, vol. LNCS 5153, pp. 54–84. Cited on page 126.

[Atkinson and Gerbig, 2013] Colin Atkinson and Ralph Gerbig (2013): *Harmonizing textual and graphical visualizations of domain specific models*. In: Proceedings of the Second Workshop on Graphical Modeling Language Development - GMLD '13, pp. 32–41. Cited on page 126.

[Atkinson and Gerbig, 2014] Colin Atkinson and Ralph Gerbig (2014): *Level-Agnostic Designation of Model Elements*. pp. 18–34. Cited on page 497.

[Atkinson and Gerbig, 2016] Colin Atkinson and Ralph Gerbig (2016): *Flexible deep modeling with melanee*. In: Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft für Informatik (GI), vol. P255, pp. 117–121. Cited on page 126.

[Atkinson, Gerbig and Kühne, 2014] Colin Atkinson, Ralph Gerbig and Thomas Kühne (2014): *Comparing Multi-Level Modeling Approaches*. In: MULTI@MoDELS, pp. 53–61. Cited on page 64.

[Atkinson, Gerbig and Kuhne, 2015] Colin Atkinson, Ralph Gerbig and Thomas Kuhne (2015): *A unifying approach to connections for multi-level modeling*. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 216–225. Cited on page 497.

[Atkinson, Gerbig and Tunjic, 2013a] Colin Atkinson, Ralph Gerbig and Christian Tunjic (2013a): *A multi-level modeling environment for SUM-based software engineering*. In: Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '13, pp. 1–9. Cited on pages 100, 126 and 497.

[Atkinson, Gerbig and Tunjic, 2013b] Colin Atkinson, Ralph Gerbig and Christian Vjekoslav Tunjic (2013b): *Enhancing classic transformation languages to support multi-level modeling*. In: Software and Systems Modeling, pp. 1–22. Cited on pages 125, 126, 271 and 497.

[Atkinson, Kennel and Goß, 2011] Colin Atkinson, Bastian Kennel and Björn Goß (2011): *The Level-Agnostic Modeling Language*. In: Software Language Engineering. SLE 2010. LNCS, vol 6563, vol. LNCS 6563, pp. 266–275. Cited on page 126.

[Atkinson and Kühne, 2001] Colin Atkinson and Thomas Kühne (2001): *The Essence of Multilevel Metamodeling*. In: 01 Proceedings of the 4th International Conference on The Unified Modeling Language Modeling Languages Concepts and Tools, vol. 2185(2185), pp. 19–33. Cited on pages 63 and 85.

[Atkinson and Kühne, 2002a] Colin Atkinson and Thomas Kühne (2002a): *Profiles in a strict metamodeling framework*. In: Science of Computer Programming, vol. 44(1), pp. 5–22. Cited on page 86.

[Atkinson and Kühne, 2002b] Colin Atkinson and Thomas Kühne (2002b): *Rearchitecting the UML infrastructure*. In: ACM Transactions on Modeling and Computer Simulation, vol. 12(4), pp. 290–321. Cited on page 62.

[Atkinson and Kühne, 2003] Colin Atkinson and Thomas Kühne (2003): *Model-driven development: A metamodeling foundation*. In: IEEE Software, vol. 20(5), pp. 36–41. Cited on page 62.

[Atkinson and Kühne, 2008] Colin Atkinson and Thomas Kühne (2008): *Reducing accidental complexity in domain models.* In: Software and Systems Modeling, vol. 7(3), pp. 345–359. Cited on page 62.

[Atkinson and Stoll, 2008a] Colin Atkinson and Dietmar Stoll (2008a): *An environment for the orthographic modeling of workflow components.* In: CEUR Workshop Proceedings, vol. 328. Cited on page 126.

[Atkinson and Stoll, 2008b] Colin Atkinson and Dietmar Stoll (2008b): *Orthographic modeling environment.* In: LNCS, vol. LNCS 4961, pp. 93–96. Cited on page 124.

[Atkinson, Stoll and Bostan, 2009] Colin Atkinson, Dietmar Stoll and Philipp Bostan (2009): *Supporting View-Based Development through Orthographic Software Modeling.* In: Evaluation of Novel Approaches to Software Engineering (ENASE), pp. 71–86. Cited on pages 120, 121 and 463.

[Atkinson, Stoll and Tunjic, 2011] Colin Atkinson, Dietmar Stoll and Christian Tunjic (2011): *Orthographic service modeling.* In: Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC, pp. 67–70. Cited on page 124.

[Atkinson, Stoll et al., 2013] Colin Atkinson, Dietmar Stoll, Christian Tunjic and Jacques Robin (2013): *A Prototype Implementation of an Orthographic Software Modeling Environment.* In: VAO 2013. Cited on pages 126, 163, 485 and 500.

[Atkinson and Tunjic, 2014a] Colin Atkinson and Christian Tunjic (2014a): *Criteria for Orthographic Viewpoints.* In: Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '14, pp. 43–50. Cited on pages 29, 55, 125 and 126.

[Atkinson and Tunjic, 2014b] Colin Atkinson and Christian Tunjic (2014b): *Towards Orthographic Viewpoints for Enterprise Architecture Modeling.* In: 2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, pp. 347–355. Cited on pages 126 and 145.

[Atkinson and Tunjic, 2017] Colin Atkinson and Christian Tunjic (2017): *A Deep View-Point Language for Projective Modeling.* In: Proceedings - 2017 IEEE 21st International Enterprise Distributed Object Computing Conference, EDOC 2017, vol. 2017-Janua, pp. 133–142. Cited on page 125.

[Awadid and Nurcan, 2019] Afef Awadid and Selmin Nurcan (2019): *Consistency requirements in business process modeling: a thorough overview.* In: Software & Systems Modeling, vol. 18(2), pp. 1097–1115. Cited on page 146.

[Aziz Ahmad and Nadeem, 2010] Mahreen Aziz Ahmad and Aamer Nadeem (2010): *Consistency Checking of UML Models using Description Logics: A Critical Review.* In: 2010 6th International Conference on Emerging Technologies (ICET), pp. 310–315. Cited on page 136.

[Babur, Cleophas et al., 2018] Önder Babur, Loek Cleophas, Mark van den Brand, Bedir Tekinerdogan and Mehmet Aksit (2018): *Models, More Models, and Then a Lot More.* In: LNCS, vol. LNCS 10748, pp. 129–135. Cited on page 86.

[Baclawski, Kokar et al., 2002] Kenneth Baclawski, Mieczyslaw M Kokar, Richard Waldinger and Paul A Kogut (2002): *Consistency Checking of Semantic Web Ontologies.* pp. 454–459. Cited on page 142.

[Balzer, 1991] Robert Balzer (1991): *Tolerating Inconsistency.* In: Proceedings of the 13th International Conference on Software Engineering (ICSE), pp. 158–165. Cited on page 461.

[Bancilhon and Spyratos, 1981] F. Bancilhon and N. Spyratos (1981): *Update semantics of relational views.* In: ACM Transactions on Database Systems, vol. 6(4), pp. 557–575. Cited on pages 40 and 141.

[Bank, Buchmann and Westfechtel, 2021] Matthias Bank, Thomas Buchmann and Bernhard Westfechtel (2021): *Combining a Declarative Language and an Imperative Language for Bidirectional Incremental Model Transformations.* In: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 15–27. Cited on pages 111, 112, 149 and 199.

[Barriga, Mandow et al., 2020] Angela Barriga, Lawrence Mandow, José Luis Pérez de la Cruz, Adrian Rutle, Rogardt Heldal and Ludovico Iovino (2020): *A comparative study of reinforcement learning techniques to repair models.* In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–9. Cited on page 107.

[Barriga, Rutle and Heldal, 2019] Angela Barriga, Adrian Rutle and Rogardt Heldal (2019): *Personalized and Automatic Model Repairing using Reinforcement Learning.* In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), (i), pp. 175–181. Cited on pages 46, 461 and 486.

[Batini, Lenzerini and Navathe, 1986] C. Batini, M. Lenzerini and S. B. Navathe (1986): *A comparative analysis of methodologies for database schema integration.* In: ACM Computing Surveys, vol. 18(4), pp. 323–364. Cited on pages 139 and 206.

[Batini and Lenzerini, 1984] Carlo Batini and Maurizio Lenzerini (1984): *A Methodology for Data Schema Integration in the Entity Relationship Model.* In: IEEE Transactions on Software Engineering, vol. SE-10(6), pp. 650–664. Cited on page 139.

[Baumgart, 2010] Andreas Baumgart (2010): *A common meta-model for the interoperation of tools with heterogeneous data models.* In: 3rd Workshop on Model Driven Tool and Process Integration (MDTPI), pp. 31–40. Cited on pages 121, 165 and 476.

[Baumgart and Ellen, 2014] Andreas Baumgart and Christian Ellen (2014): *A Recipe for Tool Interoperability.* In: Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, pp. 300–308. Cited on page 118.

[Becker, Herold et al., 2007] Simon M. Becker, Sebastian Herold, Sebastian Lohmann and Bernhard Westfechtel (2007): *A graph-based algorithm for consistency maintenance in incremental and interactive integration tools.* In: Software & Systems Modeling, vol. 6(3), pp. 287–315. Cited on pages 34 and 114.

[Becker, Koziolek and Reussner, 2007] Steffen Becker, Heiko Koziolek and Ralf Reussner (2007): *Model-Based Performance Prediction with the Palladio Component Model.* In: Proceeding WOSP '07 Proceedings of the 6th international workshop on Software and performance, pp. 54–65. Cited on page 129.

[van Belle, 2003] Jean-Paul W. G. D. van Belle (2003): A framework for the analysis and evaluation of enterprise models. Phd thesis, University of Cape Town. Cited on page 145.

[Bennani, El Hamlaoui et al., 2018] Saloua Bennani, Mahmoud El Hamlaoui, Mahmoud Nassar, Sophie Ebersold and Bernard Coulette (2018): *Collaborative model-based matching of heterogeneous models*. In: 2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD)), pp. 443–448. Cited on pages 82, 118, 490 and 493.

[Berardinelli, Biffl et al., 2015] Luca Berardinelli, Stefan Biffl, Emanuel Maetzler, Tanja Mayerhofer and Manuel Wimmer (2015): *Model-based co-evolution of production systems and their libraries with AutomationML*. In: 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), pp. 1–8. Cited on page 32.

[Bergmann, Ráth et al., 2012] Gábor Bergmann, István Ráth, Gergely Varró and Dániel Varró (2012): *Change-driven model transformations*. In: Software & Systems Modeling, vol. 11(3), pp. 431–461. Cited on page 150.

[Bernardino, Rodrigues and Zorzo, 2016] Maicon Bernardino, Elder M. Rodrigues and Avelino F. Zorzo (2016): *Performance Testing Modeling: an empirical evaluation of DSL and UML-based approaches*. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 1660–1665. Cited on pages 86 and 133.

[Bernstein, 2011] Abraham Bernstein (2011): *Software Engineering and the Semantic Web: A Match Made in Heaven or in Hell?* 200021, pp. 203–205. Cited on page 143.

[Bernstein, 2003] Philip A. Bernstein (2003): *Applying Model Management to Classical Meta Data Problems*. In: Cidr, vol. 2003, pp. 209–220. Cited on pages 140 and 186.

[Bernstein, Madhavan and Rahm, 2011] Philip a Bernstein, Jayant Madhavan and Erhard Rahm (2011): *Generic Schema Matching, Ten Years Later*. In: Pvldb, vol. 4(11), pp. 695–701. Cited on page 139.

[Bernstein and Melnik, 2007] Philip A. Bernstein and Sergey Melnik (2007): *Model management 2.0: Manipulating Richer Mappings*. In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD '07, p. 1. Cited on pages 96 and 140.

[Bertoa, Burgueño et al., 2020] Manuel F. Bertoa, Loli Burgueño, Nathalie Moreno and Antonio Vallecillo (2020): *Incorporating measurement uncertainty into OCL/UML primitive datatypes*. In: Software and Systems Modeling, vol. 19(5), pp. 1163–1189. Cited on page 107.

[Bettini, 2013] Lorenzo Bettini (2013): Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing, Birmingham, 1st edn. Cited on pages 86, 137 and 284.

[Bézivin and Kurtev, 2005] J Bézivin and Ivan Kurtev (2005): *Model-based technology integration with the technical space concept*. In: Metainformatics Symposium, (February). Cited on pages 84 and 226.

[Bézivin, 2005] Jean Bézivin (2005): *On the unification power of models*. In: Software & Systems Modeling, vol. 4(2), pp. 171–188. Cited on pages 27, 58, 60, 62, 68 and 226.

[Bézivin, 2006] Jean Bézivin (2006): *Model Driven Engineering: An Emerging Technical Space*. In: LNCS, vol. 4143, pp. 36–64. Cited on pages 58 and 459.

[Bézivin, Bouzitouna et al., 2006] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Fréderic Jouault, Dimitrios Kolovos, Ivan Kurtev and Richard F. Paige (2006): *A Canonical Scheme for Model Composition*. In: LNCS, vol. LNCS 4066, pp. 346–360. Cited on pages 101 and 102.

[Bézivin, Büttner et al., 2006] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frederic Jouault, Ivan Kurtev and Arne Lindow (2006): *Model Transformations? Transformation Models!* pp. 440–453. Cited on page 68.

[Bézivin, Jouault and Valduriez, 2004] Jean Bézivin, Freédeéric Jouault and Patrick Valduriez (2004): *On the need for megamodels.* In: Workshop on Best Practices for Model-Driven Software Development at the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications., (1), pp. 1–9. Cited on pages 51, 67 and 70.

[Blanc, Mougenot et al., 2009] Xavier Blanc, Alix Mougenot, Isabelle Mounier and Tom Mens (2009): *Incremental Detection of Model Inconsistencies Based on Model Operations.* In: LNCS, vol. LNCS 5565, pp. 32–46. Cited on pages 109 and 117.

[Blouin, Eustache and Diguet, 2014] Dominique Blouin, Yvan Eustache and Jean-Philippe Diguet (2014): *Extensible Global Model Management with Meta-model Subsets and Model Synchronization.* In: 2nd International Workshop On the Globalization of Modeling Languages (GEMOC 2014) co-located with MODELS 2014. Cited on page 114.

[Bork and Karagiannis, 2014] Domenik Bork and Dimitris Karagiannis (2014): *Model-Driven Development of Multi-View Modelling Tools: The MUVIEMOT Approach.* In: Proceedings of the 9th International Conference on Software Paradigm Trends, pp. 1–1. Cited on page 145.

[Bork and Sinz, 2013] Domenik Bork and Elmar J. Sinz (2013): *Bridging the Gap from a Multi-View Modelling Method to the Design of a Multi-View Modelling Tool.* In: Enterprise Modelling and Information Systems Architectures, vol. 8(2), pp. 25–41. Cited on pages 123, 145 and 220.

[Bork, Buchmann and Karagiannis, 2015] Dominik Bork, Robert Buchmann and Dimitris Karagiannis (2015): *Preserving Multi-view Consistency in Diagrammatic Knowledge Representation.* In: LNCS, vol. 9403, pp. 177–182. Cited on page 144.

[Boubakir and Chaoui, 2016] Mohammed Boubakir and Allaoua Chaoui (2016): *A Pairwise Approach for Model Merging.* In: Lecture Notes in Networks and Systems, vol. 1, pp. 327–340. Cited on page 206.

[Bourque and Fairley, 2014] Pierre Bourque and Richard E. Fairley (Eds.) (2014): SWEBOK V3.0: Guide to the Software Engineering Body of Knowledge Version. IEEE Computer Society. Cited on page 32.

[Brambilla, Cabot and Wimmer, 2012] Marco Brambilla, Jordi Cabot and Manuel Wimmer (2012): *Model-Driven Software Engineering in Practice.* In: Synthesis Lectures on Software Engineering, vol. 1(1), pp. 1–182. Cited on page 59.

[Branco, Xiong et al., 2014] Moisés Castelo Branco, Yingfei Xiong, Krzysztof Czarnecki, Jochen Küster and Hagen Völzer (2014): *A case study on consistency management of business and IT process models in banking.* In: Software & Systems Modeling, vol. 13(3), pp. 913–940. Cited on pages 81 and 146.

[Briand, Falessi et al., 2012] Lionel Briand, Davide Falessi, Shiva Nejati, Mehrdad Sabetzadeh and Tao Yue (2012): *Research-based innovation: A tale of three projects in model-driven engineering.* In: LNCS, vol. LNCS 7590, pp. 793–809. Cited on pages 37 and 487.

[Broy, 2018] Manfred Broy (2018): *A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability—from requirements to functional and architectural views*. In: Software & Systems Modeling, vol. 17(2), pp. 365–393. Cited on page 102.

[Broy, Feilkas et al., 2010] Manfred Broy, Martin Feilkas, Markus Herrmannsdoerfer, Stefano Merenda and Daniel Ratiu (2010): *Seamless model-based development: From isolated tools to integrated model engineering environments*. In: Proceedings of the IEEE, vol. 98(4), pp. 526–545. Cited on pages 28, 43, 45, 121, 165, 187 and 473.

[Bruel, Combemale et al., 2020] Jean-Michel Bruel, Benoit Combemale, Esther Guerra, Jean-Marc Jézéquel, Jörg Kienzle, Juan de Lara, Gunter Mussbacher, Eugene Syriani and Hans Vangheluwe (2020): *Comparing and classifying model transformation reuse approaches across metamodels*. In: Software and Systems Modeling, vol. 19(2), pp. 441–465. Cited on pages 171 and 203.

[Brun and Pierantonio, 2008] C Brun and a Pierantonio (2008): *Model differences in the eclipse modeling framework*. In: UPGRADE, The European Journal for the ..., vol. 9(June), pp. 28–34. Cited on pages 227 and 235.

[Bruneliere, Burger et al., 2019] Hugo Bruneliere, Erik Burger, Jordi Cabot and Manuel Wimmer (2019): *A feature-based survey of model view approaches*. In: Software and Systems Modeling, vol. 18(3), pp. 1931–1952. Cited on pages 56, 119, 134 and 485.

[Bruneliere, Cabot et al., 2015] Hugo Bruneliere, Jordi Cabot, Javier Luis Canovas Izquierdo, Leire Orue Echevarria Arrieta, Oliver Strauss and Manuel Wimmer (2015): *Software Modernization Revisited: Challenges and Prospects*. In: Computer, vol. 48(8), pp. 76–80. Cited on page 498.

[Bruneliere, de Kerchove et al., 2018] Hugo Bruneliere, Florent Marchand de Kerchove, Gwendal Daniel and Jordi Cabot (2018): *Towards Scalable Model Views on Heterogeneous Model Resources*. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems - MODELS '18, pp. 334–344. Cited on page 119.

[Bruneliere, de Kerchove et al., 2020] Hugo Bruneliere, Florent Marchand de Kerchove, Gwendal Daniel, Sina Madani, Dimitris Kolovos and Jordi Cabot (2020): *Scalable model views over heterogeneous modeling technologies and resources*. In: Software and Systems Modeling, vol. 19(4), pp. 827–851. Cited on page 119.

