Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Model Difference Representation

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

Doktors der Ingenieurswissenschaften (Dr.-Ing.)

angenommene Dissertation

von Herrn Dilshodbek Kuryazov

geboren am 24.09.1985 in Usbekistan

Oldenburg, February 2019

Gutachter:

Prof. Dr. Andreas Winter

Prof. Dr. Ralf Reussner

Weitere Gutachter:

Prof. Dr. Khakimjon Zayniddinov

Tag der Disputation:

11.02.2019

Abstract.

As a software engineering paradigm, Model-Driven Software Engineering (MDSE) is the modern day style of software development which supports well-suited abstraction concepts to software development activities. It intends to improve the productivity of the software development, maintenance activities, and communication among various team members and stakeholders. In MDSE, software models which also comprise source code are the central artifacts. MDSE brings several main advantages such as a productivity boost, models become a single point of truth, and they are reusable and automatically kept up-to-date with the code they represent.

Software models (e.g., in UML) are the key artifacts in MDSE activities. Software models are well-suited for designing, developing and producing large-scale software projects. In order to cope with constantly growing amounts of software artifacts and their complexity, software systems to be developed and maintained are initially shifted to abstract forms using modeling concepts. Software models are the documentation and implementation of software systems being developed and evolved.

Like the source code of software systems, software models are constantly changed during their development and evolutionary life-cycle. They are constantly evolved and maintained undergoing diverse changes such as extensions, corrections, optimization, adaptations and other improvements. All development and maintenance activities contribute to the evolution of software models resulting in several subsequent revisions. During the evolution process, models become large and complex raising a need for concurrent collaboration of several developers, designers and stakeholders (i.e., collaborators) on shared models, i.e., *Collaborative Modeling*.

Constantly modifying software models results in several subsequent *revisions* of the same modeling artifact. The differences between subsequent model revisions are identified and represented in model repositories as *modeling deltas*. Modeling deltas serve as information resources in further manipulations and analysis of software models. Modeling deltas representing changes between the subsequent revisions of models are the first-class entities in storing the histories of model changes in model repositories.

As models become large and complex over time, maintaining and developing such models require the concurrent collaboration of several developers in real-time. Concurrent collaboration is usually dedicated to instantly creating, modifying and maintaining huge, shared and centralized models in real-time by a team of collaborators. Thus, the changes made by collaborators have to be continually detected and synchronized among the several concurrent copies of that model. As long as synchronization has to occur in real-time, performance of interaction matters. Thus, model changes have to be represented and synchronized using very compact notations. Concurrent model copies can be differentiated by changes represented in small *modeling deltas*. The required performance of synchronization in real-time can be achieved by exchanging modeling deltas.

In collaborative modeling, *modeling deltas* are the first-class entities and play an essential role in storing, exchanging and synchronizing the changes between the subsequent and parallel revisions of evolving models. Thus, the efficient representation of changes in modeling deltas is crucial. An *efficient change representation notation* is needed for collaborative modeling which serves as the common underlying base-technology to represent modeling deltas.

This thesis introduces a *Difference Language (DL)* to the problem of model difference representation in collaborative modeling. The proposed DL is meta-model generic, operation-based, modeling tool generic, reusable, applicable, and extensible. DL is conceptually a family of domain-specific languages. Specific DLs for specific modeling languages can be generated from the meta-models of these modeling languages. Then, changes in instance models are described in terms of DL in modeling deltas. The DL-based modeling deltas consist of the executable descriptions of model changes.

Associated technical support further focuses on providing *a catalog of supplementary services* which allow for reusing and exploiting modeling deltas represented by DL. The DL-based modeling deltas are calculated, applied and reused by these DL services. These supplementary services extend the application areas of DL.

As the proof of the concept, the DL approach is applied to several applications areas in this thesis. *Concurrent collaborative modeling*, *sequential collaborative modeling* and *model history analysis* applications are taken into account in this thesis as the application areas of the DL-based difference representation approach. These applications are built by the specific orchestrations of the DL services following a certain data- and control-flow.

Besides this research work provides the open source, prototypical implementations of the DL services, applications and present empirical case studies and experiments for evaluating their usability and applicability.

Kurzfassung.
Als Paradigma der Softwareentwicklung ist Model-Driven Software Engineering (MDSE) eine moderne Form der Softwareentwicklung, die Softwareentwicklungsaktivitäten durch geeignete Abstraktionskonzepte unterstützt. Es zielt darauf ab, die Produktivität der Softwareentwicklung sowie Wartungsaktivitäten und Kommunikation zwischen verschiedenen Teammitgliedern und Stakeholdern zu verbessern. Im Kontext von MDSE sind Modelle, die auch Quellcode umfassen können, die zentralen Artefakte. MDSE bietet mehrere wichtige Vorteile, u.a. einen Produktivitätsschub. Modelle werden zum zentralen Entwicklungsmittel. Sie sind wiederverwendbar und werden automatisch mit dem Code aktualisiert, den sie repräsentieren.

Softwaremodelle (z. B. in der UML) sind die wichtigsten Artefakte in MDSE-Aktivitäten. Softwaremodelle eignen sich hervorragend zum Entwerfen und Entwickeln von großen Softwareprojekten. Um die ständig wachsenden Mengen von Softwareartefakten und deren Komplexität zu bewältigen, werden zu entwickelnde und zu wartende Softwaresysteme als Modelle abstrahiert. Softwaremodelle können zur Dokumentation und Implementierung von Softwaresystemen, die entwickelt und weiterentwickelt werden, erstellt werden.

Wie Quellcode von Softwaresystemen werden Softwaremodelle während ihrer Entwicklung und ihres evolutionären Lebenszyklus ständig verändert. Sie werden ständig weiterentwickelt und gewartet und durchlaufen diverse Änderungen wie Erweiterungen, Korrekturen, Optimierungen, Anpassungen und andere Verbesserungen. Alle Entwicklungs- und Wartungsaktivitäten tragen zur Entwicklung von Softwaremodellen bei, was zu mehreren sequentiellen Revisionen führt. Während des Entwicklungsprozesses werden Modelle zu großen und komplexen Artefakten, was die gleichzeitige (d.h. kollaborative) Zusammenarbeit von mehreren Entwicklern, Designern und Interessensvertretern an gemeinsamen Modellen, d.h. *Kollaborative Modellierung*, erfordert.

Das wiederholte Ändern von Softwaremodellen führt zu mehreren nachfolgenden *Revisionen* desselben Modellierungsartefakts. Die Differenzen zwischen sequentiellen Modellrevisionen werden identifiziert und in Repositories als *Modellierungsdeltas* gespeichert. *Modellierungsdeltas* dienen als Informationsressourcen bei weiteren Manipulationen und Analysen von Softwaremodellen. Modellierungsdeltas, die Veränderungen zwischen sequentiellen Revisionen von Modellen darstellen, sind zentralen Entitäten zum Speichern der Historien von Modelländerungen in Repositories.

Da Modelle im Laufe der Zeit groß und komplex werden, erfordert die Wartung und Entwicklung solcher Modelle die gleichzeitige Zusammenarbeit mehrerer Entwickler in Echtzeit. Daher müssen die von Entwicklern vorgenommenen Änderungen kontinuierlich zwischen mehreren Kopien dieses Modells erkannt und synchronisiert werden. Immer wenn die Synchronisation in Echtzeit stattfinden muss, spielt die Interaktion zwischen Entwicklern eine wichtige Rolle. Modelländerungen müssen daher mit sehr kompakten Notationen dargestellt und synchronisiert werden. Gleichzeitige Modellinstanzen können durch Änderungen in kleinen

Modelldeltas unterschieden werden. Die erforderliche Synchronisationsleistung in Echtzeit kann durch den Austausch kleiner Modelldeltas erreicht werden.

In der kollaborativen Modellierung sind *Modellierungsdeltas* Entitäten erster Klasse und spielen eine wesentliche Rolle beim Speichern, Austauschen und Synchronisieren der Änderungen zwischen den sequentiellen und parallelen Revisionen von sich entwickelnden Modellen. Daher ist die effiziente und kompakte Repräsentation von Modellierungsdeltas entscheidend. Eine *effiziente Repräsentationsnotation für Änderungen* wird für die kollaborative Modellierung benötigt, die als die gemeinsame zugrunde liegende Basistechnologie zur Repräsentation von Modellierungsdeltas dient.

Diese Dissertation führt eine *Difference Language (DL)* ein, um Modelldifferenzen zu repräsentatieren. Die vorgeschlagene DL ist generisch, operatorbasiert, metamodellgenerisch, wiederverwendbar, anwendbar und erweiterbar. DL ist konzeptionell eine Familie von domänenspezifischen Sprachen. Spezifische DLs für spezifische Modellierungssprachen können aus den Metamodellen dieser Modellierungssprachen generiert werden. Die DL-basierten Modellierungsdeltas bestehen aus ausführbaren Beschreibungen von Modelländerungen.

Die zugehörige technische Unterstützung konzentriert sich auch auf die Bereitstellung *eines Katalogs ergänzender DL Dienste (Services)*, die die Wiederverwendung und Nutzung von Modellierungsdeltas ermöglichen. Es werden Services zum Berechnen, Ausführen und zur Wiederverwenden von Modellierungsdeltas bereitgestellt. Diese Services werden wiederum für verschiedene Anwendungsfälle wiederverwendet.

Modellversionskontrolle, *gleichzeitige Modellierung in Echtzeit* und *Analyse der Modellhistorie* werden in dieser Arbeit als Anwendungsfälle des DL-basierten Differenzrepräsentation Ansatzes berücksichtigt. Diese Anwendungen werden durch spezifische Orchestrierungen der DL-Services sowohl für Daten als auch für den Kontrollfluss umgesetzt.

Darüber hinaus umfasst die Arbeit eine prototypische Implementierung der DL-Services als Open Source, Anwendungen und aktuelle empirische Fallstudien und Experimente zur Bewertung ihrer Relevanz, Nützlichkeit und Anwendbarkeit.

Acknowledgements

This thesis would not have been possible without the valuable contributions of many people. I owe my gratitude to all those, who have supported me during the time when I worked on this thesis and who made this time to be a precious experience for me.

My deepest gratitude is to my supervisor Prof. Dr. Andreas Winter, who provided me with every kind of support, assistance and freedom to explore. He constantly encouraged me to aim high and to keep pushing forward, while also giving me the guidance to get back on track when I struggled. I am indebted to Prof. Dr. Andreas Winter for the support with valuable feedback and always kindly encouraged me to succeed with my thesis and scientific career. I am very grateful to Prof. Dr. Ralf Reussner (my second supervisor) and Prof. Dr.-Ing. habil. Jorge Marx Gómez (Erasmus Mundus Target II coordinator at the University of Oldenburg) who had been both great mentors throughout my time working on my thesis.

I am deeply thankful to my colleagues Jan Jelschen, Johannes Meier, Christian Schönberg and Ruthbetha Kateule, who have always been there to share both my successes and my failings unconditionally. They always provided me with their help, advice, and their helpful comments during presentations. Furthermore, I am very thankful to students in the project group `Kotelett` for their contributions in prototypical development of an application for this thesis.

I would like to thank the full PhD scholarship program of the Erasmus Mundus Target II project and DAAD STIBET contact scholarship for granting me the scholarship and to come to Germany to write my PhD thesis.

Last but not least, I am very thankful to my beloved ones; my parents, my wife and my daughter for supporting me and always being there for me.

Contents

Abstract iii

Kurzfassung v

Acknowledgements vii

Contents viii

I Motivation and Challenges 1

1 Introduction 5
1.1 Research Objective 9
1.2 Outline of the Thesis 11

II Foundations 15

2 Basic Concepts 19
2.1 Model-Driven Software Engineering 20
2.2 Domain Specific Languages 23
2.3 Model Transformations 25
2.4 Technical Spaces 27
2.4.1 JGraLab – Java Graph Laboratory 28
2.4.2 EMF – Eclipse Modeling Framework 30
2.4.3 Further Technologies 32
2.5 Summary 33

3 Collaborative Development Use Cases 35
3.1 Concurrent Collaboration 39
3.1.1 Concurrent Text-Driven Collaboration 40
3.1.2 Concurrent Model-Driven Collaboration 44
3.1.3 Required Support 46
3.1.4 Expected Benefits by Difference Language 47

3.2 Sequential Collaboration 48
3.2.1 Sequential Text-Driven Collaboration 49
3.2.2 Sequential Collaborative Modeling 52
3.2.3 Required Support 54
3.2.4 Expected Benefits by Difference Language 55
3.3 History Analysis 56
3.3.1 Text-Driven History Analysis 60
3.3.2 Model History Analysis 61
3.3.3 Required Support 63
3.3.4 Expected Benefits by Difference Language 63
3.4 Summary 64

4 Related Approaches To Difference Representation 65
4.1 Model Difference Representation 66
4.1.1 Model- and Graph-based Difference Representation 67
4.1.2 Database-based Difference Representation 71
4.1.3 Text-based Difference Representation 71
4.1.4 Lessons Learned 74
4.2 Services 76
4.2.1 Difference Calculator 79
4.2.2 Difference Applier and Merger 84
4.2.3 Synchronization 89
4.2.4 Model Manager 90
4.2.5 Change Tracer 91
4.2.6 Lessons Learned 93
4.3 Requirements 94
4.4 Summary 98