[Bruneliere, Perez et al., 2015] Hugo Bruneliere, Jokin Garcia Perez, Manuel Wimmer and Jordi Cabot (2015): *EMF Views: A View Mechanism for Integrating Heterogeneous Models*. In: Monographs of the Society for Research in Child Development, vol. 78, pp. 317–325. Cited on pages 119 and 229.

[Bucchiarone, Cabot et al., 2020] Antonio Bucchiarone, Jordi Cabot, Richard F. Paige and Alfonso Pierantonio (2020): *Grand challenges in model-driven engineering: an analysis of the state of the research*. In: Software and Systems Modeling, vol. 19(1), pp. 5–13. Cited on pages 27, 28, 30, 37, 189 and 499.

[Buchmann and Westfechtel, 2014] Thomas Buchmann and Bernhard Westfechtel (2014): *Mapping feature models onto domain models: ensuring consistency of configured domain models*. In: Software & Systems Modeling, vol. 13(4), pp. 1495–1527. Cited on page 135.

[Buchmann and Westfechtel, 2016] Thomas Buchmann and Bernhard Westfechtel (2016): *Using triple graph grammars to realise incremental round-trip engineering.* In: IET Software, vol. 10(6), pp. 173–181. Cited on page 149.

[Buneman and Tan, 2019] Peter Buneman and Wang-Chiew Tan (2019): *Data Provenance: What next?* In: ACM SIGMOD Record, vol. 47(3), pp. 5–16. Cited on page 141.

[Burden, 2014] Håkan Burden (2014): *Putting the Pieces Together – Technical, Organisational and Social Aspects of Language Integration for Complex Systems.* In: 2nd International Workshop On the Globalization of Modeling Languages (GEMOC 2014) co-located with MODELS 2014, pp. 17–22. Cited on page 479.

[Burden, Heldal and Whittle, 2014] Håkan Burden, Rogardt Heldal and Jon Whittle (2014): *Comparing and contrasting model-driven engineering at three large companies.* In: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14, pp. 1–10. Cited on pages 27, 29 and 30.

[Burger, 2013a] Erik Burger (2013a): *Flexible views for rapid model-driven development.* In: Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '13, pp. 1–5. Cited on page 128.

[Burger and Gruschko, 2010] Erik Burger and Boris Gruschko (2010): *A Change Metamodel for the Evolution of MOF-Based Metamodels.* In: Proceedings of Modellierung 2010. Cited on pages 229 and 241.

[Burger, Henss et al., 2014] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse and Lucia Happe (2014): *View-based model-driven software development with ModelJoin.* In: Software & Systems Modeling. Cited on page 128.

[Burger, Mittelbach and Koziolek, 2016] Erik Burger, Victoria Mittelbach and Anne Koziolek (2016): *View-based and model-driven outage management for the smart grid.* In: CEUR Workshop Proceedings, vol. 1742, pp. 1–8. Cited on page 129.

[Burger and Schneider, 2016] Erik Burger and Oliver Schneider (2016): *Translatability and Translation of Updated Views in ModelJoin.* vol. 6707, pp. 55–69. Cited on pages 119 and 129.

[Burger, 2013b] Erik Johannes Burger (2013b): *Flexible views for view-based model-driven development.* In: Proceedings of the 18th international doctoral symposium on Components and architecture - WCOP '13, p. 25. Cited on page 128.

[Burger, 2014] Erik Johannes Burger (2014): Flexible Views for View-based Model-driven Development. KIT Scientific Publishing, Karlsruhe. Cited on pages 128, 229 and 495.

[Burgueño, Cabot and Gérard, 2019] Loli Burgueño, Jordi Cabot and Sébastien Gérard (2019): *The Future of Model Transformation Languages: An Open Community Discussion.* In: The Journal of Object Technology, vol. 18(3), p. 7:1. Cited on page 268.

[Canovas Izquierdo, Cosentino and Cabot, 2017] Javier Luis Canovas Izquierdo, Valerio Cosentino and Jordi Cabot (2017): *An Empirical Study on the Maturity of the Eclipse Modeling Ecosystem.* In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 292–302. Cited on page 86.

[Ceri, Valle et al., 2012] Stefano Ceri, Emanuele Della Valle, Dino Pedreschi and Roberto Trasarti (2012): *Mega-modeling for Big Data Analytics.* In: Conceptual Modeling, pp. 1–15. Cited on page 92.

[Chandra and Gupta, 2018] Pravin Chandra and Manoj K. Gupta (2018): *Comprehensive survey on data warehousing research.* In: International Journal of Information Technology, vol. 10(2), pp. 217–224. Cited on page 140.

[Chechik, Nejati and Sabetzadeh, 2012] Marsha Chechik, Shiva Nejati and Mehrdad Sabetzadeh (2012): *A relationship-based approach to model integration.* In: Innovations Syst Softw Eng, vol. 8(123), pp. 3–18. Cited on pages 131, 186 and 470.

[Chen, Doumeingts and Vernadat, 2008] David Chen, Guy Doumeingts and François Vernadat (2008): *Architectures for enterprise integration and interoperability: Past, present and future.* In: Computers in Industry, vol. 59(7), pp. 647–659. Cited on page 498.

[Cheney, Chong et al., 2009] James Cheney, Stephen Chong, Nate Foster, Margo Seltzer and Stijn Vansummeren (2009): *Provenance: A Future History.* In: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications - OOPSLA '09, vol. 64(3), p. 957. Cited on page 141.

[Cheney, Gibbons et al., 2017] James Cheney, Jeremy Gibbons, James McKinna and Perdita Stevens (2017): *On principles of Least Change and Least Surprise for bidirectional transformations.* In: The Journal of Object Technology, vol. 16(1), p. 3:1. Cited on pages 106, 107, 114 and 147.

[Choi, Song and Han, 2006] Namyoun Choi, Il-Yeol Song and Hyoil Han (2006): *A survey on ontology mapping.* In: ACM SIGMOD Record, vol. 35(3), pp. 34–41. Cited on page 143.

[Cicchetti, Ciccozzi and Leveque, 2011] Antonio Cicchetti, Federico Ciccozzi and Thomas Leveque (2011): *A hybrid approach for multi-view modeling.* In: Recent Advances in Multi-paradigm Modeling, vol. 50. Cited on page 134.

[Cicchetti, Ciccozzi and Leveque, 2012] Antonio Cicchetti, Federico Ciccozzi and Thomas Leveque (2012): *Supporting incremental synchronization in hybrid multi-view modelling.* In: LNCS, vol. LNCS 7167, pp. 89–103. Cited on pages 134, 227 and 228.

[Cicchetti, Ciccozzi et al., 2011] Antonio Cicchetti, Federico Ciccozzi, Thomas Leveque and Alfonso Pierantonio (2011): *On the concurrent versioning of metamodels and models: challenges and possible solutions.* In: Proceedings of the 2nd International Workshop on Model Comparison in Practice, pp. 16–25. Cited on page 229.

[Cicchetti, Ciccozzi et al., 2012] Antonio Cicchetti, Federico Ciccozzi, Silvia Mazzini, Stefano Puri, Marco Panunzio, Alessandro Zovi and Tullio Vardanega (2012): *CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems.* In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012, p. 362. Cited on page 122.

[Cicchetti, Di Ruscio et al., 2008] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo and Alfonso Pierantonio (2008): *Automating co-evolution in model-driven engineering.* In: EDOC'08. 12th. Cited on pages 85, 195 and 241.

[Cicchetti, Di Ruscio et al., 2011] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo and Alfonso Pierantonio (2011): *JTL: A Bidirectional and Change Propagating Transformation Language.* In: LNCS, vol. LNCS 6563, pp. 183–202. Cited on page 112.

[Cicchetti, Di Ruscio and Pierantonio, 2007] Antonio Cicchetti, Davide Di Ruscio and Alfonso Pierantonio (2007): *A metamodel independent approach to difference representation.* In: Journal of Object Technology, vol. 6(9), pp. 165–185. Cited on page 228.

[Ciccozzi, Tichy et al., 2019] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe and Danny Weyns (2019): *Blended Modelling - What, Why and How.* In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 425–430. Cited on page 138.

[Clasen, Jouault and Cabot, 2011] Cauê Clasen, Frédéric Jouault and Jordi Cabot (2011): *VirtualEMF: A model virtualization tool.* In: LNCS, vol. LNCS 6999, pp. 332–335. Cited on page 119.

[Clements, Garlan et al., 2002] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers and Reed Little (2002): Documenting Software Architectures: Views and Beyond. Addison-Wesley Professional, Bosten, 1st edn. Cited on page 56.

[Collins-Sussman, Fitzpatrick and Pilato, 2011] Ben Collins-Sussman, Brian W. Fitzpatrick and C. Michael Pilato (2011): *Server Configuration: Path-Based Authorization.* Cited on pages 285 and 286.

[Corrêa, 2011] Chessman K. F. Corrêa (2011): *Towards automatic consistency preservation for model-driven software product lines.* In: Proceedings of the 15th International Software Product Line Conference on - SPLC '11, p. 1. Cited on page 135.

[Cuadra, Martínez et al., 2013] Dolores Cuadra, Paloma Martínez, Elena Castro and Harith Al-Jumaily (2013): *Guidelines for representing complex cardinality constraints in binary and ternary relationships.* In: Software & Systems Modeling, vol. 12(4), pp. 871–889. Cited on page 147.

[Cunha, Garis and Riesco, 2015] Alcino Cunha, Ana Garis and Daniel Riesco (2015): *Translating between Alloy specifications and UML class diagrams annotated with OCL.* In: Software & Systems Modeling, vol. 14(1), pp. 5–25. Cited on pages 109 and 111.

[Cunha, Fernandes et al., 2012] Jacome Cunha, Joao Paulo Fernandes, Jorge Mendes and Joao Saraiva (2012): *MDSheet: A framework for model-driven spreadsheet engineering.* In: 2012 34th International Conference on Software Engineering (ICSE), pp. 1395–1398. Cited on page 272.

[Czarnecki and Helsen, 2006] K. Czarnecki and S. Helsen (2006): *Feature-based survey of model transformation approaches.* In: IBM Systems Journal, vol. 45(3), pp. 621–645. Cited on pages 67, 68, 70 and 506.

[Czarnecki, Foster et al., 2009] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr and James F. Terwilliger (2009): *Bidirectional Transformations: A Cross-Discipline Perspective.* pp. 260–283. Cited on page 110.

[Dam, Egyed et al., 2016] Hoa Khanh Dam, Alexander Egyed, Michael Winikoff, Alexander Reder and Roberto E. Lopez-Herrejon (2016): *Consistent merging of model versions.* In: Journal of Systems and Software, vol. 112, pp. 137–155. Cited on pages 107, 141, 214 and 464.

[Darke and Shanks, 1996] Peta Darke and Graeme Shanks (1996): *Stakeholder viewpoints in requirements definition: A framework for understanding viewpoint development approaches.* In: Requirements Engineering, vol. 1(2), pp. 88–105. Cited on pages 31, 55, 97 and 131.

[Dayal and Bernstein, 1982] Umeshwar Dayal and Philip A. Bernstein (1982): *On the correct translation of update operations on relational views.* In: ACM Transactions on Database Systems, vol. 7(3), pp. 381–416. Cited on pages 40, 113 and 141.

[Debiasi, Ihirwe et al., 2021] Alberto Debiasi, Felicien Ihirwe, Pierluigi Pierini, Silvia Mazzini and Stefano Tonetta (2021): *Model-based Analysis Support for Dependable Complex Systems in CHESS*. In: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 262–269. Cited on page 122.

[Debreceni, Horváth et al., 2014] Csaba Debreceni, Ákos Horváth, Ábel Hegedüs, Zoltán Ujhelyi, István Ráth and Dániel Varró (2014): *Query-driven incremental synchronization of view models*. In: Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '14, pp. 31–38. Cited on page 120.

[Degueule, Combemale et al., 2015] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais and Jean-Marc Jézéquel (2015): *Melange: A Meta-language for Modular and Reusable Development of DSLs*. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 25–36. Cited on page 187.

[Del Fabro, Bézivin et al., 2005] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault and Patrick Valduriez (2005): *Applying Generic Model Management to Data Mapping*. In: Proceedings of the Journes Bases de Donnes Avances (BDA05). Cited on page 102.

[Del Fabro and Jouault, 2005] Marcos Didonet Del Fabro and Frédéric Jouault (2005): *Model Transformation and Weaving in the AMMA Platform*. In: Proceedings of GTTSE 2006. Cited on pages 102 and 170.

[Del Fabro and Valduriez, 2009] Marcos Didonet Del Fabro and Patrick Valduriez (2009): *Towards the efficient development of model transformations using model weaving and matching transformations*. In: Software and Systems Modeling, vol. 8(3), pp. 305–324. Cited on page 102.

[Delen, Dalal and Benjamin, 2005] Dursun Delen, Nikunj P. Dalal and Perakath C. Benjamin (2005): *Integrated modeling: the key to holistic understanding of the enterprise*. In: Communications of the ACM, vol. 48(4), pp. 107–112. Cited on page 145.

[Delplanque, Etien et al., 2018] Julien Delplanque, Anne Etien, Nicolas Anquetil and Olivier Auverlot (2018): *Relational Database Schema Evolution: An Industrial Case Study*. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 635–644. Cited on page 195.

[Demuth, Kretschmer et al., 2016] Andreas Demuth, Roland Kretschmer, Alexander Egyed and Davy Maes (2016): *Introducing Traceability and Consistency Checking for Change Impact Analysis across Engineering Tools in an Automation Solution Company: An Experience Report*. In: 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 529–538. Cited on pages 110 and 494.

[Demuth, Lopez-Herrejon and Egyed, 2015] Andreas Demuth, Roberto Erick Lopez-Herrejon and Alexander Egyed (2015): *Constraint-driven modeling through transformation*. In: Software & Systems Modeling, vol. 14(2), pp. 573–596. Cited on pages 81, 98, 113 and 490.

[Demuth, Riedl-Ehrenleitner et al., 2016] Andreas Demuth, Markus Riedl-Ehrenleitner, Roberto E. Lopez-Herrejon and Alexander Egyed (2016): *Co-evolution of metamodels and models through consistent change propagation*. In: Journal of Systems and Software, vol. 111, pp. 281–297. Cited on page 194.

[Demuth, Riedl-Ehrenleitner et al., 2015] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman and Alexander Egyed (2015): *DesignSpace – An Infrastructure for Multi-User/Multi-Tool Engineering.* In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 1486–1491. Cited on page 109.

[Di Rocco, Di Ruscio et al., 2015] Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino and Alfonso Pierantonio (2015): *Collaborative Repositories in Model-Driven Engineering.* In: IEEE Software, vol. 32(3), pp. 28–34. Cited on page 480.

[Di Ruscio, Iovino and Pierantonio, 2012a] Davide Di Ruscio, Ludovico Iovino and Alfonso Pierantonio (2012a): *Coupled Evolution in Model-Driven Engineering.* In: IEEE Software, vol. 29(6), pp. 78–84. Cited on page 187.

[Di Ruscio, Iovino and Pierantonio, 2012b] Davide Di Ruscio, Ludovico Iovino and Alfonso Pierantonio (2012b): *Evolutionary togetherness: How to manage coupled evolution in metamodeling ecosystems.* In: LNCS, vol. LNCS 7562, pp. 20–37. Cited on page 188.

[Di Ruscio, Lämmel and Pierantonio, 2011] Davide Di Ruscio, Ralf Lämmel and Alfonso Pierantonio (2011): *Automated Co-evolution of GMF Editor Models.* In: LNCS, vol. LNCS 6563, pp. 143–162. Cited on pages 138 and 188.

[Dijkman, Quartel and van Sinderen, 2008] Remco M. Dijkman, Dick A.C. Quartel and Marten J. van Sinderen (2008): *Consistency in multi-viewpoint design of enterprise information systems.* In: Information and Software Technology, vol. 50(7-8), pp. 737–752. Cited on pages 73, 76 and 145.

[Dimitrieski, 2017] Vladimir Dimitrieski (2017): Model-Driven Technical Space Integration Based on a Mapping Approach. Phd thesis, University of Novi Sad, Serbia. Cited on page 271.

[Diskin, Gholizadeh et al., 2016] Zinovy Diskin, Hamid Gholizadeh, Arif Wider and Krzysztof Czarnecki (2016): *A three-dimensional taxonomy for bidirectional model synchronization.* In: Journal of Systems and Software, vol. 111, pp. 298–322. Cited on pages 84, 97, 104, 105 and 463.

[Diskin and König, 2016] Zinovy Diskin and Harald König (2016): *Incremental Consistency Checking of Heterogeneous Multimodels. LNCS*, vol. 9946, Springer, Cham, pp. 274–288. Cited on page 103.

[Diskin, König and Lawford, 2018] Zinovy Diskin, Harald König and Mark Lawford (2018): *Multiple Model Synchronization with Multiary Delta Lenses.* In: Science of Computer Programming, vol. 40, Springer, pp. 21–37. Cited on pages 76, 116, 149, 206, 220 and 225.

[Diskin, Xiong and Czarnecki, 2010] Zinovy Diskin, Yingfei Xiong and Krzysztof Czarnecki (2010): *From State- to Delta-Based Bidirectional Model Transformations.* pp. 61–76. Cited on pages 115 and 227.

[Diskin, Xiong and Czarnecki, 2011] Zinovy Diskin, Yingfei Xiong and Krzysztof Czarnecki (2011): *From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case.* In: The Journal of Object Technology, vol. 10. Cited on pages 115, 125, 202, 225 and 228.

[Diskin, Xiong et al., 2011] Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann and Fernando Orejas (2011): *From state- to delta-based bidirectional model transformations: The symmetric case.* In: LNCS, vol. LNCS 6981, pp. 304–318. Cited on pages 102, 115 and 214.

[Djuric, Gaševic and Devedžic, 2006] Dragan Djuric, Dragan Gaševic and Vladan Devedžic (2006): *The Tao of Modeling Spaces.* In: The Journal of Object Technology, vol. 5(8), p. 125. Cited on pages 84 and 85.

[Doan and Halevy, 2005] An Hai Doan and Alon Y. Halevy (2005): *Semantic-integration research in the database community: A brief survey.* In: AI Magazine, vol. 26(1), pp. 83–94. Cited on pages 139 and 490.

[Doan, Halevy and Ives, 2012] AnHai Doan, Alon Halevy and Zachary Ives (2012): Principles of Data Integration. Elsevier. Cited on pages 95, 141 and 480.

[Ebert and Horn, 2014] Jürgen Ebert and Tassilo Horn (2014): *GReTL: An extensible, operational, graph-based transformation language.* In: Software and Systems Modeling, vol. 13(1), pp. 301–321. Cited on page 193.