III Approach 101

5 Difference Language 105
5.1 Conceptual Idea: DL Generation 108
5.2 Motivating Example 114
5.2.1 Sample Model and Model Changes 115
5.2.2 Modeling Deltas 116
5.2.3 DL Operations 121
5.3 Summary 123

6 Difference Language Services 125
6.1 Generator 127
6.2 Adapter 129
6.3 Calculator 130
6.4 Applier 133
6.5 Synchronizer 135

6.6 Manager . . . 136
6.7 Tracer . . . 137
6.8 Optimizer . . . 140
6.9 Merger . . . 142
6.10 Service Orchestration . . . 143
6.11 Summary . . . 146

IV Applications 149

7 Concurrent Collaborative Modeling 153
7.1 Reference Architecture . . . 154
7.2 Kotelett . . . 158
7.2.1 Meta-Model . . . 158
7.2.2 Concrete Architecture . . . 160
7.2.3 Kotelett Tool . . . 161
7.3 Collaborative Modeling – CoMo . . . 164
7.3.1 Meta-Model . . . 164
7.3.2 Concrete Architecture . . . 166
7.3.3 CoMo Tool . . . 167
7.4 DL Contributions . . . 169
7.5 Summary . . . 171

8 Sequential Collaborative Modeling 173
8.1 Reference Architecture . . . 174
8.2 Generic Model Versioning System – GMoVerS . . . 178
8.2.1 Meta-Model . . . 179
8.2.2 Concrete Architecture . . . 180
8.2.3 GMoVerS Tool . . . 181
8.3 Versioning Sustainability Reports . . . 182
8.3.1 Meta-Model . . . 182
8.3.2 Concrete Architecture . . . 183
8.4 DL Contributions . . . 184
8.5 Summary . . . 185

9 Model History Analysis 187
9.1 Reference Architecture . . . 188
9.2 Model History Analysis . . . 190
9.3 DL Contributions . . . 192
9.4 Summary . . . 193

V Evaluation 195

10 Validation 199

10.1 Applicability 199
10.2 Fulfillment of Requirements 201
10.3 Fulfillment of Expected Benefits 204
10.4 Summary 205

11 Conclusion **207**
11.1 Lessons Learned 207
11.2 Contributions 209

References **211**

Declaration of Authorship **226**

List of Figures

1.1 Outline of the Thesis 12

2.1 Abstraction Levels by Object Management Group 22
2.2 Architecture of Model Transformations 26
2.3 Example TGraph 30

3.1 Time/Space Taxonomy of Collaborative Development 37
3.2 Combined Architecture of Concurrent and Sequential Collaboration 38

4.1 Related Approaches 66

5.1 Overall Architecture of Approach 107
5.2 Meta-model for UML Activity Diagrams 109
5.3 Conceptualization of DL Generation 110
5.4 Abstraction of Difference Language API 111
5.5 Difference Language API for Content Part 112
5.6 Difference Language API for Layout Part 113
5.7 Example Activity Diagram under Collaborative Development 115
5.8 Backward Delta: Δ(Rev_3, Rev_2) 118
5.9 Backward Delta: Δ(Rev_2, Rev_1) 118
5.10 Forward Delta: Δ(Rev_3, Designer_1) 119
5.11 Forward Delta: Δ(Rev_3, Designer_2) 119
5.12 Active Delta: $\Delta(\emptyset$, Rev_3) 120
5.13 Conceptualization of Difference Language Operations 121

6.1 DL Services . . . 125
6.2 DL Service: Generator . . . 127
6.3 DL Service: Adapter . . . 129
6.4 DL Service: State-based Calculator . . . 132
6.5 DL Service: Calculator – Change Listener . . . 133
6.6 DL Service: Delta Applier . . . 134
6.7 DL Service: Delta Synchronizer . . . 136
6.8 DL Service: Model Manager . . . 136
6.9 DL Service: Delta Tracer . . . 138
6.10 Change Report of *Control Flow* `g5`. . . . 139
6.11 DL Service: Delta Optimizer . . . 140
6.12 DL Service: Model Merger . . . 143

7.1 Reference Architecture for Concurrent Collaborative Modeling . . . 156
7.2 UML Class Diagram Meta-model for Kotelett . . . 159
7.3 Concrete Architecture of Concurrent Collaborative Modeling `Kotelett` 160
7.4 Screenshot of `Kotelett` Tool . . . 162
7.5 DL-based Change Representation in Kotelett Tool . . . 163
7.6 UML Activity Diagram Meta-model for `CoMo` . . . 165
7.7 Concrete Architecture of Concurrent Collaborative Modeling `CoMo` . 166
7.8 Screenshot of `CoMo` Tool . . . 167

8.1 Reference Architecture for Sequential Collaborative Modeling . . . 176
8.2 Simplified Substructure of UML State Machine Meta-model . . . 179
8.3 Concrete Architecture of `GMoVerS` . . . 180
8.4 `GMoVerS` Screenshot . . . 181
8.5 Simplified Schema of Sustainability Reports . . . 183
8.6 Abstraction of Difference Language for Sustainability Reporting . . 183

9.1 Reference Architecture for Model History Analysis . . . 189

9.2 Concrete Architecture of Model History Analysis 191

9.3 Screenshot of `MoHA` 191

Part I

Motivation and Challenges

As a software engineering paradigm, *Model-Driven Software Engineering (MDSE)* has become a novel means of software development which supports well-suited abstraction concepts to software development activities. It intends to improve the productivity of the software development, maintenance activities, and communication among various stakeholders. MDSE is widely used in designing business processes, work-flows, object-oriented software systems, documentations, and implementations. In MDSE, *software models* are the central artifact of software development rather than the source code implementing the systems.

In order to cope with the large-scale and complex software systems to be developed or maintained, they are initially shifted to abstract forms using modeling concepts. *Software models* are usually more abstract representations of the planned or running software systems. Software models focus on the relevant aspects of the problem domain to be designed and developed. Simultaneously, software models are the *documentation and implementation* of the software systems being developed and evolved.

Like the source code of software systems, software models constantly evolve and are maintained in order to meet various user changes such as improvements, extensions, and optimization. Evolution and maintenance of software models require a need for collaborative work of several collaborators. There are several tools and approaches for collaboratively developing and maintaining source code-driven software systems, but they are not fully capable of handling model-driven software projects. There is a strong need for such tool support and novel approaches for software models. Model change representation lies at the core of collaborative modeling. Thus, this thesis addresses to the research problem of change representation for collaborative MDSE.

In general, this thesis is presented in five parts: Part I. Motivation and Challenges, Part II. Foundations, Part III. Approach, Part IV. Applications and Part V. Evaluation. This, the first part (Part I), aims at establishing the context of the work, its motivation and relevance, challenges and core research objectives. The second part (Part II) explains the basic concepts of MDSE, the main use cases of collaborative development, the state of the art in collaborative modeling, and derives several requirements for collaborative modeling and its change representation. The third, the approach part (Part III), describes the solution for model change representation by the conceptual idea and motivating example. The fourth part (Part IV) depicts the several applications of the proposed solution. The thesis ends up in the evaluation part (Part V) by demonstrating evaluation, validation results and drawing conclusions.

Chapter 1

Introduction

In the different periods of the software development history, various techniques and technologies were used to develop and produce software systems. The former developers used machine level commands such as raw machine code and assembly language [Kleppe et al., 2003]. Since the late sixties, software development is extended to more software engineering principles and paradigms [Naur and Randell, 1969] in order to form complex and large software systems with more structured, abstraction, and reusable technologies. It was a time of procedural programming languages [Dahl et al., 1972], which were built on top of the assembly language and were more easy to understand. From the mid-eighties, more structured and powerful object-oriented programming languages [Meyer, 1988, Kay, 1996, Dahl, 2004] started to play an essential role in software development. These programming languages follow *code-driven concepts* and technologies where programs are plain text documents [Kleppe et al., 2003].

The diversity of programming languages and the complexity of software systems has been raising more and more difficulties during the software development process. Hence, the variety of the programming languages and a vast capacity of software projects made the software development process too expensive and a far traceable task. In order to either partly or entirely deal with these challenges, software systems had to be formed by more *abstraction concepts* by taking advantages of *model-driven* technologies, principles, and concepts that software engineering offers. In order to cope with the large size and complexity of software systems, software engineering has long since become an integral and indispensable part of software development and production. It has been widely used over all phases of software development, starting from requirements elicitation and management, to software specification, architecture, design, and implementation, as well as testing and verification, maintenance and evolution.

In the late eighties, early software engineering activities started using computer-aided software engineering (CASE) tools, which focused on software development methods using general-purpose graphical programming technologies such as *meta-modeling* tools [Tolvanen, 2016, Ebert et al., 1997] and *UML-based development*

[Raumbaugh et al., 2004] including state machines, structure diagrams and activity diagrams. The CASE tools enabled developers to use graphical programs that provide more abstraction features than general-purpose programming languages. It also allowed for reducing the effort of manually coding software systems [Schmidt, 2006]. The CASE tools are the type of software development tools that are often entitled as a predecessor of *Model-Driven Software Engineering (MDSE)* (also known as Model-Driven Development (MDD) and Model-Driven Engineering (MDE)). MDSE has been strongly influenced by the Model-Driven Architecture (MDA) standardization [Miller and Mukerji, 2003] efforts driven by the Object Management Group [OMG, 2014]. The main concept of "model-driven" technologies is to make *software models* the primary artifact of software development and evolution.

In MDSE, *software models* are the central artifact of software development rather than the source code implementing the systems. Moreover, model-driven approaches aim at generating either fully or partly executable software from higher-level models by means of the model transformations, in order to avoid the redundancies involved in manual implementations. This leads to the several advantages of MDSE according to [Kleppe et al., 2003, pp. 9ff]: the productivity boost, models become a single point of truth and are automatically kept up-to-date with the code they modeled.

Software modeling (e.g., in UML – Unified Modeling Language [Raumbaugh et al., 2004]) is becoming one of the key fields of modern day software development activities which is well-suited to design, develop and produce large-scale software projects. In order to cope with the constantly growing amount of software artifacts and their complexity, software systems to be developed or maintained are initially shifted to abstract forms using modeling concepts. Software models are usually more abstract representations of the planned or running software systems. Software models can represent the more relevant aspects of the problem domain to be designed and developed. Simultaneously, software models are the *documentation and implementation* of software systems. Software models also help to understand the different aspects of the existing or planned software systems faster [Kleppe et al., 2003]. The further concepts of MDSE are explained in Section 2.1 in detail.

Like the source code of software systems, software models are constantly maintained in order to meet various user changes such as improvements, extensions, and optimization.

Software Evolution and Maintenance. Software systems including software source code and models are initially developed, thereafter they continue to *evolve* after their initial deployment. All maintenance activities after initial deployment contribute to their evolution. These activities may include adding new features or removing existing ones, fixing bugs, optimizing the existing components, etc

[Bourque et al., 2014] (discussed in Chapter 3 in detail). One of the today's software production challenge is to maintain existing, running software systems fulfilling user needs and requirements, as well as adapting to the new user and technological environments. Thus, the existing software systems are usually changed to meet new user needs and requirements. They have to be continually adapted, optimized, corrected to remain useful, operational, simple, high qualitative [Lehman, 1996]. Any type of the aforementioned changes that may occur during the initial development and evolution of software systems is referred to as software *changes* in this thesis.

Software systems usually possess several thousands of software artifacts such as design documents, software modeling artifacts, implementation artifacts, test cases, and associated datasets. Therefore, the initial development and evolution of such large-scale and complex software systems require the *combined teamwork* of project managers, software designers, developers, testers, and maintainers (i.e., *collaborators*). Designing, developing and maintaining the large-scale software systems entails the need for collaboration of a large number of developers in order to accomplish successful results [Herbsleb and Moitra, 2001]. Several collaborators involved in the software production process apply changes to the shared software systems.

The rough idea and scenario behind collaborative software development are as follows: Large and complex software systems, i.e., all artifacts produced during evolution and maintenance are usually placed in the central software repositories. A group of collaborators is assigned to that repository and each collaborator is responsible for development and maintenance of (a part of) shared software artifacts. Then, each collaborator checks out a copy of the central repository (i.e., development branch) or joins the shared repository, applies changes to the own copy and combines these changes back to the central repository or applies changes to the shared artifacts in real-time.

Depending on the nature of interaction, collaborative development systems can be divided into two main forms, namely *concurrent collaborative modeling* and *sequential collaborative modeling* [Clarence et al., 1991], [Booch and Brown, 2003].

Concurrent Collaboration. The concurrent collaboration is usually dedicated to creating, modifying and maintaining the huge, shared and centralized documents and software artifacts in real-time. The collaborators of the shared documents or software systems can access to the centralized and shared repository, and directly modify the software projects on the central repository or their local copies. There exist several approaches widely used in the concurrent collaborative editing of textual documents, e.g., Google Docs [Google Inc., 2017], Etherpad [AppJet Inc., 2017], Firepad [Firebase Inc., 2017] and many more. As long as synchronization of changes occur in real-time, performance of interaction matters. In concurrent collaboration, the changes can be constantly detected by listening users actions on a particular software project and produced in form of the small difference documents consisting of only small set of changes. These changes are usually

not stored, yet synchronized among collaborators. Concurrent collaborative development support is needed for model-driven software projects, as well. Unlike textual document editing, the increased performance of change synchronization is very crucial in concurrent collaborative modeling because of the graph-like complex structures of software models. Thus, model changes have to be identified continually, represented and synchronized using very compact notations. Concurrent collaborative modeling is enabled by *micro-versioning* [Appeldorn et al., 2018, Kuryazov et al., 2018] (Chapter 5) in this thesis.