[Ebert, Riediger and Winter, 2008] Jürgen Ebert, Volker Riediger and Andreas Winter (2008): *Graph Technology in Reverse Engineering: The TGraph Approach.* In: Rainer Gimnich, Uwe Kaiser, Jochen Quante and Andreas Winter (Eds.): Software archeology and the handbook of software architecture, Gesellschaft für Informatik e. V., Bonn, pp. 67–81. Cited on page 85.

[Ebert, Kluge and Götz, 2021] Sebastian Ebert, Tim Kluge and Sebastian Götz (2021): *Resolving synchronization conflicts in role-based multimodel-synchronization environments.* In: Proceedings of the 13th ACM International Workshop on Context-Oriented Programming and Advanced Modularity, pp. 1–8. Cited on page 131.

[Egyed, Letier and Finkelstein, 2008] Alexander Egyed, Emmanuel Letier and Anthony Finkelstein (2008): *Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models.* In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 99–108. Cited on page 136.

[Egyed, Zeman et al., 2018] Alexander Egyed, Klaus Zeman, Peter Hehenberger and Andreas Demuth (2018): *Maintaining Consistency across Engineering Artifacts.* In: Computer, vol. 51(2), pp. 28–35. Cited on pages 30, 42, 76, 109, 132, 476 and 480.

[Ehrig, Ehrig et al., 2014] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann and Gabriele Taentzer (2014): *Information Preserving Bidirectional Model Transformations.* In: Fundamental Approaches to Software Engineering, Springer, Berlin, Heidelberg, pp. 72–86. Cited on page 217.

[Ehrig, Ehrig et al., 2008] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel and Ulrike Prange (2008): *Consistent Integration of Models Based on Views of Visual Languages.* In: Fundamental Approaches to Software Engineering (FASE 2008), LNCS, vol. 4961, pp. 62–76. Cited on page 134.

[Ehrig, Ermel et al., 2015a] Hartmut Ehrig, Claudia Ermel, Ulrike Golas and Frank Hermann (2015a): *Enterprise Modelling and Model Integration.* pp. 327–349. Cited on pages 144 and 146.

[Ehrig, Ermel et al., 2015b] Hartmut Ehrig, Claudia Ermel, Ulrike Golas and Frank Hermann (2015b): Graph and Model Transformation. Monographs in Theoretical Computer Science. An EATCS Series, Springer, Berlin, Heidelberg, monographs edn. Cited on pages 68 and 86.

[El Hamlaoui, Bennani et al., 2019] Mahmoud El Hamlaoui, Saloua Bennani, Sophie Ebersold, Mahmoud Nassar and Bernard Coulette (2019): *AHM: Handling Heterogeneous Models Matching and Consistency via MDE.* In: Communications in Computer and Information Science, vol. 1023, Springer, pp. 288–313. Cited on pages 82, 117, 118, 121 and 494.

[El Hamlaoui, Trojahn et al., 2014] Mahmoud El Hamlaoui, Cassia Trojahn, Sophie Ebersold and Bernard Coulette (2014): *Towards an Ontology-based Approach for Heterogeneous Model Matching.* In: 2nd International Workshop On the Globalization of Modeling Languages (GEMOC 2014) co-located with MODELS 2014. Cited on page 143.

[Engels, Heckel et al., 2002] Gregor Engels, Reiko Heckel, Jochen M. Küster and Luuk Groenewegen (2002): *Consistency-Preserving Model Evolution through Transformations.* In: LNCS, vol. LNCS 2460, pp. 212–227. Cited on page 102.

[Engels, Küster et al., 2001] Gregor Engels, Jochem M. Küster, Reiko Heckel and Luuk Groenewegen (2001): *A methodology for specifying and analyzing consistency of object-oriented behavioral models.* In: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-9, p. 186. Cited on pages 32, 33 and 43.

[Eramo, Marinelli and Pierantonio, 2014] Romina Eramo, Romeo Marinelli and Alfonso Pierantonio (2014): *Towards a taxonomy for bidirectional transformation.* In: CEUR Workshop Proceedings, vol. 1354, pp. 122–131. Cited on pages 114 and 218.

[Eramo, Pierantonio and Rosa, 2015] Romina Eramo, Alfonso Pierantonio and Gianni Rosa (2015): *Managing uncertainty in bidirectional model transformations.* In: SLE 2015 - Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 49–58. Cited on page 112.

[Eramo, Pierantonio and Tucci, 2018] Romina Eramo, Alfonso Pierantonio and Michele Tucci (2018): *Enhancing the JTL tool for bidirectional transformations.* In: Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, vol. Part F1376, pp. 36–41. Cited on pages 112 and 216.

[Erdweg, Storm et al., 2013] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth and Jimi van ver Woning (2013): *The state of the art in language workbenches: Conclusions from the language workbench challenge.* In: Proceedings of the 6th International Conference on Software Language Engineering (SLE 2013), LNCS, vol. LNCS 8225, pp. 197–217. Cited on pages 85 and 137.

[Estublier, Ionita et al., 2009] Jacky Estublier, Anca Daniela Ionita, Thomas Leveque and Tam Nguyen (2009): *Bi-dimensional Composition with Domain Specific Languages.* In: e-Informatica Software Engineering Journal, vol. 3(1), pp. 27–41. Cited on page 133.

[Estublier, Vega and Ionita, 2005] Jacky Estublier, German Vega and Anca Daniela Ionita (2005): *Composing domain-specific languages for wide-scope software engineering applications.* In: LNCS, vol. LNCS 3713, pp. 69–83. Cited on page 133.

[Etien, Muller et al., 2015] Anne Etien, Alexis Muller, Thomas Legrand and Richard F Paige (2015): *Localized model transformations for building large-scale transformations.* In: Software & Systems Modeling, vol. 14(3), pp. 1189–1213. Cited on page 204.

[Favre and Nguyen, 2005] Jean Marie Favre and Tam Nguyen (2005): *Towards a megamodel to model software evolution through transformations.* In: Electronic Notes in Theoretical Computer Science, vol. 127(3), pp. 59–74. Cited on pages 71 and 78.

[Feldmann, Herzig et al., 2015a] Stefan Feldmann, Sebastian J. I. Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan J. J. Paredis and Birgit Vogel-Heuser (2015a): *A comparison of inconsistency management approaches using a mechatronic manufacturing system design case study.* In: 2015 IEEE International Conference on Automation Science and Engineering (CASE), vol. 2015-Octob, pp. 158–165. Cited on pages 102, 103, 105, 109, 135, 165 and 461.

[Feldmann, Herzig et al., 2015b] Stefan Feldmann, Sebastian J.I. Herzig, Konstantin Kernschmidt, Thomas Wolfenstetter, Daniel Kammerl, Ahsan Qamar, Udo Lindemann, Helmut Krcmar, Christiaan J.J. Paredis and Birgit Vogel-Heuser (2015b): *Towards Effective Management of Inconsistencies in Model-Based Engineering of Automated Production Systems.* In: IFAC-PapersOnLine, vol. 48(3), pp. 916–923. Cited on pages 28, 143 and 455.

[Feldmann, Wimmer et al., 2016] Stefan Feldmann, Manuel Wimmer, Konstantin Kernschmidt and Birgit Vogel-Heuser (2016): *A comprehensive approach for managing intermodel inconsistencies in automated production systems engineering.* In: IEEE International Conference on Automation Science and Engineering, vol. 2016-Novem, pp. 1120–1127. Cited on pages 28, 100, 118 and 165.

[Fernandes, Li et al., 2009] Ronald Fernandes, Biyan Li, Perakath Benjamin and Richard Mayer (2009): *Collaboration support for executable enterprise architectures.* In: 2009 International Symposium on Collaborative Technologies and Systems, pp. 520–527. Cited on page 145.

[Fernandes, Li et al., 2010] Ronald Fernandes, Biyan Li, Perakath Benjamin and Richard Mayer (2010): *Applying semantic technologies for Enterprise Application Integration.* In: 2010 International Symposium on Collaborative Technologies and Systems, pp. 416–422. Cited on page 145.

[Finkelstein, Gabbay et al., 1993] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh (1993): *Inconsistency handling in multi-perspective specifications.* In: LNCS, vol. LNCS 717, pp. 84–99. Cited on pages 102, 202 and 461.

[Finkelstein, Kramer and Goedicke, 1990] Anthony Finkelstein, Jeff Kramer and Michael Goedicke (1990): *ViewPoint Oriented Software Development.* In: 3rd International Workshop Software Engineering and its Applications, (December 1990), pp. 1–21. Cited on pages 31 and 55.

[Fondement, Muller et al., 2013] Frédéric Fondement, Pierre Alain Muller, Laurent Thiry, Brice Wittmann and Germain Forestier (2013): *Big metamodels are evil: Package unmerge - A technique for downsizing metamodels.* In: LNCS, vol. LNCS 8107, pp. 138–153. Cited on page 476.

[Foster, Greenwald et al., 2007] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce and Alan Schmitt (2007): *Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem.* In: ACM Transactions on Programming Languages and Systems, vol. 29(3), p. 17. Cited on pages 85, 114, 116 and 141.

[Fouquet, Nain et al., 2012] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau and Jean-Marc Jézéquel (2012): *An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements.* In: LNCS, vol. LNCS 7590, pp. 87–101. Cited on page 495.

[France and Rumpe, 2007] R. France and B. Rumpe (2007): *Model-driven Development of Complex Software: A Research Roadmap.* In: Future of Software Engineering (FOSE '07). Cited on pages 28, 30, 40, 122, 143, 166, 473 and 476.

[Francis, Kolovos et al., 2013] Martiņš Francis, Dimitrios S. Kolovos, Nicholas Matragkas and Richard F. Paige (2013): *Adding spreadsheets to the MDE toolkit.* In: LNCS, vol. LNCS 8107, pp. 35–51. Cited on page 272.

[Frank, 2014] Ulrich Frank (2014): *Multi-perspective enterprise modeling: Foundational concepts, prospects and future research challenges.* In: Software and Systems Modeling, vol. 13(3), pp. 941–962. Cited on pages 28, 145 and 499.

[Franzago, Ruscio et al., 2018] Mirco Franzago, Davide Di Ruscio, Ivano Malavolta and Henry Muccini (2018): *Collaborative Model-Driven Software Engineering: A Classification Framework and a Research Map.* In: IEEE Transactions on Software Engineering, vol. 44(12), pp. 1146–1175. Cited on page 135.

[Fuhr, Winter et al., 2012] Andreas Fuhr, Andreas Winter, Uwe Erdmenger, Tassilo Horn, Uwe Kaiser, Volker Riediger and Werner Teppe (2012): *Model-driven software migration: Process model, tool support, and application.* In: Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments, pp. 153–184. Cited on page 498.

[Gabmeyer, Kaufmann et al., 2019] Sebastian Gabmeyer, Petra Kaufmann, Martina Seidl, Martin Gogolla and Gerti Kappel (2019): *A feature-based classification of formal verification techniques for software models.* In: Software & Systems Modeling, vol. 18(1), pp. 473–498. Cited on pages 102 and 136.

[Garcés, Vara et al., 2014] Kelly Garcés, Juan M. Vara, Frédéric Jouault and Esperanza Marcos (2014): *Adapting transformations to metamodel changes via external transformation composition.* In: Software & Systems Modeling, vol. 13(2), pp. 789–806. Cited on page 187.

[García, Diaz and Azanza, 2013] Jokin García, Oscar Diaz and Maider Azanza (2013): *Model Transformation Co-evolution: A Semi-automatic Approach.* In: Software Language Engineering (SLE 2012), LNCS, vol. 7745, pp. 144–163. Cited on page 187.

[Garis, Paiva et al., 2012] Ana Garis, Ana C. R. Paiva, Alcino Cunha and Daniel Riesco (2012): *Specifying UML Protocol State Machines in Alloy.* In: LNCS, vol. LNCS 7321, pp. 312–326. Cited on page 109.

[Gašević, Kaviani and Hatala, 2007] Dragan Gašević, Nima Kaviani and Marek Hatala (2007): *On Metamodeling in Megamodels.* In: Model Driven Engineering Languages and Systems, pp. 91–105. Cited on page 64.

[Giachetti, 2004] Ronald E. Giachetti (2004): *A framework to review the information integration of the enterprise.* In: International Journal of Production Research, vol. 42(6), pp. 1147–1166. Cited on pages 144 and 145.

[Giese and Wagner, 2009] Holger Giese and Robert Wagner (2009): *From model transformation to incremental bidirectional model synchronization.* In: Software and Systems Modeling, vol. 8(1), pp. 21–43. Cited on pages 113 and 114.

[Giunchiglia, Shvaiko and Yatskevich, 2005] Fausto Giunchiglia, Pavel Shvaiko and Mikalai Yatskevich (2005): *Semantic schema matching.* In: LNCS, vol. LNCS 3760, pp. 347–365. Cited on page 139.

[Goldschmidt, 2010] Thomas Goldschmidt (2010): View-Based Textual Modelling. KIT Scientific Publishing, Karlsruhe. Cited on pages 225 and 495.

[Goldschmidt, Becker and Burger, 2012] Thomas Goldschmidt, Steffen Becker and Erik Burger (2012): *Towards a Tool-Oriented Taxonomy of View-Based Modelling.* In: Modellierung 2012, pp. 59–74. Cited on pages 29, 42, 56 and 157.

[Gonzalez-Perez and Henderson-Sellers, 2008] Cesar Gonzalez-Perez and Brian Henderson-Sellers (2008): Metamodelling for Software Engineering. John Wiley & Sons, Chichester, West Sussex. Cited on page 64.

[Gorp, Altheide and Janssens, 2006] Pieter Van Gorp, Frank Altheide and Dirk Janssens (2006): *Traceability and Fine-Grained Constraints in Interactive Inconsistency Management.* In: Science, (June 2014), pp. 1–13. Cited on pages 34 and 44.

[Götz, Tichy and Groner, 2021] Stefan Götz, Matthias Tichy and Raffaela Groner (2021): *Claimed advantages and disadvantages of (dedicated) model transformation languages: a systematic literature review.* In: Software and Systems Modeling, vol. 20(2), pp. 469–503. Cited on pages 171, 189 and 267.

[Götz, Tichy and Kehrer, 2021] Stefan Götz, Matthias Tichy and Timo Kehrer (2021): *Dedicated Model Transformation Languages vs. General-purpose Languages: A Historical Perspective on ATL vs. Java.* In: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 122–135. Cited on page 267.

[Greenyer and Kindler, 2010] Joel Greenyer and Ekkart Kindler (2010): *Comparing relational model transformation technologies: implementing Query/View/Transformation with Triple Graph Grammars.* In: Software & Systems Modeling, vol. 9(1), pp. 21–46. Cited on page 111.

[Greenyer, Pook and Rieke, 2011] Joel Greenyer, Sebastian Pook and Jan Rieke (2011): *Preventing information loss in incremental model synchronization by reusing elements.* In: LNCS, vol. LNCS 6698, pp. 144–159. Cited on page 114.

[Grundy, Mugridgett and Hosking, 1998] J.C. Grundy, W.B. Mugridgett and J.G. Hosking (1998): *Visual specification of multi-view visual environments.* In: Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No.98TB100254), vol. 1998-Septe, pp. 236–243. Cited on page 138.

[Grundy, Hosking and Mugridge, 1998] John Grundy, John Hosking and W.B. Mugridge (1998): *Inconsistency management for multiple-view software development environments.* In: IEEE Transactions on Software Engineering, vol. 24(11), pp. 960–981. Cited on pages 29 and 461.

[Grundy, Hosking et al., 2013] John C. Grundy, John Hosking, Karen Na Li, Norhayati Mohd Ali, Jun Huh and Richard Lei Li (2013): *Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications*. In: IEEE Transactions on Software Engineering, vol. 39(4), pp. 487–515. Cited on page 138.

[Gruschko, Kolovos and Paige, 2007] Boris Gruschko, Dimitrios S. Kolovos and Richard F. Paige (2007): *Towards Synchronizing Models with Evolving Metamodels*. In: Workshop on Model-Driven Software Evolution (MODSE), 11th European Conference on Software Maintenance and Reengineering. Cited on pages 85 and 196.

[Guerra, Diaz and de Lara, 2005] Esther Guerra, P. Diaz and J. de Lara (2005): *A Formal Approach to the Generation of Visual Language Environments Supporting Multiple Views*. In: 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), vol. 2005, pp. 284–286. Cited on pages 121 and 134.

[Guerra and de Lara, 2006] Esther Guerra and Juan de Lara (2006): *Model View Management with Triple Graph Transformation Systems*. pp. 351–366. Cited on pages 31, 102 and 134.

[Guerra and de Lara, 2014] Esther Guerra and Juan de Lara (2014): *Colouring: execution, debug and analysis of QVT-relations transformations through coloured Petri nets*. In: Software & Systems Modeling, vol. 13(4), pp. 1447–1472. Cited on page 111.

[Guerra, de Lara et al., 2010] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos and Richard F. Paige (2010): *Inter-modelling: From theory to practice*. In: LNCS, vol. LNCS 6394(PART 1), pp. 376–391. Cited on page 113.

[Guerra, de Lara and Orejas, 2013] Esther Guerra, Juan de Lara and Fernando Orejas (2013): *Inter-modelling with patterns*. In: Software & Systems Modeling, vol. 12(1), pp. 145–174. Cited on page 113.

[Gupta, Mumick and Subrahmanian, 1993] Ashish Gupta, Inderpal Singh Mumick and V. S. Subrahmanian (1993): *Maintaining views incrementally*. In: ACM SIGMOD Record, vol. 22(2), pp. 157–166. Cited on page 140.

[Gupta and Mumick, 2006] Himanshu Gupta and Inderpal Singh Mumick (2006): *Incremental maintenance of aggregate and outerjoin expressions*. In: Information Systems, vol. 31(6), pp. 435–464. Cited on page 140.

[Haase and Stojanovic, 2005] Peter Haase and Ljiljana Stojanovic (2005): *Consistent Evolution of OWL Ontologies*. pp. 182–197. Cited on page 142.

[Haesen and Snoeck, 2005] Raf Haesen and Monique Snoeck (2005): *Implementing Consistency Management Techniques for Conceptual Modeling*. In: Proceedings of the 3rd International Workshop on Consistency Problems in UML-Based Software Development, III Understanding and Usage of Dependency Relationships (at the International Conference on Unified Modeling Language (UML2004)). Cited on pages 105 and 121.

[Hailpern and Tarr, 2006] B. Hailpern and P. Tarr (2006): *Model-driven development: The good, the bad, and the ugly*. In: IBM Systems Journal, vol. 45(3), pp. 451–461. Cited on pages 29, 479 and 487.

[Hakimpour and Geppert, 2001] Farshad Hakimpour and Andreas Geppert (2001): *Resolving Semantic Heterogeneity in Schema Integration: an Ontology Based Approach*. In: Proceedings of the international conference on Formal Ontology in Information Systems - FOIS '01, vol. 2001, pp. 297–308. Cited on page 143.

[Halevy, Ashish et al., 2005] Alon Y. Halevy, Naveen Ashish, Dina Bitton, Michael Carey, Denise Draper, Jeff Pollock, Arnon Rosenthal and Vishal Sikka (2005): *Enterprise information integration: Successes, Challenges and Controversies*. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data - SIGMOD '05, vol. 29(2), p. 778. Cited on pages 144, 145, 472 and 474.