Sequential Collaboration. Sequential collaboration intends to identify changes between the subsequent revisions of software artifacts, store and reuse these changes when needed. There exist several source code-driven sequential collaborative development approaches (also known as *version control systems*) such as Subversion [Collins-Sussman et al., 2004], Git [Swicegood, 2008], VCS [Baudivs, 2014], Monotone [Hoare et al., 2005], etc. These sequential collaborative development systems for the source code are the best aid in handling development and evolution of large-scale, complex and continually evolving software systems. The same support is needed for software models. For instance, while designing software models in concurrent collaboration, collaborators intend to store the correct and complete revision of their model and reopen it after a while to continue development and maintenance. These model management activities are facilitated by *macro-versioning* [Appeldorn et al., 2018, Kuryazov et al., 2018] in this thesis.

Model History Analysis. Typically, the concurrent and sequential collaborative development approaches for document editing and source code-driven development provide features for mining, browsing and visualizing their software repositories. These features allow collaborators to analyse and trace the histories of their evolving software systems. Likewise, software models evolve undergoing different changes resulting in several different revisions. In the same vein, the history analysis of evolving software models is quite interesting and important. Thus, both, concurrent and sequential, collaborative modeling should provide history analysis support by mining, browsing and visualizing models under evolution. As the present of models is often understandable by looking at the past, analyzing the histories of evolving software models is significant support for software stakeholders in order to be familiar with the change histories, and make further decisions and plans [Collberg et al., 2003], [Singer et al., 2005], accordingly. Collaborators are usually interested in how the development and evolution process is going on, so that they are fully able to keep the development and evolution process under their control.

There are already solid concurrent and sequential collaborative development approaches including history analysis support for source code-based software systems and textual documents. As discussed above, software models are also the subject to constant changes because of development in novel technologies, increasing user requirements, improvements, and corrections. Development and maintenance of software models need concurrent and sequential collaborative modeling, as well as model history analysis support.

It is commonly agreed that the collaborative development approaches for the source code-driven software systems do not sufficiently fit to the model-driven software projects because of a paradigm shift between source code-driven and model-driven concepts (further discussed in Section 2.1) [Cicchetti, 2008], [Steinberg et al., 2008]. Currently used tools compute and produce line by line differences (line additions/removals/keeps) between textual files [Hunt and MacIlroy, 1976]. Since software models do not follow similar principles and syntax as code-based projects, the aforementioned differentiation approaches cannot fully handle the compound and associated data structure of models. Thus, there is a need for the extensive research on difference representation techniques focusing on model-driven software engineering, especially.

To sum up, MDSE is currently playing an essential role in software development activities. Like source code-based software systems, software models are also subject to constant changes which leads to evolution. There are several tools and approaches for concurrently and sequentially developing and maintaining software source code, but they are not fully capable of handling model-driven software projects. There is a need for such tool support and novel approaches for software models. Since difference (i.e., change) representation lies at the core of these approaches, this thesis addresses the research problem of change representation for collaborative MDSE. The research objectives of this thesis are further described in Section 1.1 in detail. A brief outline of this thesis is given in Section 1.2.

1.1 Research Objective

There already exist outstanding concurrent and sequential collaborative development applications for the source code-driven of software systems and satisfactory approaches for representing differences on the basis of these applications. However, there is no comparably advanced collaborative development support for MDSE. As software models have become one of the main and widely used technologies in today's software development activities, MDSE requires a solid, configurable and adaptable concurrent and sequential collaborative modeling techniques. Since change representation is a central challenge for both collaborative modeling scenarios, there is a need for a generic, sophisticated, applicable and adaptable change representation which serves as a common underlying change representation approach for concurrent and sequential collaborative modeling, as well as model history analysis. Such advanced collaborative modeling approaches will provide efficient development and evolution of model-driven projects.

Since model change representation lies at the core of concurrent, sequential collaborative modeling and model history analysis, these collaborative modeling scenarios can be developed on the top of the same underlying common model difference representation technique.

The core objective of this thesis is to introduce a generic *Difference Language (DL)* for representing model differences in MDSE and provide *a catalog of supplementary services* for operating with DL.

The proposed DL intends to be independent of modeling languages, modeling tools and aims at reusability, applicability and extendability. In order to provide the aforementioned properties, this thesis further focuses on providing a catalog of supplementary services together with their realizations which allow for reusing and exploiting difference information represented using DL. These supplementary services extend the application areas of DL. In general, the objectives of this thesis are threefold:

- introducing a novel means to model difference representation, i.e., *Difference Language* (introduced in Chapter 5)
- providing a catalog of supplementary *services* for reusing and exploiting DL-based model differences (introduced in Chapter 6)
- extending the *applications* of the DL approach by the specific orchestration of supplementary services (explained in Part IV)

These main contributions of this thesis are discussed below in more detail.

Difference Language.
Since the model difference representation is the foundation for applications such as concurrent, sequential collaborative modeling and model history analysis, this thesis aims at supporting a meta-model generic *difference representation language*. During the initial development or evolution process of model-driven projects, new modeling artifacts can be created, existing ones can be deleted or the attribute values of existing artifacts can be changed. Thus, edit operations like the creations, deletions of modeling artifacts and the changes of attribute values are used to describe model differences. Using these edit operations, DL can form directly the executable descriptions of model changes/differences. DL aims at embedding sufficient and complete data about model changes behind each operation. DL is practically useful for further tool developers and provides several technical properties being executable, implementable in an efficient way, fully expressive, yet unambiguous, for transforming existing models to new revisions, as well as facilitate developer productivity with precise, concise and clear descriptions. This thesis introduces DL in Chapter 5.

Services.
Research in this thesis further addresses the applicability and reusability of the difference representation information in further manipulations and analysis of evolving software models. Difference information represented by DL intends to be accessible for further manipulations and analysis. To this end, this thesis provides a catalog of significant components and services (Definition 4.1) for producing and reusing DL-based model differences. For instance, several scenarios exist directly

associated with difference information such as computing differences (changes) between model revisions, applying differences to the models to transform one revision to another, tracing model change histories, synchronizing model differences among the various copies of the shared software models. These operative services described in Chapter 6 make model differences quite handy in various application areas.

Applications.
As discussed in Section 1, MDSE requires a support for the concurrent, sequential collaborative modeling and history analysis applications. This thesis develops these applications by introducing a generic underlying difference representation language for software models and a catalog of valuable services. Since supplementary services can directly operate on difference information represented by DL, it serves as the common underlying representation format for concurrent, sequential collaborative modeling and model history analysis applications. These applications, in turn, are established by the specific orchestrations of supplementary DL services. DL is applied to these applications in Part IV.

1.2 Outline of the Thesis

In Part I, this chapter has given overall insights into the research area, brief motivation and problem definition in this thesis. The remainder of this thesis consists of four parts: II Foundations, III Approach, IV Applications and V Evaluation. Figure 1.1 depicts the outline of this thesis.

Part II. Foundations. Chapter 2 initially gives a brief preamble to the basic concepts of MDSE. It further gives a brief survey on domain-specific languages, model transformations, and technical spaces that are involved in realizing and developing the approach in this thesis. Chapter 3 investigates collaborative development and history analysis approaches for source code-driven and model-driven software development and presents the current state of the art. These approaches are investigated in the same chapter in order to derive underlying common concepts, technologies, architectures and terminologies for collaborative MDSE. The same chapter defines main prerequisites for collaborative MDSE support and expected benefits by that support. Chapter 4 provides the extended literature study on the fundamentals and the state of the art on difference representation approaches for collaborative MDSE. The related work chapter studies the state of the art on additional services provided by existing related approaches. The same chapter defines the list of requirements for a difference representation support and its collaborative modeling application.

Part III. Approach. This part of the thesis is dedicated to the core ideas behind this thesis. First of all, Chapter 5 of this part explains the conceptual idea how specific difference languages can be generated for concrete modeling languages. The same chapter depicts a motivating example for the difference representation approach combining the concurrent and sequential collaborative

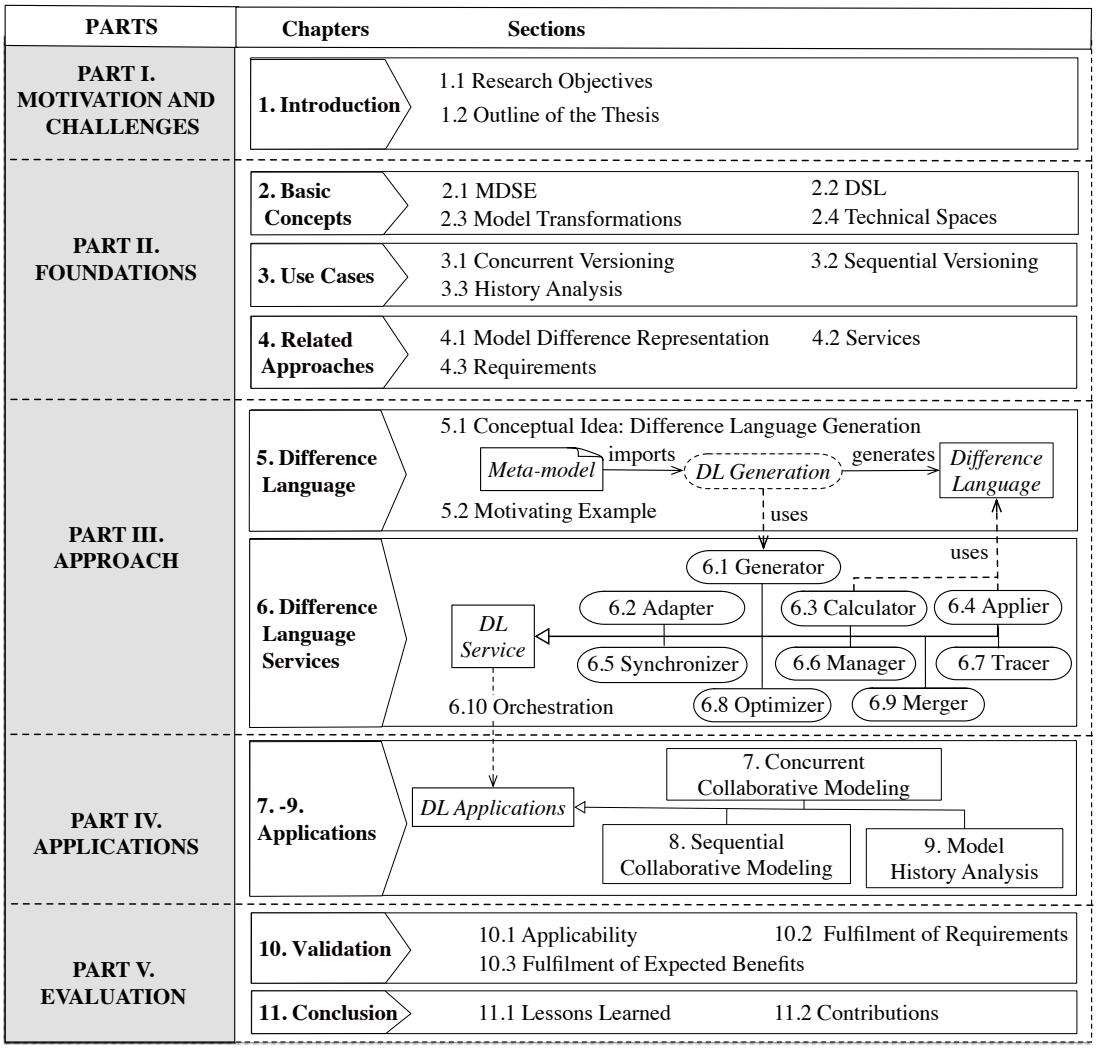

FIGURE 1.1: Outline of the Thesis

modeling scenarios. Chapter 6 introduces a catalog of the supplementary DL services. Each of these supplementary services is enlightened in a separate section.

Part IV. Applications. The applications part of the thesis discusses the main application areas of the model difference representation approach. This part clarifies concurrent collaborative modeling applications in Chapter 7, sequential collaborative modeling application in Chapter 8, and model history analysis application in Chapter 9. Each of these chapters follows a concrete structure; the overall idea and reference architecture of each collaborative modeling scenario, meta-models, the concrete architectures and tool support for the collaborative modeling applications developed in the framework of this thesis.

Part V. Evaluation. The evaluation part of the thesis explains applicability of DL in Chapter 10. The same chapter discusses the fulfillment of requirements (recalling requirements defined in Chapter 4.3) and the fulfillment of expected benefits (revisiting expected benefits explained in Chapter 3) by DL. This thesis ends up in Chapter 11 by discussing learned lessons and contributions of the research work.

The research topic and approach in this thesis is regularly discussed with research committee and has received valuable feedbacks, as well as the results have been presented at various workshops and conferences [Kuryazov et al., 2012], [Kuryazov et al., 2013], [Kuryazov, 2014], [Kuryazov and Winter, 2015b], [Kuryazov et al., 2018], [Appeldorn et al., 2018], etc.

Part II

Foundations

Before explaining the ideas behind the research work, Chapter 2 of this part gives a brief insight into the basic principles and main concepts of MDSE. As long as this thesis is dedicated to the research problem of collaborative MDSE, inspecting the essential MDSE concepts is inevitably required to clearly express the ideas, contributions and applications in this research work, later on. This thesis further intends to realize the theoretical results as the research prototypes and to validate them in various application areas. To that end, several technologies are utilized in the realization of these prototypes. The same chapter illustrates these underlying implementation technologies and concepts involved in developing the research prototypes.