[Hardebolle and Boulanger, 2009] Cécile Hardebolle and Frédéric Boulanger (2009): *Exploring Multi-Paradigm Modeling Techniques*. In: SIMULATION, vol. 85(11-12), pp. 688–708. Cited on page 45.

[Hearnden, Lawley and Raymond, 2006] David Hearnden, Michael Lawley and Kerry Raymond (2006): *Incremental Model Transformation for the Evolution of Model-Driven Systems*. pp. 321–335. Cited on page 106.

[Hebig, Khelladi and Bendraou, 2017] Regina Hebig, Djamel Eddine Khelladi and Reda Bendraou (2017): *Approaches to co-evolution of metamodels and models: A survey*. In: IEEE Transactions on Software Engineering, vol. 43(5), pp. 396–414. Cited on pages 193 and 199.

[Hegedus, Horvath et al., 2011] A. Hegedus, A. Horvath, I. Rath, Moisés Castelo Branco and D. Varro (2011): *Quick fix generation for DSMLs*. In: 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 17–24. Cited on page 110.

[Heidenreich, Johannes et al., 2009] Florian Heidenreich, Jendrik Johannes, Mirko Seifert and Christian Wende (2009): *Closing the Gap between Modelling and Java*. In: Software Language Engineering, vol. LNCS 5969, pp. 374–383. Cited on pages 27 and 96.

[Hein, Ritter and Wagner, 2009] Christian Hein, Tom Ritter and Michael Wagner (2009): *Model-Driven Tool Integration with ModelBus*. In: Proceedings of the 1st International Workshop on Future Trends of Model-Driven Development, pp. 35–39. Cited on page 121.

[Helms, 2020] Sören Helms (2020): Kombination von Datenbanken durch den Ansatz der Integration. Bachelor thesis, Carl von Ossietzky University of Oldenburg, Germany. Cited on pages 139 and 140.

[Hermann, Ehrig et al., 2012] Frank Hermann, Hartmut Ehrig, Claudia Ermel and Fernando Orejas (2012): *Concurrent model synchronization with conflict resolution based on triple graph grammars*. In: LNCS, vol. LNCS 7212, pp. 178–193. Cited on page 463.

[Hermann, Ehrig et al., 2011] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin and Yingfei Xiong (2011): *Correctness of Model Synchronization Based on Triple Graph Grammars*. In: LNCS, vol. LNCS 6981, pp. 668–682. Cited on pages 111, 114 and 490.

[Hermann, Ehrig et al., 2015] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann and Thomas Engel (2015): *Model synchronization based on triple graph grammars: correctness, completeness and invertibility*. In: Software & Systems Modeling, vol. 14(1), pp. 241–269. Cited on pages 111 and 116.

[Herrmann, Voigt et al., 2018] Kai Herrmann, Hannes Voigt, Torben Bach Pedersen and Wolfgang Lehner (2018): *Multi-schema-version data management: data independence in the twenty-first century*. In: The VLDB Journal, vol. 27(4), pp. 547–571. Cited on page 195.

[Herrmannsdoerfer, 2010] Markus Herrmannsdoerfer (2010): *COPE - A workbench for the coupled evolution of metamodels and models.* In: LNCS, vol. LNCS 6563, pp. 286–295. Cited on page 195.

[Herrmannsdoerfer, Benz and E, 2008] Markus Herrmannsdoerfer, Sebastian Benz and E (2008): *COPE: a language for the coupled evolution of metamodels and models.* In: Proc. 1st International Workshop on Model Co-evolution and Consistency Management (MCCM'08), Toulouse, France, Sept. 30, 2008, pp. 1–15. Cited on page 195.

[Herrmannsdoerfer, Benz and Juergens, 2009] Markus Herrmannsdoerfer, Sebastian Benz and Elmar Juergens (2009): *COPE : Coupled Evolution of Metamodels and Models for the Eclipse Modeling Framework.* In: ECOOP 2009, LNCS vol. 5653, vol. LNCS 5653, pp. 1–10. Cited on pages 195, 222, 224 and 508.

[Herrmannsdoerfer and Koegel, 2010] Markus Herrmannsdoerfer and Maximilian Koegel (2010): *Towards a generic operation recorder for model evolution.* In: Proceedings of the 1st International Workshop on Model Comparison in Practice - IWMCP '10, p. 76. Cited on pages 228 and 235.

[Herrmannsdoerfer, Ratiu and Wachsmuth, 2010] Markus Herrmannsdoerfer, Daniel Ratiu and Guido Wachsmuth (2010): *Language Evolution in Practice: The History of GMF.* In: LNCS, vol. LNCS 5969, pp. 3–22. Cited on page 188.

[Herrmannsdoerfer, Vermolen and Wachsmuth, 2011] Markus Herrmannsdoerfer, Sander D. Vermolen and Guido Wachsmuth (2011): *An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models.* In: Software Language Engineering, vol. LNCS 6563, pp. 163–182. Cited on pages 195, 241, 470 and 471.

[Hesse, 2002] Wolfgang Hesse (2002): *Ontologie(n).* In: Informatik-Spektrum, pp. 477–480. Cited on page 142.

[Hesse and Mayr, 2008] Wolfgang Hesse and Heinrich C. Mayr (2008): *Modellierung in der Softwaretechnik: eine Bestandsaufnahme.* In: Informatik-Spektrum, vol. 31(5), pp. 377–393. Cited on pages 59, 62 and 64.

[Hettel, Lawley and Raymond, 2008] Thomas Hettel, Michael Lawley and Kerry Raymond (2008): *Model Synchronisation: Definitions for Round-Trip Engineering.* In: Theory and Practice of Model Transformations, Springer, Berlin, Heidelberg, pp. 31–45. Cited on pages 73, 106, 116, 117, 152 and 202.

[Hettel, Lawley and Raymond, 2009] Thomas Hettel, Michael Lawley and Kerry Raymond (2009): *Towards model round-trip engineering: An abductive approach.* In: LNCS, vol. LNCS 5563, pp. 100–115. Cited on page 117.

[Hidaka, Tisi et al., 2016] Soichiro Hidaka, Massimo Tisi, Jordi Cabot and Zhenjiang Hu (2016): *Feature-based classification of bidirectional transformation approaches.* In: Software & Systems Modeling, vol. 15(3), pp. 907–928. Cited on pages 70, 111, 216, 469 and 486.

[Hildebrandt, Lambers et al., 2013] Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin and Andy Schürr (2013): *A Survey of Triple Graph Grammar Tools.* In: Eceasst, vol. 57, pp. 1–18. Cited on pages 70 and 111.

[Hinkel and Burger, 2018] Georg Hinkel and Erik Burger (2018): *On the Influence of Metamodel Design to Analyses and Transformations.* vol. 7949, Springer, pp. 63–79. Cited on page 475.

[Hinkel and Burger, 2019] Georg Hinkel and Erik Burger (2019): *Change propagation and bidirectionality in internal transformation DSLs*. In: Software and Systems Modeling, vol. 18(1), pp. 249–278. Cited on page 271.

[Hinkel, Heinrich and Reussner, 2019] Georg Hinkel, Robert Heinrich and Ralf Reussner (2019): *An extensible approach to implicit incremental model analyses*. In: Software & Systems Modeling, vol. 18(5), pp. 3151–3187. Cited on page 120.

[Hofmeister, Nord and Soni, 2000] Christine Hofmeister, Robert L. Nord and Dilip Soni (2000): Applied Software Architecture. Addison-Wesley Professional. Cited on pages 122, 373, 384 and 488.

[Horst, Bachmann and Hesse, 2012] Benjamin Horst, Andrej Bachmann and Wolfgang Hesse (2012): *Ontologien als ein Mittel zur Wiederverwendung von Domänen-spezifischem Wissen in der Software- Entwicklung*. In: Modellierung 2012, pp. 171–186. Cited on page 142.

[Huzar, Kuzniarz et al., 2005] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio and Jean Louis Sourrouille (2005): *Consistency Problems in UML-Based Software Development*. In: LNCS, vol. 3297, pp. 1–12. Cited on pages 39, 42 and 43.

[Iacob, Meertens et al., 2012] M. E. Iacob, L. O. Meertens, H. Jonkers, D. A. C. Quartel, L. J. M. Nieuwenhuis and M. J. van Sinderen (2012): *From enterprise architecture to business models and back*. In: Software & Systems Modeling, pp. 1059–1083. Cited on page 146.

[IEEE, 2011] IEEE (2011): *ISO/IEC/IEEE 42010:2011 - Systems and software engineering - Architecture description*. Cited on pages 27, 53, 54, 55, 56, 57, 100, 101, 121 and 384.

[IEEE Standards Board, 1990] IEEE Standards Board (1990): IEEE Standard Glossary of Software Engineering Terminology. Tech. rep. Cited on page 32.

[Jackson, 2019] Daniel Jackson (2019): *Alloy: A Language and Tool for Exploring Software Designs*. In: Communications of the ACM, vol. 62(9), pp. 66–76. Cited on page 109.

[Jahed, Bagherzadeh and Dingel, 2021] Karim Jahed, Mojtaba Bagherzadeh and Juergen Dingel (2021): *On the benefits of file-level modularity for EMF models*. In: Software and Systems Modeling, vol. 20(1), pp. 267–286. Cited on pages 88 and 222.

[Jakob, Königs and Schürr, 2006] Johannes Jakob, Alexander Königs and Andy Schürr (2006): *Non-materialized Model View Specification with Triple Graph Grammars*. In: LNCS, vol. LNCS 4178, pp. 321–335. Cited on pages 135 and 156.

[Jakumeit, Buchwald et al., 2014] Edgar Jakumeit, Sebastian Buchwald, Dennis Wagelaar, Li Dan, Ábel Hegedüs, Markus Herrmannsdörfer, Tassilo Horn, Elina Kalnina, Christian Krause, Kevin Lano, Markus Lepper, Arend Rensink, Louis Rose, Sebastian Wätzoldt and Steffen Mazanek (2014): *A survey and comparison of transformation tools based on the transformation tool contest*. In: Science of Computer Programming, vol. 85(PART A), pp. 41–99. Cited on pages 68 and 85.

[Jelschen, 2015] Jan Jelschen (2015): *Service-oriented toolchains for software evolution*. In: 2015 IEEE 9th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA), pp. 51–58. Cited on page 44.

[Jelschen, 2024] Jan Jelschen (2024): Software Evolution Services. Phd thesis, University of Oldenburg. Cited on pages 58, 59, 67, 85, 163 and 226.

[Jézéquel, 2008] Jean-Marc Jézéquel (2008): *Model driven design and aspect weaving.* In: Software & Systems Modeling, vol. 7(2), pp. 209–218. Cited on pages 102 and 132.

[Jézéquel, Combemale et al., 2015] Jean Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus and François Fouquet (2015): *Mashup of metalanguages and its implementation in the Kermeta language workbench.* In: Software and Systems Modeling, vol. 14(2), pp. 905–920. Cited on page 138.

[Jin, Cordy and Dean, 2002] D. Jin, J.R. Cordy and T.R. Dean (2002): *Where's the schema? A taxonomy of patterns for software exchange.* In: Proceedings 10th International Workshop on Program Comprehension, pp. 65–74. Cited on pages 36, 45, 226 and 489.

[Jin, Cordy and Dean, 2003] D. Jin, J.R. Cordy and T.R. Dean (2003): *Transparent reverse engineering tool integration using a conceptual transaction adapter.* In: Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings., (March), pp. 399–408. Cited on pages 143 and 165.

[Jouault, Allilaire et al., 2008] Frédéric Jouault, Freddy Allilaire, Jean Bézivin and Ivan Kurtev (2008): *ATL: A model transformation tool.* In: Science of Computer Programming, vol. 72(1-2), pp. 31–39. Cited on pages 85, 110, 113 and 126.

[Jouault and Bézivin, 2006] Frédéric Jouault and Jean Bézivin (2006): *KM3: A DSL for metamodel specification.* In: LNCS, vol. LNCS 4037, pp. 171–185. Cited on page 85.

[Jukšs, Verbrugge et al., 2018] Māris Jukšs, Clark Verbrugge, Maged Elaasar and Hans Vangheluwe (2018): *Scope in model transformations.* In: Software & Systems Modeling, vol. 17(4), pp. 1227–1252. Cited on page 222.

[Jung, Heinrich et al., 2014] Reiner Jung, Robert Heinrich, Eric Schmieders, Misha Strittmatter and Wilhelm Hasselbring (2014): *A Method for Aspect-oriented Meta-Model Evolution.* In: Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '14, pp. 19–22. Cited on page 132.

[Kahani, Bagherzadeh et al., 2019] Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel and Daniel Varró (2019): *Survey and classification of model transformation tools.* In: Software & Systems Modeling, vol. 18(4), pp. 2361–2397. Cited on pages 68, 85, 86, 110, 149 and 479.

[Kainz, Buckl and Knoll, 2011] Gerd Kainz, Christian Buckl and Alois Knoll (2011): *Automated Model-to-Metamodel Transformations Based on the Concepts of Deep Instantiation.* pp. 17–31. Cited on page 186.

[Kainz, Buckl and Knoll, 2012] Gerd Kainz, Christian Buckl and Alois Knoll (2012): *A Generic Approach Simplifying Model-to-Model Transformation Chains.* In: Transformation, pp. 579–594. Cited on pages 185 and 190.

[Kang, Cohen et al., 1990] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak and A. Spencer Peterson (1990): Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep., Carnegie-Mellon University, Software Engineering Institute. Cited on page 22.

[Kappel, Kapsammer et al., 2006] G.a Kappel, E.b Kapsammer, H.a Kargl, G.a Kramler, T.b Reiter, W.b Retschitzegger, W.c Schwinger and M.a Wimmer (2006): *On models and ontologies - A layered approach for model-based tool integration.* In: Modellierung 2006, pp. 11–27. Cited on pages 121 and 143.

[Kappel, Langer et al., 2012] Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger and Manuel Wimmer (2012): *Model Transformation By-Example: A Survey of the First Wave*. In: LNCS, vol. LNCS 7260, pp. 197–215. Cited on page 494.

[Karsai, 2014] Gabor Karsai (2014): *Unification or integration? The Challenge of Semantics in Heterogeneous Modeling Languages*. In: 2nd International Workshop On the Globalization of Modeling Languages (GEMOC 2014) co-located with MODELS 2014. Cited on page 486.

[Kateule, 2019] Ruthbetha Kateule (2019): Reference Architecture for Smart Environmental Information Systems. Phd thesis, University of Oldenburg. Cited on pages 373, 374, 383 and 384.

[Kats, Visser and Wachsmuth, 2010] Lennart C.L. Kats, Eelco Visser and Guido Wachsmuth (2010): *Pure and Declarative Syntax Definition: Paradise Lost and Regained*. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10), pp. 918–932. Cited on page 138.

[Kehrer, Taentzer et al., 2016] Timo Kehrer, Gabriele Taentzer, Michaela Rindt and Udo Kelter (2016): *Automatically Deriving the Specification of Model Editing Operations from Meta-Models*. In: LNCS, vol. 9765, pp. 173–188. Cited on page 46.

[Kensche and Quix, 2007] David Kensche and Christoph Quix (2007): *Transformation of Models in(to) a Generic Metamodel*. In: BTW Workshops, pp. 4–15. Cited on page 187.

[Kern and Kühne, 2007] Heiko Kern and Stefan Kühne (2007): *Model interchange between aris and eclipse emf*. In: 7th OOPSLA Workshop on Domain-Specific .... Cited on page 146.

[Kessentini, Sahraoui and Wimmer, 2016] Wael Kessentini, Houari Sahraoui and Manuel Wimmer (2016): *Automated Metamodel/Model Co-evolution Using a Multi-objective Optimization Approach*. vol. 7949, pp. 138–155. Cited on page 194.

[Khelladi, Hebig et al., 2016] Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, Jacques Robin and Marie-Pierre Gervais (2016): *Metamodel and Constraints Co-evolution: A Semi Automatic Maintenance of OCL Constraints*. In: Georgia M. Kapitsaki and Eduardo Santana de Almeida (Eds.): CEUR Workshop Proceedings, *LNCS*, vol. 9679, Springer, Cham, pp. 333–349. Cited on page 187.

[Kienzle, Al Abed and Jacques, 2009] Jörg Kienzle, Wisam Al Abed and Klein Jacques (2009): *Aspect-oriented multi-view modeling*. In: Proceedings of the 8th ACM international conference on Aspect-oriented software development - AOSD '09, p. 87. Cited on pages 132 and 213.

[Kienzle, Mussbacher et al., 2016] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien DeAntoni, Jacques Klein and Bernhard Rumpe (2016): *VCU: The Three Dimensions of Reuse*. In: LNCS, vol. 9679, pp. 122–137. Cited on page 36.

[Kienzle, Mussbacher et al., 2019] Jörg Kienzle, Gunter Mussbacher, Benoit Combemale and Julien Deantoni (2019): *A unifying framework for homogeneous model composition*. In: Software & Systems Modeling, vol. 18(5), pp. 3005–3023. Cited on page 206.

[Klare, 2018] Heiko Klare (2018): *Multi-model consistency preservation.* In: Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 156–161. Cited on pages 128, 151, 166, 491, 495 and 497.

[Klare and Gleitze, 2019] Heiko Klare and Joshua Gleitze (2019): *Commonalities for Preserving Consistency of Multiple Models.* In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 371–378. Cited on pages 128, 476 and 478.

[Klare, Kramer et al., 2021] Heiko Klare, Max E. Kramer, Michael Langhammer, Dominik Werle, Erik Burger and Ralf Reussner (2021): *Enabling consistency in view-based system development — The Vitruvius approach.* In: Journal of Systems and Software, vol. 171, p. 110815. Cited on pages 126, 129 and 166.

[Klare, Syma et al., 2019] Heiko Klare, Torsten Syma, Erik Burger and Ralf Reussner (2019): *A categorization of interoperability issues in networks of transformations.* In: Journal of Object Technology, vol. 18(3), pp. 1–20. Cited on pages 128, 213, 490 and 491.

[Kleppe, Warmer and Bast, 2003] Anneke Kleppe, Jos Warmer and Wim Bast (2003): MDA Explained: The Model Driven Architecture: Practice and Promise. Addison Wesley, Bosten. Cited on pages 59, 67, 68, 193 and 216.

[Knapp and Mossakowski, 2018] Alexander Knapp and Till Mossakowski (2018): *Multi-view Consistency in UML: A Survey.* In: LNCS, vol. LNCS 10800, pp. 37–60. Cited on pages 105, 109, 136, 166 and 206.

[Kolovos, Di Ruscio et al., 2009] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio and Richard F. Paige (2009): *Different models for model matching: An analysis of approaches to support model differencing.* In: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, CVSM 2009, pp. 1–6. Cited on page 228.