Chapter 3 inspects the main use cases and scenarios of collaborative development in the context of source code-driven software development, textual document writing and collaborative modeling. It intends to study the underlying infrastructures, collaboration architectures, repository architectures of collaborative development approaches and derive some common terminologies that can be adopted for collaborative MDSE.

The problem of model difference representation is the actively discussed and extensively addressed topic among the research community in software engineering and modeling field. There is a large number of research works addressing the problem of model difference representation and its collaborative modeling applications. Chapter 4 studies the existing approaches and the state of the art in the field of model difference representation, as well as the potential operative services and scenarios they provide. Eventually, the same chapter defines a list of requirements for model difference representation and collaborative modeling to be fulfilled in this thesis.

Chapter 2

Basic Concepts

This thesis entails the realization of the research ideas and its validation in various application areas. As proof of the concept and to demonstrate applicability of the approach, the core ideas and supplementary services are implemented as the research prototypes. Several technical spaces are utilized in implementing these prototypes. This chapter illustrates these technical spaces and fundamental concepts involved in developing the research prototypes. It is inevitably required to give insights into these foundations so that the approach can clearly be expressed in remaining part. Discussions on these basic concepts assist the reader to comprehend the relevant fundamental techniques and technologies that take place in the prototypical development of the approach.

All in all, this chapter explains basic concepts such as MDSE, model transformations, domain-specific languages and technical spaces.

- *Model-Driven Software Engineering (MSDE).* As long as this thesis copes with the research problem of difference representation in collaborative MDSE, the main principles and core concepts have to be clarified at first. It contributes to comprehend the clear distinction between the source code-driven and model-driven software development paradigms. Consequently, Section 2.1 inspects the core concepts and definitions of MDSE and its abstraction levels.
- *Domain-Specific Languages.* In order to achieve efficient results in model difference representation, this thesis takes advantage of the underlying syntactical principles of domain-specific languages (DSL). Thus, Section 2.2 provides a brief insight into the basic concepts of DSLs.
- *Model Transformations.* This thesis intends to realize its representation descriptions by model transformations. It takes advantage of model transformation principles for extending applicability of its difference representations. Thus, Section 2.3 shortly reviews the potential candidates for choosing the most suitable transformation approach for implementing model difference representations.

– *Technical Spaces.* Due to realization, implementation and validation of the research idea, this thesis requests several underlying implementation techniques and technologies, i.e., technical spaces. Section 2.4 discloses the list of concrete implementation technologies employed in realizing the application areas of the approach in this thesis.

Eventually, this chapter ends up in Section 2.5 by summarizing what is the outcome of this chapter.

2.1 Model-Driven Software Engineering

The diversity of programming languages and the complexity of software systems made the software development process expensive and a far traceable task. In order to cope with the complexity and avoid redundancies in software development activities, software systems have to be documented, designed, planned and specified during both, initial development and evolution life-cycle. Either partly or entirely to deal with these design level challenges, MDSE aims at defining the existing or planned software systems by *software models* [Kleppe et al., 2003]. The key idea of "model-driven" approaches is to make software models the first-class citizens of software development and evolution, rather than the source code implementing software systems.

Definition 2.1. **Software Model.**
There are several definitions of *software models*:
A model is the simplification of a system built with an intended goal in mind [Bézivin and Gerbé, 2001];
A model is the description or specification of a system and its environment for some certain purpose [Miller and Mukerji, 2003];
A model is the coherent set of formal elements describing something built for some purpose that is amenable to a particular form of analysis [Mellor et al., 2003];
A model is a purposeful image of a system that enables observations and statements similar to those of this system and that simplifies the reality through abstraction to the respective problem-relevant aspects [Winter, 2000].

In general, these definitions (Definition 2.1) of software models can be adopted to define software models in this thesis. However, this thesis strictly requires that the construction concepts of software models have to be defined by formal specifications.

Definition 2.2. **Adopted Definition of Software Models.**
A software model is a purposeful abstraction of a planned or existing system and describes the respective problem-related aspects. A software model conforms to its predefined formal specifications (i.e., construction concepts).

Furthermore, the model-driven approaches contribute to avoiding the redundancies involved in the manual implementation of software systems by generating executable software systems from higher-level models by means of model transformations. According to [Kleppe et al., 2003], the use of model-driven approaches provides several advantages such as productivity boost, models become a single point of truth, portability, and interoperability. In MDSE, models are either descriptive, i.e., they have been derived from a system under study (SUS), or they are prescriptive, which means a system under development (SUD) is derived from models [Aßmann et al., 2006], [Seidewitz, 2003].

Models form the basis to specify the design of systems and automatically generate an executable software as well as well-formed documentations for software systems. Thereby, model designers may build models that are less limited to underlying implementation techniques and technologies. MDSE promises to raise the efficiency and ease of developing a software system by abstraction. Although, MDSE is a promising approach to cope with the ever-growing complexity of systems. They are used to understand and trace system aspects in the different viewpoint of users. According to their descriptions, software models are classified into *dynamic models* where a model represents or describes some behavioral aspects, and *static models* where a model describes the structural aspects or even changes of real-world systems [Bezivin, 2005]. For instance, UML class diagrams, UML activity diagrams [Raumbaugh et al., 2004] or the entity-relationship [Chen, 1976] can be used for describing object-oriented software systems [Rentsch, 1982], the control- and object-flows or the relational databases, respectively.

Meta-Layers.
Like grammars specify programming languages, modeling concepts are always expressed by modeling language specifications. For instance, a software system written in Java programming language conforms to the Java grammar which is described, e.g., by the Extended Backus-Naur Form (EBNF) meta-syntax notations. Similarly, in MDSE software models are specified according to their "grammars" called *meta-models*. A meta-model depicts the collection of notations that is used to describe the simplified abstraction of a particular domain. It can usually be considered as the set of rules for producing the correct and legal abstractions of the existing or planned real-world (software) systems. Meta-models prescribe the abstract syntax for modeling languages by means of elements and relations between them. In the same vein, a modeling language defining a specific domain and model should be complete and unambiguous to the purpose it has been conceived. If models are precise, more useful and effective, software artifacts can be derived from these models. A model is defined as the description of (part of) a real-world system written in *a well-defined modeling language* [Kleppe et al., 2003, p. 16].

Definition 2.3. **Well-Defined Language.**
A *well-defined language* is a language which is suitable for automated interpretation by a computer. In the MDSE context, languages are defined by the meta-modeling mechanisms [Kleppe et al., 2003, p. 114].