[Kolovos, Paige and Polack, 2006] Dimitrios S. Kolovos, Richard F. Paige and Fiona A. C. Polack (2006): *Merging Models with the Epsilon Merging Language (EML).* In: LNCS, vol. LNCS 4199, pp. 215–229. Cited on page 186.

[Kolovos, Paige and Polack, 2009] Dimitrios S. Kolovos, Richard F. Paige and Fiona A. C. Polack (2009): *On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages.* In: LNCS, vol. LNCS 5115, pp. 204–218. Cited on pages 110, 111 and 269.

[Kolovos, Tisi et al., 2013] Dimitrios S. Kolovos, Massimo Tisi, Jordi Cabot, Louis M. Rose, Nicholas Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan de Lara, István Ráth and Dániel Varró (2013): *A research roadmap towards achieving scalability in model driven engineering.* In: Proceedings of the Workshop on Scalability in Model Driven Engineering - BigMDE '13, pp. 1–10. Cited on page 492.

[König and Diskin, 2017] Harald König and Zinovy Diskin (2017): *Efficient consistency checking of interrelated models.* In: LNCS, vol. LNCS 10376, pp. 161–178. Cited on pages 108 and 461.

[Kosar, Bohra and Mernik, 2016] Tomaž Kosar, Sudev Bohra and Marjan Mernik (2016): *Domain-Specific Languages: A Systematic Mapping Study.* In: Information and Software Technology, vol. 71(March), pp. 77–91. Cited on page 137.

[Kramer and Langhammer, 2014] M E Kramer and M Langhammer (2014): *Proposal for a multi-view modelling case study: Component-based software engineering with UML, plug-ins, and Java*. In: 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, VAO 2014, pp. 7–10. Cited on page 129.

[Kramer, 2015] Max E. Kramer (2015): *A Generative Approach to Change-Driven Consistency in Multi-View Modeling*. In: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures - QoSA '15, pp. 129–134. Cited on page 127.

[Kramer, Burger and Langhammer, 2013] Max E Kramer, Erik Burger and Michael Langhammer (2013): *View-centric engineering with synchronized heterogeneous models*. In: Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '13, pp. 1–6. Cited on page 126.

[Kramer, Langhammer et al., 2015] Max E. Kramer, Michael Langhammer, Dominik Messinger, Stephan Seifermann and Erik Burger (2015): *Change-Driven Consistency for Component Code, Architectural Models, and Contracts*. In: Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering - CBSE '15, (section 5), pp. 21–26. Cited on page 129.

[Kramer and Rakhman, 2016] Max E. Kramer and Kirill Rakhman (2016): *Automated inversion of attribute mappings in bidirectional model transformations*. In: CEUR Workshop Proceedings, vol. 1571(Bx), pp. 61–76. Cited on page 129.

[Kramer, 2017] Max Emanuel Kramer (2017): Specification Languages for Preserving Consistency between Models of Different Languages. KIT Scientific Publishing, Karlsruhe. Cited on pages 76, 85, 129, 170, 202, 214, 222, 229, 490, 493 and 495.

[Kramler, Kappel et al., 2006] G. Kramler, G. Kappel, T. Reiter, E. Kapsammer, W. Retschitzegger and W. Schwinger (2006): *Towards a semantic infrastructure supporting model-based tool integration*. In: Proceedings of the 2006 international workshop on Global integrated model management - GaMMa '06, p. 43. Cited on page 143.

[Kress, 2021] Dominik Kress (2021): GraphQL : eine Einführung in APIs mit GraphQL. dpunkt.verlag, Heidelberg. Cited on page 200.

[Kruse, 2011] Steffen Kruse (2011): *On the Use of Operators for the Co-Evolution of Metamodels and Transformations*. In: Preliminary Proceedings of the International Workshop on Models and Evolution, (June). Cited on page 187.

[Kuhn, Murphy and Thompson, 2012] Adrian Kuhn, Gail C. Murphy and C. Albert Thompson (2012): *An exploratory study of forces and frictions affecting large-scale model-driven development*. In: LNCS, vol. LNCS 7590, pp. 352–367. Cited on pages 33 and 486.

[Kühn, Bayer and Karagiannis, 2003] Harald Kühn, Franz Bayer and Dimitris Karagiannis (2003): *Enterprise Model Integration*. In: Proceedings of the 4th International Conference, pp. 379–392. Cited on pages 144 and 187.

[Kühn, Böhme et al., 2015] Thomas Kühn, Stephan Böhme, Sebastian Götz and Uwe Aßmann (2015): *A Combined Formal Model for Relational Context-Dependent Roles*. In: SLE 2015 - Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 113–124. Cited on pages 85 and 130.

[Kühn, Leuthäuser et al., 2014] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl and Uwe Aßmann (2014): *A Metamodel Family for Role-Based Modeling and Programming Languages*. pp. 141–160. Cited on page 129.

[Kühne, 2006] Thomas Kühne (2006): *Matters of (meta-) modeling*. In: Software and Systems Modeling, vol. 5(4), pp. 369–385. Cited on pages 60, 61, 62 and 64.

[Kühne, 2018] Thomas Kühne (2018): *Exploring Potency*. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 2–12. Cited on page 63.

[Kulkarni, Reddy and Rajbhoj, 2010] Vinay Kulkarni, Sreedhar Reddy and Asha Rajbhoj (2010): *Scaling Up Model Driven Engineering – Experience and Lessons Learnt*. In: Model Driven Engineering Languages and Systems, pp. 331–345. Cited on page 492.

[Kullbach, Winter et al., 1998] Bernt Kullbach, Andreas Winter, Peter Dahm and Jürgen Ebert (1998): *Program comprehension in multi-language systems*. In: Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261), pp. 135–143. Cited on page 498.

[Kurpjuweit and Winter, 2007] S Kurpjuweit and R Winter (2007): *Viewpoint-based meta model engineering*. In: Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures - Concepts and Applications, EMISA 2007, pp. 143–162. Cited on pages 165 and 497.

[Kurtev, Bézivin and Akcsit, 2002] I Kurtev, J Bézivin and M Akcsit (2002): *Technological Spaces: An Initial Appraisal*. In: International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track, Irvine, USA, pp. 1–6. Cited on page 84.

[Kurtev, 2008] Ivan Kurtev (2008): *State of the art of QVT: A model transformation language standard*. In: LNCS, vol. LNCS 5088(ii), pp. 377–393. Cited on pages 104, 111, 121 and 165.

[Kuryazov, 2019] Dilshodbek Kuryazov (2019): Model Difference Representation. Phd thesis, University of Oldenburg. Cited on pages 225, 228, 234 and 479.

[Kuryazov and Winter, 2014] Dilshodbek Kuryazov and Andreas Winter (2014): *Representing Model Differences by Delta Operations*. In: 18th International Enterprise Distributed Object Oriented Computing Conference, Workshops and Demonstrations (EDOCW), pp. 211–220. Cited on page 228.

[Kusel, Schönböck et al., 2015] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger and W. Schwinger (2015): *Reuse in model-to-model transformation languages: are we there yet?* In: Software and Systems Modeling, vol. 14(2), pp. 537–572. Cited on pages 203 and 204.

[Kusel, Etzlstorfer et al., 2013] Angelika Kusel, Juergen Etzlstorfer, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger and Manuel Wimmer (2013): *A Survey on Incremental Model Transformation Approaches*. In: ME 2013–Models and Evolution Workshop Proceedings, vol. 1090, pp. 4–13. Cited on pages 70 and 149.

[Küster, Völzer et al., 2016] Jochen Küster, Hagen Völzer, Cédric Favre, Moisés Castelo Branco and Krzysztof Czarnecki (2016): *Supporting different process views through a Shared Process Model.* In: Software & Systems Modeling, vol. 15(4), pp. 1207–1233. Cited on page 146.

[Lämmel, 2016] Ralf Lämmel (2016): *Coupled software transformations revisited.* In: Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, pp. 239–252. Cited on page 105.

[Lange, Atkinson and Tunjic, 2020] Arne Lange, Colin Atkinson and Christian Tunjic (2020): *Simplified View Generation in a Deep View-Based Modeling Environment.* In: Communications in Computer and Information Science, vol. 1262 CCIS, Springer, pp. 163–179. Cited on page 125.

[Langer, Wieland et al., 2011] Philip Langer, Konrad Wieland, Manuel Wimmer and Jordi Cabot (2011): *From UML Profiles to EMF Profiles and Beyond.* In: LNCS, vol. LNCS 6705, pp. 52–67. Cited on page 133.

[Langhammer, 2017] Michael Langhammer (2017): Automated Coevolution of Source Code and Software Architecture Models. KIT Scientific Publishing, Karlsruhe. Cited on page 495.

[Lano and Kolahdouz-Rahimi, 2021] K. Lano and S. Kolahdouz-Rahimi (2021): *Implementing QVT-R via semantic interpretation in UML-RSDS.* In: Software and Systems Modeling, vol. 20(3), pp. 725–766. Cited on page 111.

[Lano, Kolahdouz-Rahimi et al., 2014] K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, J. Terrell and S. Zschaler (2014): *Correct-by-construction synthesis of model transformations using transformation patterns.* In: Software & Systems Modeling, vol. 13(2), pp. 873–907. Cited on page 186.

[Lano, Kolahdouz-Rahimi et al., 2018] Kevin Lano, Shekoufeh Kolahdouz-Rahimi, Sobhan Yassipour-Tehrani and Mohammadreza Sharbaf (2018): *A survey of model transformation design patterns in practice.* In: Journal of Systems and Software, vol. 140, pp. 48–73. Cited on page 186.

[de Lara and Guerra, 2012] Juan de Lara and Esther Guerra (2012): *Inter-Modelling with Graphical Constraints: Foundations and Applications.* In: 11th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2012), vol. 47, pp. 1–16. Cited on pages 113 and 118.

[de Lara and Guerra, 2017] Juan de Lara and Esther Guerra (2017): *A Posteriori Typing for Model-Driven Engineering: Concepts, Analysis, and Applications.* In: ACM Transactions on Software Engineering and Methodology, vol. 25(4), pp. 1–60. Cited on page 230.

[de Lara, Guerra and Vangheluwe, 2006] Juan de Lara, Esther Guerra and Hans Vangheluwe (2006): *A Multi-View Component Modelling Language for Systems Design: Checking Consistency and Timing Constrains.* In: 1st International TGG workshop. Cited on pages 97, 121, 122 and 134.

[Lauder, Anjorin et al., 2012] Marius Lauder, Anthony Anjorin, Gergely Varró and Andy Schürr (2012): *Efficient model synchronization with precedence triple graph grammars.* In: LNCS, vol. LNCS 7562, pp. 401–415. Cited on page 114.

[Le Noir, Delande et al., 2011] Jerome Le Noir, Olivier Delande, Daniel Exertier, Marcos Aurélio Almeida Da Silva and Xavier Blanc (2011): *Operation based model representation: Experiences on inconsistency detection.* In: LNCS, vol. LNCS 6698, pp. 85–96. Cited on pages 228 and 267.

[Leblebici, Anjorin et al., 2017] Erhan Leblebici, Anthony Anjorin, Lars Fritsche, Gergely Varró and Andy Schürr (2017): *Leveraging Incremental Pattern Matching Techniques for Model Synchronisation.* In: LNCS, vol. LNCS 10373, pp. 179–195. Cited on page 114.

[Leblebici, Anjorin et al., 2014] Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke and Joel Greenyer (2014): *A comparison of incremental Triple Graph Grammar tools.* In: Electronic Communications of the EASST, vol. 67. Cited on page 114.

[Leblebici, Anjorin et al., 2015] Erhan Leblebici, Anthony Anjorin, Andy Schürr and Gabriele Taentzer (2015): *Multi-amalgamated Triple Graph Grammars.* In: LNCS, vol. 9151, pp. 87–103. Cited on page 111.

[Leduc, Degueule et al., 2020] Manuel Leduc, Thomas Degueule, Eric Van Wyk and Benoit Combemale (2020): *The Software Language Extension Problem.* In: Software and Systems Modeling, vol. 19(2), pp. 263–267. Cited on page 133.

[Lee, 2010] Edward a. Lee (2010): *Disciplined heterogeneous modeling: Invited paper.* In: LNCS, vol. LNCS 6395(PART 2), pp. 273–287. Cited on pages 27 and 117.

[Leonhardt, Hettwer et al., 2015] Sven Leonhardt, Benjamin Hettwer, Johannes Hoor and Michael Langhammer (2015): *Integration of Existing Software Artifacts into a View- and Change-Driven Development Approach.* In: Proceedings of the 2015 Joint MORSE/-VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering, pp. 17–24. Cited on page 127.

[Leser and Naumann, 2007] Ulf Leser and Felix Naumann (2007): Informationsintegration - Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen. dpunkt.verlag, Heidelberg. Cited on pages 95, 96, 140, 142, 143 and 472.

[Lettner, Tschernuth and Mayrhofer, 2011] Michael Lettner, Michael Tschernuth and Rene Mayrhofer (2011): *A critical review of applied MDA for embedded devices: Identification of problem classes and discussing porting efforts in practice.* In: LNCS, vol. LNCS 6981, pp. 228–242. Cited on pages 30 and 487.

[Leuthäuser and Aßmann, 2015] Max Leuthäuser and Uwe Aßmann (2015): *Enabling view-based programming with SCROLL: Using roles and dynamic dispatch for etablishing view-based programming.* In: ACM International Conference Proceeding Series, vol. 21-July-20, pp. 25–33. Cited on page 131.

[Levendovszky, Balasubramanian et al., 2010] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan and Gabor Karsai (2010): *A novel approach to semi-automated evolution of DSML model transformation.* In: LNCS, vol. LNCS 5969, pp. 23–41. Cited on page 187.

[Levendovszky, Balasubramanian et al., 2013] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, Feng Shi, Chris van Buskirk and Gabor Karsai (2013): *A semi-formal description of migrating domain-specific models with evolving domains.* In: Software & Systems Modeling, vol. 13(2), pp. 807–823. Cited on page 194.

[Li, Li and Stolz, 2011] Dan Li, Xiaoshan Li and Volker Stolz (2011): *QVT-based model transformation using XSLT*. In: ACM SIGSOFT Software Engineering Notes, vol. 36(1), p. 1. Cited on page 111.

[Liebel, Marko et al., 2018] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner and Jörgen Hansson (2018): *Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice*. In: Software & Systems Modeling, vol. 17(1), pp. 91–113. Cited on page 499.

[Lopes, Hammoudi et al., 2006] Denivaldo Lopes, Slimane Hammoudi, Jose Souza and Alan Bontempo (2006): *Metamodel Matching: Experiments and Comparison*. In: 2006 International Conference on Software Engineering Advances (ICSEA'06), vol. 00(c), pp. 2–2. Cited on page 480.

[López-Fernández, Cuadrado et al., 2015] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra and Juan de Lara (2015): *Example-driven meta-model development*. In: Software & Systems Modeling, vol. 14(4), pp. 1323–1347. Cited on page 494.

[López-Fernández, Garmendia et al., 2019] Jesús J. López-Fernández, Antonio Garmendia, Esther Guerra and Juan de Lara (2019): *An example is worth a thousand words: Creating graphical modelling environments by example*. In: Software & Systems Modeling, vol. 18(2), pp. 961–993. Cited on page 494.

[Lu and Holubová, 2019] Jiaheng Lu and Irena Holubová (2019): *Multi-model Databases: A New Journey to Handle the Variety of Data*. In: ACM Computing Surveys, vol. 52(3), pp. 1–38. Cited on page 139.

[Lucas, Molina and Toval, 2009] Francisco J. Lucas, Fernando Molina and Ambrosio Toval (2009): *A systematic review of UML model consistency management*. In: Information and Software Technology, vol. 51(12), pp. 1631–1645. Cited on pages 35, 43, 136, 267, 479 and 499.

[Lúcio, Amrani et al., 2016] Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan M. K. Selim, Eugene Syriani and Manuel Wimmer (2016): *Model transformation intents and their properties*. In: Software & Systems Modeling, vol. 15(3), pp. 647–684. Cited on pages 104 and 112.

[Lúcio, Mustafiz et al., 2013] Levi Lúcio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe and Maris Jukss (2013): *FTG+PM: An Integrated Framework for Investigating Model Transformation Chains*. In: LNCS, vol. LNCS 7916, pp. 182–202. Cited on page 204.

[Ludewig, 2004] Jochen Ludewig (2004): *Models in software engineering - an introduction*. In: Informatik - Forschung und Entwicklung, vol. 18(3-4), pp. 105–112. Cited on page 27.

[Macedo and Cunha, 2013] Nuno Macedo and Alcino Cunha (2013): *Implementing QVT-R Bidirectional Model Transformations Using Alloy*. In: LNCS, vol. LNCS 7793, pp. 297–311. Cited on pages 109, 111 and 149.

[Macedo and Cunha, 2016] Nuno Macedo and Alcino Cunha (2016): *Least-change bidirectional model transformation with QVT-R and ATL*. In: Software & Systems Modeling, vol. 15(3), pp. 783–810. Cited on page 114.

[Macedo, Cunha and Pacheco, 2014] Nuno Macedo, Alcino Cunha and Hugo Pacheco (2014): *Towards a Framework for Multidirectional Model Transformations.* In: Workshop Pro- ceedings of the EDBT/ICDT 2014 Joint Conference, pp. 71–74. Cited on pages 111, 147 and 486.

[Macedo, Jorge and Cunha, 2017] Nuno Macedo, Tiago Jorge and Alcino Cunha (2017): *A Feature-Based Classification of Model Repair Approaches.* In: IEEE Transactions on Software Engineering, vol. 43(7), pp. 615–640. Cited on pages 106, 107, 108, 202 and 206.

[Madani, Kolovos and Paige, 2021] Sina Madani, Dimitris Kolovos and Richard F. Paige (2021): *Distributed model validation with Epsilon.* In: Software and Systems Modeling. Cited on page 110.

[Makedonski and Grabowski, 2020] Philip Makedonski and Jens Grabowski (2020): *Facilitating the Co-Evolution of Semantic Descriptions in Standards and Models.* In: Proceedings of the 12th System Analysis and Modelling Conference, pp. 75–84. Cited on page 122.

[Mallet, Lagarde et al., 2010] Frédéric Mallet, François Lagarde, Charles André, Sébastien Gérard and François Terrier (2010): *An Automated Process for Implementing Multilevel Domain Models.* In: LNCS, vol. LNCS 5969, pp. 314–333. Cited on page 86.

[Mannadiar and Vangheluwe, 2011] Raphael Mannadiar and Hans Vangheluwe (2011): *Debugging in domain-specific modelling.* In: LNCS, vol. LNCS 6563, pp. 276–285. Cited on page 493.

[Maoz and Ringert, 2018] Shahar Maoz and Jan Oliver Ringert (2018): *A framework for relating syntactic and semantic model differences.* In: Software & Systems Modeling, vol. 17(3), pp. 753–777. Cited on page 229.

[Maraee and Sturm, 2021] Azzam Maraee and Arnon Sturm (2021): *Imperative versus declarative constraint specification languages: a controlled experiment.* In: Software and Systems Modeling, vol. 20(1), pp. 27–48. Cited on page 268.

[Margaria and Steffen, 2009] Tiziana Margaria and Bernhard Steffen (2009): *The One-Thing-Approach.* In: Handbook of Research on Business Process Modeling, vol. 49, IGI Global, pp. 1–26. Cited on page 121.

[Maro, Steghöfer et al., 2015] Salome Maro, Jan-Philipp Steghöfer, Anthony Anjorin, Matthias Tichy and Lars Gelin (2015): *On integrating graphical and textual editors for a UML profile based domain specific language: an industrial experience.* In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 1–12. Cited on pages 32, 56 and 132.

[Maschotta, Wichmann et al., 2019] Ralph Maschotta, Alexander Wichmann, Armin Zimmermann and Kristina Gruber (2019): *Integrated Automotive Requirements Engineering with a SysML-Based Domain-Specific Language.* In: 2019 IEEE International Conference on Mechatronics (ICM), pp. 402–409. Cited on page 86.

[Mazkatli, Burger et al., 2017] Manar Mazkatli, Erik Burger, Anne Koziolek and Ralf H Reussner (2017): *Automotive Systems Modelling with Vitruvius.* In: INFORMATIK 2017, pp. 1487–1498. Cited on page 129.

[Mazkatli, Burger et al., 2018] Manar Mazkatli, Erik Burger, Jochen Quante and Anne Koziolek (2018): *Integrating semantically-related legacy models in vitruvius*. In: Proceedings of the 10th International Workshop on Modelling in Software Engineering - MiSE '18, pp. 41–48. Cited on page 128.

[Mazkatli, Monschein et al., 2020] Manar Mazkatli, David Monschein, Johannes Grohmann and Anne Koziolek (2020): *Incremental Calibration of Architectural Performance Models with Parametric Dependencies*. In: 2020 IEEE International Conference on Software Architecture (ICSA), pp. 23–34. Cited on page 129.

[McBrien and Poulovassilis, 1999] Peter McBrien and Alexandra Poulovassilis (1999): *A Uniform Approach to Inter-model Transformations*. In: Matthias Jarke and Andreas Oberweis (Eds.): Advanced Information Systems Engineering. CAiSE 1999, Springer, Berlin, Heidelberg, vol 1626 edn., pp. 333–348. Cited on page 85.

[Mehner, Monga and Taentzer, 2009] Katharina Mehner, Mattia Monga and Gabriele Taentzer (2009): *Analysis of Aspect-Oriented Model Weaving*. In: LNCS, vol. LNCS 5490, pp. 235–263. Cited on page 118.

[Meier, Kateule and Winter, 2020] Johannes Meier, Ruthbetha Kateule and Andreas Winter (2020): *Operator-based viewpoint definition*. In: MODELSWARD 2020 - Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, pp. 401–408. Cited on pages 19, 185, 216 and 383.

[Meier, Klare et al., 2019] Johannes Meier, Heiko Klare, Christian Tunjic, Colin Atkinson, Erik Burger, Ralf Reussner and Andreas Winter (2019): *Single Underlying Models for Projectional, Multi-View Environments*. In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, pp. 119–130. Cited on pages 26, 82, 123, 164, 167, 169 and 483.

[Meier, Kuryazov et al., 2015] Johannes Meier, Dilshodbek Kuryazov, Jan Jelschen and Andreas Winter (2015): *A Quality Control Center for Software Migration*. In: Softwaretechnik-Trends. Cited on page 19.

[Meier, Werner et al., 2020] Johannes Meier, Christopher Werner, Heiko Klare, Christian Tunjic, Uwe Aßmann, Colin Atkinson, Erik Burger, Ralf Reussner and Andreas Winter (2020): *Classifying Approaches for Constructing Single Underlying Models*. In: Slimane Hammoudi, Luís Pires Ferreira and Bran Selic (Eds.): Model-Driven Engineering and Software Development. MODELSWARD 2019. Communications in Computer and Information Science (CCIS), Springer, Cham, pp. 350–375. Cited on pages 26, 80, 81, 82, 121, 123, 124, 125, 126, 127, 130, 131, 172, 473, 477, 484, 485, 495, 496, 498, 507 and 510.

[Meier and Winter, 2016] Johannes Meier and Andreas Winter (2016): *Towards Metamodel Integration Using Reference Metamodels*. In: Proceedings of the 4th Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO 2016), pp. 19–22. Cited on pages 21 and 497.

[Meier and Winter, 2018a] Johannes Meier and Andreas Winter (2018a): *Model Consistency ensured by Metamodel Integration*. In: 6th International Workshop on The Globalization of Modeling Languages, co-located with MODELS 2018. Cited on pages 26 and 163.

[Meier and Winter, 2018b] Johannes Meier and Andreas Winter (2018b): *Towards Evolution Scenarios of Integrated Software Artifacts*. In: Softwaretechnik-Trends, vol. 38(2), pp. 63–64. Cited on page 477.

[Meier and Winter, 2018c] Johannes Meier and Andreas Winter (2018c): *Traceability enabled by Metamodel Integration.* In: Softwaretechnik-Trends, vol. 38(1), pp. 21–26. Cited on pages 460 and 487.

[Melnik and Bernstein, 2004] S Melnik and PA Bernstein (2004): A semantics for model management operators. Tech. rep. Cited on page 140.

[Méndez Fernández, Böhm et al., 2019] Daniel Méndez Fernández, Wolfgang Böhm, Andreas Vogelsang, Jakob Mund, Manfred Broy, Marco Kuhrmann and Thorsten Weyer (2019): *Artefacts in software engineering: a fundamental positioning.* In: Software & Systems Modeling, vol. 18(5), pp. 2777–2786. Cited on page 36.

[Méndez Fernández, Penzenstadler et al., 2010] Daniel Méndez Fernández, Birgit Penzenstadler, Marco Kuhrmann and Manfred Broy (2010): *A Meta Model for Artefact-Orientation: Fundamentals and Lessons Learned in Requirements Engineering.* In: LNCS, vol. LNCS 6395, pp. 183–197. Cited on page 36.

[Mens and Van Der Straeten, 2007] Tom Mens and Ragnhild Van Der Straeten (2007): *Incremental Resolution of Model Inconsistencies.* In: LNCS, vol. LNCS 4409, pp. 111–126. Cited on pages 117, 150 and 214.

[Mens, Van Der Straeten and D'Hondt, 2006] Tom Mens, Ragnhild Van Der Straeten and Maja D'Hondt (2006): *Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis.* In: Oscar Nierstrasz, Jon Whittle, David Harel and Gianna Reggio (Eds.): Model Driven Engineering Languages and Systems. MODELS 2006. LNCS, Springer, Berlin, Heidelberg, pp. 200–214. Cited on pages 117, 150 and 166.

[Mens and Van Gorp, 2006] Tom Mens and Pieter Van Gorp (2006): *A taxonomy of model transformation.* In: Electronic Notes in Theoretical Computer Science, vol. 152(1-2), pp. 125–142. Cited on pages 68, 69 and 226.

[Mens, Wermelinger et al., 2005] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld and Mehdi Jazayeri (2005): *Challenges in Software Evolution.* In: Eighth International Workshop on Principles of Software Evolution (IWPSE'05), vol. 2005, pp. 13–22. Cited on page 498.

[Meyer, 2016] Aljoscha Meyer (2016): A Framework for Abstract Semantic Graphs - Applied to Java 8. Bachelor thesis, Carl von Ossietzky University of Oldenburg, Germany. Cited on page 96.

[Meyers, Wimmer et al., 2012] Bart Meyers, Manuel Wimmer, Antonio Cicchetti and Jonathan Sprinkle (2012): *A generic in-place transformation-based approach to structured model co-evolution.* In: Electronic Communications of the EASST, vol. 42. Cited on page 241.

[Michail, Kinable et al., 2020] Dimitrios Michail, Joris Kinable, Barak Naveh and John V. Sichi (2020): *JGraphT—A Java Library for Graph Data Structures and Algorithms.* In: ACM Transactions on Mathematical Software, vol. 46(2), pp. 1–29. Cited on page 85.

[Michel, 2019] Stefan Michel (2019): Synchronisierung verteilter Zugangsdaten mit MoConseMI. Bachelor thesis, Carl von Ossietzky University of Oldenburg, Germany. Cited on pages 283, 285 and 496.

[Miller, Ioannidis and Ramakrishnan, 1993] Rj Miller, Ye Ioannidis and R Ramakrishnan (1993): *The use of information capacity in schema integration and translation.* In: Vldb, pp. 120–133. Cited on page 139.

[Miller, Ioannidis and Ramakrishnan, 1994] R.J. Miller, Y.E. Ioannidis and R. Ramakrishnan (1994): *Schema equivalence in heterogeneous systems: Bridging theory and practice.* In: Information Systems, vol. 19(1), pp. 3–31. Cited on page 139.

[Milovanovic and Milicev, 2015] Vukasin Milovanovic and Dragan Milicev (2015): *An interactive tool for UML class model evolution in database applications.* In: Software & Systems Modeling, vol. 14(3), pp. 1273–1295. Cited on page 195.

[Misbhauddin and Alshayeb, 2019] Mohammed Misbhauddin and Mohammad Alshayeb (2019): *An integrated metamodel-based approach to software model refactoring.* In: Software & Systems Modeling, vol. 18(3), pp. 2013–2050. Cited on pages 206 and 479.

[Mohagheghi, Gilani et al., 2013a] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu and Miguel A. Fernandez (2013a): *An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases.* In: Empirical Software Engineering, vol. 18(1), pp. 89–116. Cited on pages 30 and 479.

[Mohagheghi, Gilani et al., 2013b] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, Miguel A. Fernandez, Bjørn Nordmoen and Mathias Fritzsche (2013b): *Where does model-driven engineering help? Experiences from three industrial cases.* In: Software & Systems Modeling, vol. 12(3), pp. 619–639. Cited on page 86.

[Mohania, Konomi and Kambayashi, 1997] Mukesh Mohania, Shin'ichi Konomi and Yahiko Kambayashi (1997): *Incremental maintenance of materialized views.* pp. 551–560. Cited on page 140.

[Möller, Scherzinger et al., 2020] Mark Lukas Möller, Stefanie Scherzinger, Meike Klettke and Uta Störl (2020): *Why It Is Time for Yet Another Schema Evolution Benchmark.* In: Lecture Notes in Business Information Processing, vol. 386 LNBIP, pp. 113–125. Cited on page 195.

[Monschein, Mazkatli et al., 2021] David Monschein, Manar Mazkatli, Robert Heinrich and Anne Koziolek (2021): *Enabling Consistency between Software Artefacts for Software Adaption and Evolution.* In: 2021 IEEE 18th International Conference on Software Architecture (ICSA), pp. 1–12. Cited on page 129.

[Moreno and Vallecillo, 2004] Nathalie Moreno and Antonio Vallecillo (2004): *What do we do with re-use in MDA?* In: Computer Science at Kent, pp. 136–145. Cited on pages 37 and 165.

[Mossakowski, Codescu et al., 2015] Till Mossakowski, Mihai Codescu, Fabian Neuhaus and Oliver Kutz (2015): *The Distributed Ontology, Modeling and Specification Language – DOL.* No. 2 in Studies in Universal Logic, Springer, Cham, pp. 489–520. Cited on page 142.

[Mosser and Blay-Fornarino, 2013] Sébastien Mosser and Mireille Blay-Fornarino (2013): *" Adore ", a logical meta-model supporting business process evolution.* In: Science of Computer Programming, vol. 78(8), pp. 1035–1054. Cited on page 189.

[Mosterman and Vangheluwe, 2004] Pieter J. Mosterman and Hans Vangheluwe (2004): *Computer Automated Multi-Paradigm Modeling: An Introduction.* In: SIMULATION, vol. 80(9), pp. 433–450. Cited on page 45.

[Mukherjee, Kovacevic et al., 2008] Patrick Mukherjee, Aleksandra Kovacevic, Michael Benz and Andy Schürr (2008): *Towards a Peer-to-Peer Based Global Software Development Environment.* In: Proceedings of the Software Engineering Conference SE2008, pp. 204–216. Cited on page 108.

[Mussbacher, Amyot et al., 2014] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-michel Bruel, Betty H C Cheng, Philippe Collet, Benoit Combemale, Robert B France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum and Jon Whittle (2014): *The Relevance of Model-Driven Engineering Thirty Years from Now.* In: Model-Driven Engineering Languages and Systems, pp. 183–200. Cited on pages 29, 30 and 497.

[Mussbacher, Combemale et al., 2020] Gunter Mussbacher, Benoit Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien Mosser, Houari Sahraoui, Eugene Syriani, Dániel Varró and Martin Weyssow (2020): *Opportunities in intelligent modeling assistance.* In: Software and Systems Modeling, vol. 19(5), pp. 1045–1053. Cited on page 106.

[Narayanan, Levendovszky et al., 2009] Anantha Narayanan, Tihamer Levendovszky, Daniel Balasubramanian and Gabor Karsai (2009): *Automatic Domain Model Migration to Manage Metamodel Evolution.* In: LNCS, vol. LNCS 5795, pp. 706–711. Cited on page 194.

[Nassar, Radke and Arendt, 2017] Nebras Nassar, Hendrik Radke and Thorsten Arendt (2017): *Rule-Based Repair of EMF Models: An Automated Interactive Approach.* In: LNCS, vol. LNCS 10374, pp. 171–181. Cited on page 46.

[Nentwich, Emmerich et al., 2003] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelsteiin and Ernst Ellmer (2003): *Flexible consistency checking.* In: ACM Transactions on Software Engineering and Methodology, vol. 12(1), pp. 28–63. Cited on page 109.

[Nentwich, Emmerich and Finkelstein, 2003] Christian Nentwich, Wolfgang Emmerich and Anthony Finkelstein (2003): *Consistency management with repair actions.* In: 25th International Conference on Software Engineering, 2003. Proceedings., vol. 6, pp. 455–464. Cited on pages 82 and 109.

[Nešić, Krüger et al., 2019] Damir Nešić, Jacob Krüger, Ștefan Stănciulescu and Thorsten Berger (2019): *Principles of feature modeling.* In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 62–73. Cited on page 22.

[Neubauer, Bill et al., 2017] Patrick Neubauer, Robert Bill, Tanja Mayerhofer and Manuel Wimmer (2017): *Automated generation of consistency-achieving model editors.* In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 127–137. Cited on page 138.

[Noy, 2004] Natalya F. Noy (2004): *Semantic Integration: A Survey Of Ontology-Based Approaches.* In: ACM SIGMOD Record, vol. 33(4), pp. 65–70. Cited on page 143.

[Noy and Klein, 2004] Natalya F. Noy and Michel Klein (2004): *Ontology Evolution: Not the Same as Schema Evolution.* In: Knowledge and Information Systems, vol. 6(4), pp. 428–440. Cited on page 195.

[Noyrit, Gérard et al., 2010] Florian Noyrit, Sébastien Gérard, François Terrier and Bran Selic (2010): *Consistent modeling using multiple UML profiles.* In: LNCS, vol. LNCS 6394(PART 1), pp. 392–406. Cited on page 133.

[Object Management Group, 2013] Object Management Group (2013): MOF Support for Semantic Structures (Version 1.0). Tech. rep. Cited on page 231.

[Object Management Group, 2015] Object Management Group (2015): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (Version 1.2). Tech. rep. Cited on pages 110 and 111.

[Object Management Group, 2017] Object Management Group (2017): OMG Unified Modeling Language (OMG UML), Version 2.5.1. Tech. rep. Cited on pages 27, 61, 85 and 136.

[Object Management Group, 2019] Object Management Group (2019): OMG Meta Object Facility (MOF) Core Specification, Version 2.5.1. Tech. rep. Cited on pages 60, 61 and 85.

[Oueslati and Akaichi, 2010] Wided Oueslati and Jalel Akaichi (2010): *A Survey on Data Warehouse Evolution.* In: International Journal of Database Management Systems, vol. 2(4), pp. 11–24. Cited on page 473.

[Özsu and Valduriez, 2020] M. Tamer Özsu and Patrick Valduriez (2020): Principles of Distributed Database Systems. Springer, Cham. Cited on page 139.

[Paige, Brooke and Ostroff, 2007] Richard F. Paige, Phillip J. Brooke and Jonathan S. Ostroff (2007): *Metamodel-based model conformance and multiview consistency checking.* In: ACM Transactions on Software Engineering and Methodology, vol. 16(3), p. 11. Cited on pages 32 and 136.

[Paige, Kolovos et al., 2009] Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos and Fiona A.C. Polack (2009): *The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering.* In: 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, pp. 162–171. Cited on pages 110 and 194.

[Paige, Matragkas and Rose, 2016] Richard F. Paige, Nicholas Matragkas and Louis M. Rose (2016): *Evolving models in Model-Driven Engineering: State-of-the-art and future challenges.* In: Journal of Systems and Software, vol. 111, pp. 272–280. Cited on page 188.

[Paige and Ostroff, 2002] Richard F. Paige and Jonathan S. Ostroff (2002): *The Single Model Principle.* In: The Journal of Object Technology, vol. 1(5), p. 63. Cited on page 121.

[Pardillo, 2010] Jesús Pardillo (2010): *A Systematic Review on the Definition of UML Profiles.* In: Dorina C. Petriu, Nicolas Rouquette and Øystein Haugen (Eds.): LNCS, vol. 6394, Springer, pp. 407–422. Cited on pages 85 and 133.

[Parreiras, Staab et al., 2008] Fernando Silva Parreiras, Steffen Staab, Simon Schenk and Andreas Winter (2008): *Model Driven Specification of Ontology Translations.* In: Conceptual Modeling - Er 2008, Proceedings, vol. 5231, pp. 484–497. Cited on page 143.

[Parreiras, Staab and Winter, 2007] Fernando Silva Parreiras, Steffen Staab and Andreas Winter (2007): *On marrying ontological and metamodeling technical spaces.* In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 439–488. Cited on page 142.

[Pastor and Molina, 2007] Oscar Pastor and Juan Carlos Molina (2007): Model-Driven Architecture in Practice. Springer, Berlin, Heidelberg. Cited on page 59.

[Persson, Torngren et al., 2013] Magnus Persson, Martin Torngren, Ahsan Qamar, Jonas Westman, Matthias Biehl, Stavros Tripakis, Hans Vangheluwe and Joachim Denil (2013): *A characterization of integrated multi-view modeling in the context of embedded and cyber-physical systems.* In: 2013 Proceedings of the International Conference on Embedded Software (EMSOFT), pp. 1–10. Cited on pages 29, 30, 32, 33, 40, 42, 44, 121, 187, 214, 459, 473 and 499.

[Pfeiffer and Wasowski, 2012] Rolf-Helge Pfeiffer and Andrzej Wasowski (2012): *Cross-Language Support Mechanisms Significantly Aid Software Development.* In: LNCS, vol. LNCS 7590, pp. 168–184. Cited on page 167.

[Pfeiffer and Wasowski, 2013] Rolf Helge Pfeiffer and Andrzej Wasowski (2013): *The design space of multi-language development environments.* In: Software and Systems Modeling, pp. 1–29. Cited on pages 36 and 459.

[Pinna Puissant, Van Der Straeten and Mens, 2015] Jorge Pinna Puissant, Ragnhild Van Der Straeten and Tom Mens (2015): *Resolving model inconsistencies using automated regression planning.* In: Software & Systems Modeling, vol. 14(1), pp. 461–481. Cited on page 109.

[Pohlmann, Meyer et al., 2014] Uwe Pohlmann, Matthias Meyer, Andreas Dann and Christopher Brink (2014): *Viewpoints and views in hardware platform modeling for safe deployment.* In: ACM International Conference Proceeding Series, pp. 23–30. Cited on page 28.

[Pol'la, Buccella and Cechich, 2021] Matias Pol'la, Agustina Buccella and Alejandra Cechich (2021): *Analysis of variability models: a systematic literature review.* In: Software and Systems Modeling, vol. 20(4), pp. 1043–1077. Cited on pages 108 and 135.

[Qiao, Li et al., 2015] Lin Qiao, Yinan Li, Sahil Takiar, Ziyang Liu, Narasimha Veeramreddy, Min Tu, Ying Dai, Issac Buenrostro, Kapil Surlaker, Shirshanka Das and Chavdar Botev (2015): *Gobblin: Unifying Data Ingestion for Hadoop.* In: Proceedings of the VLDB Endowment, vol. 8(12), pp. 1764–1769. Cited on page 140.

[Rahm and Bernstein, 2001] Erhard Rahm and Philip a. Bernstein (2001): *A survey of approaches to automatic schema matching.* In: VLDB Journal, vol. 10(4), pp. 334–350. Cited on page 139.

[Rahm and Bernstein, 2006] Erhard Rahm and Philip a. Bernstein (2006): *An online bibliography on schema evolution.* In: ACM SIGMOD Record, vol. 35(4), pp. 30–31. Cited on page 195.

[Rahmani, Oberle and Dahms, 2010] Tirdad Rahmani, Daniel Oberle and Marco Dahms (2010): *An Adjustable Transformation from OWL to Ecore.* pp. 243–257. Cited on page 142.

[Rani, Goyal and Gadia, 2015] Asma Rani, Navneet Goyal and Shashi K. Gadia (2015): *Data Provenance for Historical Queries in Relational Database.* In: Proceedings of the 8th Annual ACM India Conference on - Compute '15, vol. 29-31-Octo, pp. 117–122. Cited on page 141.

[Ráth, Hegedüs and Varró, 2012] István Ráth, Ábel Hegedüs and Dániel Varró (2012): *Derived features for EMF by integrating advanced model queries.* In: LNCS, vol. LNCS 7349, pp. 102–117. Cited on page 223.

[Reder and Egyed, 2012] Alexander Reder and Alexander Egyed (2012): *Computing repair trees for resolving inconsistencies in design models.* In: 2012 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012 - Proceedings, pp. 220–229. Cited on pages 40, 73, 109 and 267.

[Reimann, Seifert and Aßmann, 2010] Jan Reimann, Mirko Seifert and Uwe Aßmann (2010): *Role-Based Generic Model Refactoring.* In: LNCS, vol. LNCS 6395, pp. 78–92. Cited on page 188.

[Reiter, Kapsammer et al., 2005] Th. Reiter, E. Kapsammer, W. Retschitzegger and W. Schwinger (2005): *Model Integration Through Mega Operations.* In: Workshop on Model-driven Web Engineering. Cited on page 187.

[Rinker, Waltersdorfer et al., 2021] Felix Rinker, Laura Waltersdorfer, Kristof Meixner, Dietmar Winkler, Arndt Lüder and Stefan Biffl (2021): *Continuous Integration in Multi-view Modeling: A Model Transformation Pipeline Architecture for Production Systems Engineering.* In: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 286–293. Cited on page 122.

[Roddick, 1995] John F. Roddick (1995): *A survey of schema versioning issues for database systems.* In: Information and Software Technology, vol. 37(7), pp. 383–393. Cited on page 195.

[Romero, Jaén and Vallecillo, 2009] J. R. Romero, Juan Ignacio Jaén and Antonio Vallecillo (2009): *Realizing correspondences in multi-viewpoint specifications.* In: Proceedings - 13th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2009, pp. 163–172. Cited on pages 102 and 459.

[Rose, Herrmannsdoerfer et al., 2012] Louis M. Rose, Markus Herrmannsdoerfer, Steffen Mazanek, Pieter Van Gorp, Sebastian Buchwald, Tassilo Horn, Elina Kalnina, Andreas Koch, Kevin Lano, Bernhard Schätz and Manuel Wimmer (2012): *Graph and Model Transformation Tools for Model Migration.* In: Software & Systems Modeling, vol. 13(1), pp. 323–359. Cited on pages 193, 197, 231, 472 and 494.

[Rose, Kolovos et al., 2010] Louis M. Rose, Dimitrios S. Kolovos, Nicholas Drivalos, James R. Williams, Richard F. Paige, Fiona A.C. Polack and Kiran J. Fernandes (2010): *Concordance: A framework for managing model integrity.* In: LNCS, vol. LNCS 6138, pp. 245–260. Cited on page 32.

[Rose, Kolovos et al., 2014] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, Fiona A. C. Polack and Simon Poulding (2014): *Epsilon Flock: a model migration language.* In: Software & Systems Modeling, vol. 13(2), pp. 735–755. Cited on pages 194 and 267.

[Rose, Matragkas et al., 2012] Louis M. Rose, Nicholas Matragkas, Dimitrios S. Kolovos and Richard F. Paige (2012): *A feature model for model-to-text transformation languages.* In: 2012 4th International Workshop on Modeling in Software Engineering (MISE), pp. 57–63. Cited on page 271.

[Roy-Hubara and Sturm, 2020] Noa Roy-Hubara and Arnon Sturm (2020): *Design methods for the new database era: a systematic literature review.* In: Software and Systems Modeling, vol. 19(2), pp. 297–312. Cited on page 139.

[Rubin and Chechik, 2013] Julia Rubin and Marsha Chechik (2013): *N-way model merging*. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013, p. 301. Cited on page 206.

[Ruiz-González, Koch et al., 2009] Daniel Ruiz-González, Nora Koch, Christian Kroiss, José Raúl Romero and Antonio Vallecillo (2009): *Viewpoint synchronization of UWE models*. In: CEUR Workshop Proceedings, vol. 455, pp. 46–60. Cited on page 118.

[Rutle, Iovino et al., 2018] Adrian Rutle, Ludovico Iovino, Harald König and Zinovy Diskin (2018): *Automatic Transformation Co-evolution Using Traceability Models and Graph Transformation*. In: LNCS, vol. LNCS 10890, pp. 80–96. Cited on page 187.

[Rutle, Iovino et al., 2020] Adrian Rutle, Ludovico Iovino, Harald König and Zinovy Diskin (2020): *A query-retyping approach to model transformation co-evolution*. In: Software and Systems Modeling, vol. 19(5), pp. 1107–1138. Cited on page 187.

[Sabetzadeh and Easterbrook, 2003] Mehrdad Sabetzadeh and Steve Easterbrook (2003): *Traceability in Viewpoint Merging: A Model Management Perspective*. In: Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE), pp. 44–49. Cited on page 464.

[Sabetzadeh and Easterbrook, 2006] Mehrdad Sabetzadeh and Steve Easterbrook (2006): *View merging in the presence of incompleteness and inconsistency*. In: Requirements Engineering, vol. 11(3), pp. 174–193. Cited on page 464.

[Säfken, 2020] Pascal Säfken (2020): Transformation von Excel-Tabellen in EMF-Modelle. Bachelor thesis, Carl von Ossietzky University of Oldenburg, Germany. Cited on page 274.

[Saito and Shapiro, 2005] Yasushi Saito and Marc Shapiro (2005): *Optimistic replication*. In: ACM Computing Surveys, vol. 37(1), pp. 42–81. Cited on page 45.

[Salay and Chechik, 2015] Rick Salay and Marsha Chechik (2015): *A Generalized Formal Framework for Partial Modeling*. In: LNCS, vol. 9033, pp. 133–148. Cited on page 108.

[Salay, Gorzny and Chechik, 2013] Rick Salay, Jan Gorzny and Marsha Chechik (2013): *Change propagation due to uncertainty change*. In: LNCS, vol. LNCS 7793, pp. 21–36. Cited on page 107.

[Salay, Kokaly et al., 2020] Rick Salay, Sahar Kokaly, Alessio Di Sandro, Nick L. S. Fung and Marsha Chechik (2020): *Heterogeneous megamodel management using collection operators*. In: Software and Systems Modeling, vol. 19(1), pp. 231–260. Cited on page 186.

[Salay, Mylopoulos and Easterbrook, 2009] Rick Salay, John Mylopoulos and Steve Easterbrook (2009): *Using Macromodels to Manage Collections of Related Models*. In: Advanced Information Systems Engineering, vol. 5565, pp. 141–155. Cited on page 78.

[Samimi-Dehkordi, Zamani and Kolahdouz-Rahimi, 2016] Leila Samimi-Dehkordi, Bahman Zamani and Shekoufeh Kolahdouz-Rahimi (2016): *Bidirectional Model Transformation Approaches: A Comparative Study*. In: 2016 6th International Conference on Computer and Knowledge Engineering (ICCKE), (Iccke), pp. 314–320. Cited on pages 111 and 149.

[Samimi-Dehkordi, Zamani and Kolahdouz-Rahimi, 2018] Leila Samimi-Dehkordi, Bahman Zamani and Shekoufeh Kolahdouz-Rahimi (2018): *EVL+Strace: a novel bidirectional model transformation approach*. In: Information and Software Technology, vol. 100, pp. 47–72. Cited on pages 70, 112, 269 and 463.

[Sánchez-Cuadrado, de Lara and Guerra, 2012] Jesús Sánchez-Cuadrado, Juan de Lara and Esther Guerra (2012): *Bottom-Up Meta-Modelling: An Interactive Approach*. In: Models 2012, pp. 3–19. Cited on page 493.

[Sandmann, 2020] Christian Sandmann (2020): *Graph Repair and its Application to Meta-Modeling*. In: Electronic Proceedings in Theoretical Computer Science, vol. 330(Gcm), pp. 13–34. Cited on page 107.

[Sandmann and Habel, 2019] Christian Sandmann and Annegret Habel (2019): *Rule-based Graph Repair*. In: Electronic Proceedings in Theoretical Computer Science, vol. 309(Gcm), pp. 87–104. Cited on pages 44, 97 and 107.

[Schneider, 2015] Oliver Schneider (2015): Translatability and Translation of Updated Views in ModelJoin. Master thesis, Karlsruhe Institute of Technology, Germany. Cited on page 129.

[Schröpfer, Buchmann and Westfechtel, 2020] Johannes Schröpfer, Thomas Buchmann and Bernhard Westfechtel (2020): *A Generic Projectional Editor for EMF Models*. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 381–392. Cited on page 137.

[Schröpfer, Buchmann and Westfechtel, 2021] Johannes Schröpfer, Thomas Buchmann and Bernhard Westfechtel (2021): *A Framework for Projectional Multi-variant Model Editors*. In: Proceedings of the 9th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 294–305. Cited on page 138.

[Schuler and Kessleman, 2019] Robert E. Schuler and Carl Kessleman (2019): *A High-level User-oriented Framework for Database Evolution*. In: Proceedings of the 31st International Conference on Scientific and Statistical Database Management, (1), pp. 157–168. Cited on page 195.

[Schürr and Klar, 2008] Andy Schürr and Felix Klar (2008): *15 Years of triple graph grammars: Research challenges, new contributions, open problems*. In: LNCS, vol. LNCS 5214, pp. 411–425. Cited on pages 70 and 111.

[Schwarz, Ebert and Winter, 2010] Hannes Schwarz, Jürgen Ebert and Andreas Winter (2010): *Graph-based traceability: a comprehensive approach*. In: Software & Systems Modeling, vol. 9(4), pp. 473–492. Cited on pages 70, 85, 102 and 497.

[Seibel, Neumann and Giese, 2010] Andreas Seibel, Stefan Neumann and Holger Giese (2010): *Dynamic hierarchical mega models: Comprehensive traceability and its efficient maintenance*. In: Software and Systems Modeling, vol. 9(4), pp. 493–528. Cited on pages 92, 486 and 497.

[Selic, 2011] Bran V Selic (2011): *A Short Catalogue of Abstraction Patterns for Model-Based Software Engineering*. In: International Journal of Software Informatics, vol. 5(1), pp. 313–334. Cited on page 499.

[Selonen and Kettunen, 2007] Petri Selonen and Markus Kettunen (2007): *Metamodel-Based Inference of Inter-Model Correspondence*. In: 11th European Conference on Software Maintenance and Reengineering (CSMR'07), pp. 71–80. Cited on page 102.

[Semeráth, Debreceni et al., 2016] Oszkár Semeráth, Csaba Debreceni, Ákos Horváth and Dániel Varró (2016): *Incremental backward change propagation of view models by logic solvers*. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pp. 306–316. Cited on page 120.

[Shah, Kerzhner et al., 2010] Aditya A. Shah, Aleksandr A. Kerzhner, Dirk Schaefer and Christiaan J.J. Paredis (2010): *Multi-view modeling to support embedded systems engineering in SysML*. In: LNCS, vol. LNCS 5765, pp. 580–601. Cited on page 122.

[Sharma, Tripathi and Srivastava, 2021] Garima Sharma, Vikas Tripathi and Awadhesh Srivastava (2021): *Recent Trends in Big Data Ingestion Tools: A Study*. Springer, pp. 873–881. Cited on page 140.

[Shinkawa, 2006] Yoshiyuki Shinkawa (2006): *Inter-Model Consistency in UML Based on CPN Formalism*. In: 2006 13th Asia Pacific Software Engineering Conference (APSEC'06), pp. 411–418. Cited on pages 85 and 165.

[Shvaiko, 2005] Pavel Shvaiko (2005): *A Survey of Schema-based Matching Approaches*. In: Journal on Data Semantics, vol. 3730, pp. 146–171. Cited on page 151.

[Shvaiko and Euzenat, 2013] Pavel Shvaiko and Jérôme Euzenat (2013): *Ontology Matching: State of the Art and Future Challenges*. In: IEEE Transactions on Knowledge and Data Engineering, vol. 25(1), pp. 158–176. Cited on page 143.

[Almeida da Silva, Mougenot et al., 2010] Marcos Aurélio Almeida da Silva, Alix Mougenot, Xavier Blanc and Reda Bendraou (2010): *Towards Automated Inconsistency Handling in Design Models*. In: LNCS, vol. LNCS 6051, pp. 348–362. Cited on pages 109 and 149.

[Simmonds, Perovich et al., 2015] Jocelyn Simmonds, Daniel Perovich, Maria Cecilia Bastarrica and Luis Silvestre (2015): *A megamodel for Software Process Line modeling and evolution*. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 406–415. Cited on page 92.

[Sindico, Natale and Sangiovanni-Vincentelli, 2012] Andrea Sindico, Marco Natale and Alberto Sangiovanni-Vincentelli (2012): *An Industrial System Engineering Process Integrating Model Driven Architecture and Model Based Design*. In: Model Driven Engineering Languages and Systems SE - 51, vol. 7590, pp. 810–826. Cited on pages 30 and 110.

[Skok, 2020] Fabian Skok (2020): Kombination von Datenbanken durch Synchronisation. Bachelor thesis, Carl von Ossietzky University of Oldenburg, Germany. Cited on pages 139 and 140.

[Skyttner, 2005] Lars Skyttner (2005): General Systems Theory: Problems, Perspectives, Practice. World Scientific Publishing, Singapore, 2nd edn. Cited on pages 59 and 60.

[Snoeck, Michiels and Dedene, 2003] Monique Snoeck, Cindy Michiels and Guido Dedene (2003): *Consistency by Construction: The Case of MERODE*. In: Conceptual Modeling for Novel Application Domains SE - 11, vol. 2814, pp. 105–117. Cited on page 105.

[Somogyi and Asztalos, 2020] Ferenc Attila Somogyi and Mark Asztalos (2020): *Systematic review of matching techniques used in model-driven methodologies*. In: Software and Systems Modeling, vol. 19(3), pp. 693–720. Cited on pages 151 and 494.

[Spanoudakis and Zisman, 2001] George Spanoudakis and Andrea Zisman (2001): *Inconsistency Management in Software Engineering: Survey and Open Research Issues*. In: S. K. Chang (Ed.): Handbook of Software Engineering and Knowledge Engineering, Volume I: Fundamentals, World Scientific Publishing Company, pp. 329–380. Cited on pages 32, 35, 72, 75, 79 and 105.

[Staab, Walter et al., 2010] Steffen Staab, Tobias Walter, Gerd Gröner and Fernando Silva Parreiras (2010): Model driven engineering with ontology technologies, vol. LNCS 6325. Cited on page 142.

[Stachowiak, 1973] Herbert Stachowiak (1973): Allgemeine Modelltheorie. Springer, Wien. Cited on page 59.

[Stahl, Völter et al., 2005] Thomas Stahl, Markus Völter, Jorn Bettin, Arno Haase and Simon Helsen (2005): Model-Driven Software Development: Technology, Engineering, Management. Cited on page 117.

[Steinberg, Budinsky et al., 2009] Dave Steinberg, Frank Budinsky, Marcelo Paternostro and Ed Merks (2009): EMF: Eclipse Modeling Framework. Addison Wesley, Boston, second edn. Cited on pages 85, 86, 87 and 265.

[Stephan and Cordy, 2013] Matthew Stephan and James R. Cordy (2013): *A Survey of Model Comparison Approaches and Applications.* In: Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, pp. 265–277. Cited on page 228.

[Stevens, 2008] Perdita Stevens (2008): *A landscape of bidirectional model transformations.* In: LNCS, vol. LNCS 5235, pp. 408–424. Cited on pages 30, 70, 74, 75, 111 and 489.

[Stevens, 2010] Perdita Stevens (2010): *Bidirectional model transformations in QVT: Semantic issues and open questions.* In: Software and Systems Modeling, vol. 9(1), pp. 7–20. Cited on pages 74, 104, 111, 114, 147, 149, 170, 199, 214 and 469.

[Stevens, 2013] Perdita Stevens (2013): *A simple game-theoretic approach to checkonly QVT Relations.* In: Software & Systems Modeling, vol. 12(1), pp. 175–199. Cited on page 111.

[Stevens, 2017] Perdita Stevens (2017): *Bidirectional Transformations in the Large.* In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 1–11. Cited on pages 78, 92, 105, 147, 150, 166 and 486.

[Stevens, 2018] Perdita Stevens (2018): *Is Bidirectionality Important?* In: LNCS, vol. LNCS 10890, Springer, pp. 1–11. Cited on pages 67, 106 and 150.

[Störrle, 2013] Harald Störrle (2013): *Towards clone detection in UML domain models.* In: Software & Systems Modeling, vol. 12(2), pp. 307–329. Cited on pages 45 and 461.

[Strüber, Born et al., 2017] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf and Matthias Tichy (2017): *Henshin: A Usability-Focused Framework for EMF Model Transformation Development.* In: LNCS, vol. LNCS 10373, pp. 196–208. Cited on page 86.

[Stünkel, König et al., 2018] Patrick Stünkel, Harald König, Yngve Lamo and Adrian Rutle (2018): *Multimodel correspondence through inter-model constraints.* In: Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, vol. Part F1376, pp. 9–17. Cited on pages 76, 131 and 132.

[Stünkel, König et al., 2021] Patrick Stünkel, Harald König, Yngve Lamo and Adrian Rutle (2021): *Comprehensive Systems: A formal foundation for Multi-Model Consistency Management.* In: Formal Aspects of Computing. Cited on pages 147 and 155.

[Taentzer, Ehrig et al., 2005] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Daniel Varro and Szilvia Varro-Gyapay (2005): *Model Transformations by Graph Transformations: A Comparative Study*. In: Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005. Cited on page 84.

[Taentzer, Mantz et al., 2013] Gabriele Taentzer, Florian Mantz, Thorsten Arendt and Yngve Lamo (2013): *Customizable model migration schemes for meta-model evolutions with multiplicity changes*. In: LNCS, vol. LNCS 8107, pp. 254–270. Cited on page 187.

[Tarr, Ossher et al., 1999] Peri Tarr, Harold Ossher, William Harrison and Stanley M. Sutton (1999): *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. In: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002), pp. 107–119. Cited on page 31.

[The Apache Software Foundation, 2020a] The Apache Software Foundation (2020a): *Apache HTTP Server Tutorial: .htaccess files*. Cited on page 289.

[The Apache Software Foundation, 2020b] The Apache Software Foundation (2020b): *htpasswd - Manage user files for basic authentication*. Cited on page 284.

[Thomas and Nejmeh, 1992] I. Thomas and B.A. Nejmeh (1992): *Definitions of tool integration for environments*. In: IEEE Software, vol. 9(2), pp. 29–35. Cited on pages 28, 44, 96, 97, 124 and 460.

[Tisi, Martínez et al., 2011] Massimo Tisi, Salvador Martínez, Frédéric Jouault and Jordi Cabot (2011): *Lazy execution of model-to-model transformations*. In: LNCS, vol. LNCS 6981, pp. 32–46. Cited on page 69.

[Tos, Mokadem et al., 2015] Uras Tos, Riad Mokadem, Abdelkader Hameurlain, Tolga Ayav and Sebnem Bora (2015): *Dynamic replication strategies in data grid systems: a survey*. In: Journal of Supercomputing, vol. 71(11), pp. 4116–4140. Cited on page 45.

[Tran, Kato and Hu, 2020] Van-Dang Tran, Hiroyuki Kato and Zhenjiang Hu (2020): *Programmable view update strategies on relations*. In: Proceedings of the VLDB Endowment, vol. 13(5), pp. 726–739. Cited on pages 141 and 199.

[Trollmann and Albayrak, 2015] Frank Trollmann and Sahin Albayrak (2015): *Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models*. In: LNCS, vol. 9152, pp. 214–229. Cited on page 111.

[Trollmann and Albayrak, 2016] Frank Trollmann and Sahin Albayrak (2016): *Extending Model Synchronization Results from Triple Graph Grammars to Multiple Models*. In: LNCS, vol. 9765, pp. 91–106. Cited on page 111.

[Troya, Moreno et al., 2021] Javier Troya, Nathalie Moreno, Manuel F. Bertoa and Antonio Vallecillo (2021): *Uncertainty representation in software models: a survey*. In: Software and Systems Modeling, vol. 20(4), pp. 1183–1213. Cited on page 107.

[Tunjic and Atkinson, 2015] Christian Tunjic and Colin Atkinson (2015): *Synchronization of Projective Views on a Single-Underlying-Model*. In: Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering - MORSE/VAO '15, pp. 55–58. Cited on page 125.

[Tunjic, Atkinson and Draheim, 2018] Christian Tunjic, Colin Atkinson and Dirk Draheim (2018): *Supporting the Model-Driven Organization Vision through Deep, Orthographic Modeling.* In: Enterprise Modelling and Information Systems Architectures-an International Journal, vol. 13(SI), pp. 1–39. Cited on pages 126 and 145.

[Usman, Nadeem et al., 2008] Muhammad Usman, Aamer Nadeem, Tai-hoon Kim and Eun-suk Cho (2008): *A Survey of Consistency Checking Techniques for UML Models.* In: 2008 Advanced Software Engineering and Its Applications, pp. 57–62. Cited on pages 105 and 136.

[Van Der Straeten, Mens et al., 2003] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds and Viviane Jonckers (2003): *Using Description Logic to Maintain Consistency between UML Models.* In: LNCS, vol. 2863, pp. 326–340. Cited on pages 43, 109, 150, 228 and 230.

[Vangheluwe, de Lara and Mosterman, 2002] Hans Vangheluwe, Juan de Lara and Pieter J. Mosterman (2002): *An Introduction to Multi-Paradigm Modelling and Simulation.* In: Proceedings of the AIS'2002 conference (AI, Simulation and Planning in High Autonomy Systems), Lisboa, Portugal. Cited on page 45.

[Vanhooff, Ayed et al., 2007] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen and Yolande Berbers (2007): *UniTI: A Unified Transformation Infrastructure.* In: LNCS, vol. LNCS 4735, pp. 31–45. Cited on page 204.

[Vanhooff and Berbers, 2005] Bert Vanhooff and Yolande Berbers (2005): *Supporting Modular Transformation Units with Precise Transformation Traceability Metadata.* In: ECMDATW Workshop SINTEF, (point 2), pp. 15–27. Cited on page 70.

[Vara Larsen, DeAntoni et al., 2015] Matias Ezequiel Vara Larsen, Julien DeAntoni, Benoit Combemale and Frederic Mallet (2015): *A Behavioral Coordination Operator Language (BCOoL).* In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 186–195. Cited on page 82.

[Varde and Rundensteiner, 2002] Aparna S. Varde and Elke A. Rundensteiner (2002): *MEDWRAP: Consistent View Maintenance over Distributed Multi-relation Sources.* In: LNCS, vol. 2453, pp. 341–350. Cited on page 140.

[Varró, Bergmann et al., 2016] Dániel Varró, Gábor Bergmann, Ábel Hegedüs, Ákos Horváth, István Ráth and Zoltán Ujhelyi (2016): *Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework.* In: Software & Systems Modeling, vol. 15(3), pp. 609–629. Cited on page 120.

[Vermolen, Wachsmuth and Visser, 2012] Sander D. Vermolen, Guido Wachsmuth and Eelco Visser (2012): *Reconstructing complex metamodel evolution.* In: LNCS, vol. LNCS 6940, pp. 201–221. Cited on pages 85 and 229.

[Viotti and Vukolić, 2016] Paolo Viotti and Marko Vukolić (2016): *Consistency in Non-Transactional Distributed Storage Systems.* In: ACM Computing Surveys, vol. 49(1), pp. 1–34. Cited on page 45.

[Visser, 2008] Joost Visser (2008): *Coupled Transformation of Schemas, Documents, Queries, and Constraints.* In: Electronic Notes in Theoretical Computer Science, vol. 200(3), pp. 3–23. Cited on page 187.

[Viyovic, Maksimovic and Perisic, 2014] Vladimir Viyovic, Mirjam Maksimovic and Branko Perisic (2014): *Sirius: A rapid development of DSM graphical editor.* In: IEEE 18th International Conference on Intelligent Engineering Systems INES 2014, pp. 233–238. Cited on pages 86, 137 and 138.

[Voelter, 2010] Markus Voelter (2010): *Embedded Software Development with Projectional Language Workbenches.* In: Development, pp. 32–46. Cited on pages 137 and 138.

[Voelter, Kolb et al., 2019] Markus Voelter, Bernd Kolb, Tamás Szabó, Daniel Ratiu and Arie van Deursen (2019): *Lessons learned from developing mbeddr: a case study in language engineering with MPS.* In: Software & Systems Modeling, vol. 18(1), pp. 585–630. Cited on page 137.

[Voelter, Siegmund et al., 2014] Markus Voelter, Janet Siegmund, Thorsten Berger and Bernd Kolb (2014): *Towards User-Friendly Projectional Editors.* In: Software Language Engineering (SLE 2014), LNCS, vol. LNCS 8706, pp. 41–61. Cited on page 137.

[Voelter, Warmer and Kolb, 2015] Markus Voelter, Jos Warmer and Bernd Kolb (2015): *Projecting a Modular Future.* In: IEEE Software, vol. 32(5), pp. 46–52. Cited on page 137.

[Vogel-Heuser, Fay et al., 2015] Birgit Vogel-Heuser, Alexander Fay, Ina Schaefer and Matthias Tichy (2015): *Evolution of software in automated production systems: Challenges and research directions.* In: Journal of Systems and Software, vol. 110, pp. 54–84. Cited on pages 122, 461 and 487.

[Von Pilgrim, Vanhooff et al., 2008] Jens Von Pilgrim, Bert Vanhooff, Immo Schulz-Gerlach and Yolande Berbers (2008): *Constructing and visualizing transformation chains.* In: LNCS, vol. LNCS 5095, pp. 17–32. Cited on page 497.

[Wachsmuth, 2007] Guido Wachsmuth (2007): *Metamodel adaptation and model co-adaptation.* In: ECOOP 2007–Object-Oriented Programming, vol. 4609, pp. 600–624. Cited on pages 46, 85, 195 and 241.

[Wachsmuth, Konat and Visser, 2014] Guido H. Wachsmuth, Gabriel D.P. Konat and Eelco Visser (2014): *Language Design with the Spoofax Language Workbench.* In: IEEE Software, vol. 31(5), pp. 35–43. Cited on page 137.

[Wagelaar, Iovino et al., 2012] Dennis Wagelaar, Ludovico Iovino, Davide Di Ruscio and Alfonso Pierantonio (2012): *Translational semantics of a co-evolution specific language with the EMF transformation virtual machine.* In: LNCS, vol. LNCS 7307, pp. 192–207. Cited on page 194.

[Wagelaar, Van Der Straeten and Deridder, 2010] Dennis Wagelaar, Ragnhild Van Der Straeten and Dirk Deridder (2010): *Module superimposition: a composition technique for rule-based model transformation languages.* In: Software & Systems Modeling, vol. 9(3), pp. 285–309. Cited on page 203.

[Walter and Ebert, 2009] Tobias Walter and Juergen Ebert (2009): *Combining DSLs and Ontologies Using Metamodel Integration.* In: Domain-Specific Languages, Proceedings, vol. 5658, pp. 148–169. Cited on pages 131, 206 and 279.

[Walter, Parreiras and Staab, 2014] Tobias Walter, Fernando Silva Parreiras and Steffen Staab (2014): *An ontology-based framework for domain-specific modeling.* In: Software & Systems Modeling, vol. 13(1), pp. 83–108. Cited on page 143.

[Wang, Crawl et al., 2015] Jianwu Wang, Daniel Crawl, Shweta Purawat, Mai Nguyen and Ilkay Altintas (2015): *Big data provenance: Challenges, state of the art and opportunities.* In: 2015 IEEE International Conference on Big Data (Big Data), pp. 2509–2516. Cited on page 141.

[Wang, Gibbons and Wu, 2011] Meng Wang, Jeremy Gibbons and Nicolas Wu (2011): *Incremental updates for efficient bidirectional transformations.* In: ACM SIGPLAN Notices, vol. 46(9), p. 392. Cited on page 135.

[Wasserman, 1990] Anthony I. Wasserman (1990): *Tool integration in software engineering environments.* In: LNCS, vol. LNCS 467, pp. 137–149. Cited on pages 44, 45 and 96.

[Wegner, 2021] Stefan Wegner (2021): Modellgetriebene Vernetzung von Informationsquellen. Bachelor thesis, Carl von Ossietzky University of Oldenburg, Germany. Cited on pages 387, 388 and 496.

[Wei, Kolovos et al., 2016] Ran Wei, Dimitrios S. Kolovos, Antonio Garcia-Dominguez, Konstantinos Barmpis and Richard F. Paige (2016): *Partial loading of XMI models.* In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pp. 329–339. Cited on page 89.

[Wenzel, 2014] Sven Wenzel (2014): *Unique identification of elements in evolving software models.* In: Software and Systems Modeling, vol. 13(2), pp. 679–711. Cited on page 230.

[Werner and Aßmann, 2018] Christopher Werner and Uwe Aßmann (2018): *Model Synchronization with the Role-oriented Single Underlying Model.* In: Proceedings of MODELS 2018 Workshops co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), vol. 2245, pp. 62–71. Cited on pages 129 and 131.

[Werner, Bergmann et al., 2019] Christopher Werner, Rico Bergmann, Johannes Mey, René Schöne and Uwe Aβmann (2019): *Transforming truth tables to binary decision diagrams using the role-based synchronization approach.* In: CEUR Workshop Proceedings, vol. 2550, pp. 45–50. Cited on page 131.

[Werner, Schön et al., 2018] Christopher Werner, Hendrik Schön, Thomas Kühn, Sebastian Götz and Uwe Aßmann (2018): *Role-based runtime model synchronization.* In: Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018, pp. 306–313. Cited on page 131.

[Werner, Wimmer and Aßmann, 2019] Christopher Werner, Manuel Wimmer and Uwe Aßmann (2019): *A Generic Language for Query and Viewtype Generation By-Example.* In: 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 379–386. Cited on page 494.

[Westfechtel, 2014] Bernhard Westfechtel (2014): *Merging of EMF models: Formal foundations.* In: Software and Systems Modeling, vol. 13(2), pp. 757–788. Cited on page 231.

[Whittle, Hutchinson et al., 2013] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden and Rogardt Heldal (2013): *Industrial adoption of model-driven engineering: Are the tools really the problem?* In: LNCS, vol. LNCS 8107, pp. 1–17. Cited on page 37.

[Wiederhold, 1999] Gio Wiederhold (1999): *Mediation to Deal with Heterogeneous Data Sources.* In: LNCS, vol. 1580, pp. 1–16. Cited on page 496.

[Wimmer, Kusel et al., 2010] M. Wimmer, A. Kusel, J. Schoenboeck, W. Retschitzegger, W. Schwinger and G. Kappel (2010): *On using inplace transformations for model co-evolution*. In: CEUR Workshop Proceedings, vol. 711, pp. 65–78. Cited on pages 194 and 198.

[Wimmer, Schauerhuber et al., 2011] Manuel Wimmer, Andrea Schauerhuber, Gerti Kappel, Werner Retschitzegger, Wieland Schwinger and Elizabeth Kapsammer (2011): *A survey on UML-based aspect-oriented design modeling*. In: ACM Computing Surveys, vol. 43(4), pp. 1–33. Cited on page 132.

[Winter, 2000] Andreas Winter (2000): Referenz-Metaschema für visuelle Modellierungssprachen. Deutscher Universitäts-Verlag, Wiesbaden. Cited on pages 27 and 470.

[Wittern, Cha et al., 2019] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart and Louis Mandel (2019): *An Empirical Study of GraphQL Schemas*. In: LNCS, vol. LNCS 11895, Springer, pp. 3–19. Cited on page 200.

[Wolny, Mazak et al., 2020] Sabine Wolny, Alexandra Mazak, Christine Carpella, Verena Geist and Manuel Wimmer (2020): *Thirteen years of SysML: a systematic mapping study*. In: Software and Systems Modeling, vol. 19(1), pp. 111–169. Cited on page 86.

[Wortmann, Barais et al., 2020] Andreas Wortmann, Olivier Barais, Benoit Combemale and Manuel Wimmer (2020): *Modeling languages in Industry 4.0: an extended systematic mapping study*. In: Software and Systems Modeling, vol. 19(1), pp. 67–94. Cited on pages 28 and 499.

[Wu and Farrell, 2021] Hao Wu and Marie Farrell (2021): *A formal approach to finding inconsistencies in a metamodel*. In: Software and Systems Modeling, vol. 20(4), pp. 1271–1298. Cited on page 490.

[Xiong, Hu et al., 2009] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi and Hong Mei (2009): *Supporting automatic model inconsistency fixing*. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - E, p. 315. Cited on page 118.

[Xiong, Liu et al., 2007] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi and Hong Mei (2007): *Towards automatic model synchronization from model transformations*. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07, p. 164. Cited on page 113.

[Xiong, Song et al., 2013] Yingfei Xiong, Hui Song, Zhenjiang Hu and Masato Takeichi (2013): *Synchronizing concurrent model updates based on bidirectional transformation*. In: Software and Systems Modeling, vol. 12(1), pp. 89–104. Cited on page 463.

[Yie, Casallas et al., 2009a] Andres Yie, Rubby Casallas, Dirk Deridder and Dennis Wagelaar (2009a): *A practical approach to multi-modeling views composition*. In: 3rd International Workshop on Multi-Paradigm Modeling (MPM 2009), vol. 21, pp. 1–10. Cited on page 133.

[Yie, Casallas et al., 2010] Andrés Yie, Rubby Casallas, Dirk Deridder and Dennis Wagelaar (2010): *Deriving correspondence relationships to guide a multi-view heterogeneous composition*. In: LNCS, vol. LNCS 6002, pp. 225–239. Cited on page 133.

[Yie, Casallas et al., 2009b] Andrés Yie, Rubby Casallas, Dennis Wagelaar and Dirk Deridder (2009b): *An Approach for Evolving Transformation Chains.* In: LNCS, vol. LNCS 5795, pp. 551–555. Cited on page 166.

[Yue and Ali, 2016] Tao Yue and Shaukat Ali (2016): *Empirically evaluating OCL and Java for specifying constraints on UML models.* In: Software & Systems Modeling, vol. 15(3), pp. 757–781. Cited on page 268.

[Zan, Pacheco and Hu, 2014] Tao Zan, Hugo Pacheco and Zhenjiang Hu (2014): *Writing bidirectional model transformations as intentional updates.* In: Companion Proceedings of the 36th International Conference on Software Engineering, pp. 488–491. Cited on pages 114 and 199.

[Zhang and Moller-Pedersen, 2013] Weiqing Zhang and Birger Moller-Pedersen (2013): *Tool Integration Models.* In: 2013 20th Asia-Pacific Software Engineering Conference (APSEC), pp. 485–494. Cited on page 119.

[Zheng and Taylor, 2013] Yongjie Zheng and Richard N Taylor (2013): *A classification and rationalization of model-based software development.* In: Software & Systems Modeling, vol. 12(4), pp. 669–678. Cited on page 499.

[Zimmermann and Reussner, 2018] Daniel Zimmermann and Ralf H Reussner (2018): *Automated Consistency Preservation in Electronics Development of Cyber-Physical Systems.* Springer, pp. 506–511. Cited on page 129.