In order to highly benefit from the adoption of MDSE techniques, all information related to the problem domains have to be described by means of the some form of abstraction. This eventually requires the precise specification of each abstraction. To cope with this task, several meta-models are introduced supporting such new languages/specifications. In turns, that collection of meta-models have to be specified by means of rigorous manner called *meta-meta-model* which all the languages can be derived from and believed to be the minimum set of concepts. Conclusively, meta-meta-models are used to describe themselves. To this end, Object Management Group (OMG) [OMG, 2014] has introduced the four-level architecture as depicted in Figure 2.1, which organizes artifacts in a hierarchy of model layers.

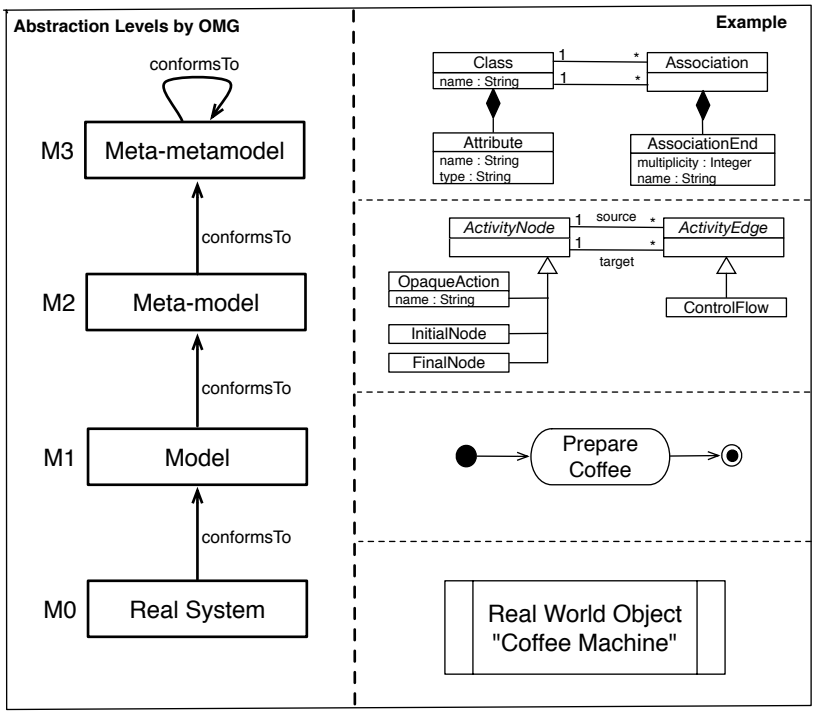

FIGURE 2.1: Abstraction levels

By looking at these abstraction levels, the M0 level which is more bottom level is always a starting point for designing the system under study. Therefore, these abstraction levels are explained starting from M0, the real-world systems to the M3 top of the higher level of abstraction. Below, a *coffee machine* example is modeled as the system under study:

– *System Under Study (Level M0).* This might be the systems that are under study, development or evolution, i.e., real-world systems that are being modeled or software systems that have to be modeled.

– *Models (Level M1).* Software models are the abstract descriptions of real-world systems. The structure and behavior of any system under study is modeled using a collection of concepts defined on meta-layer M2. The example on layer M1 shows the model of the coffee machine with one *opaque action*, *initial node* and *final node* as well as the *control flow* edges connecting these activity nodes. The example is designed using the UML activity diagram notations [Raumbaugh et al., 2004, pp. 115ff].
– *Meta-models (Level M2).* This layer supports the collections of constructions and concepts for models to design systems. The models on layer M1 conform to this meta-layer adding extra information about systems. For instance, the example displays the simplified version of the UML activity diagram meta-model used for modeling the M1 layer. On this level, the notation is used to define models. Usually, UML class diagrams are widely used to design meta-models in UML.
– *Meta-meta-models (Level M3).* The M3 abstraction level consists of the core constructions of the domain modeling language. Finally, this layer holds a reflexively defined model of information at the level M2. The level M3 defines the collection of meta-models which all languages can be derived from and believed to be the minimum set of concepts. Meta-meta-models are used to describe themselves. The examples of meta-meta-models are OMG/MOF [MOF, 2003], EMF/Ecore [Steinberg et al., 2008] and KM3 [Jouault and Bézivin, 2006].

It is not required to refer to any further abstraction layers after the meta-meta-model (M3) layer. Because, further abstraction levels can be defined by the last, M3 layer. Thus, it is considered that there is no need for repeatedly defining further abstraction levels and M3 level conforms to itself.

As long as this thesis is dedicated to the research question of model change representation, more important use cases of these abstraction levels are *models* and *meta-models* in this thesis. In order to recognize the modeling concepts and constructions of modeling languages, this thesis takes advantage of the meta-models of that models conform to. Therefore, models and meta-models representing modeling concepts are mainly the usages of the approach in this thesis. The modeling concepts and notations of any modeling language can be recognized by looking at the meta-models they conform to. Thus, this thesis refers to the meta-models of modeling languages which allows the approach for being applicable to several modeling languages conforming appropriate meta-models.

2.2 Domain Specific Languages

Whether it is a modeling language or programming language, it has to be expressive, useful in a technical manner, and well-defined encompassing sufficient, yet consist of minimal information about a problem domain. Language definitions

are the trade-offs between precisions and expressiveness which are usually gained through *Domain Specific Languages (DSL)* [van Deursen et al., 1998].

Definition 2.4. **Domain-Specific Language.**
According to [van Deursen et al., 1998], *"a domain-specific language (DSL) is a programming language or specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."*

DSLs are the languages designed for specific domains so that they serve to express solutions for these domains. Mostly, domain experts can read, understand and validate the code of DSLs. The syntax of DSLs is well-designed for, expressed focusing on, and usually restricted to, a particular problem domain [Van Deursen et al., 2000]. In contrast to general purpose languages like Java, DSL can be focused on a particular problem domain and designed with a more narrowed scope.

DSLs allow for expressing domain information in idioms and at the level of abstraction of a concrete domain, thereupon domain experts can understand, validate, modify, and often even develop the DSL programs. The DSL program is self-documented to a large-extend and can be reused for different purposes supplying readability of the syntax and hiding more information. Moreover, DSLs embody knowledge and information within a specific domain and provides reuse of this knowledge [Ghosh, 2010].

Development of DSL typically involves the following steps [Van Deursen et al., 2000]:

- *Analysis:* (1) identify the problem domain, (2) gather all relevant knowledge about the domain, (3) cluster gathered knowledge in a handful of semantic notions and operations on them, (4) design a DSL that concisely describes applications in the domain;
- *Implementation:* (5) construct a library that implements the semantic notions, (6) design a compiler that translates DSL programs into a sequence of library calls;
- *Usage:* (7) write DSL programs for all desired applications and compile them.

Mostly, information domains use domain models to carry out information which belongs to that domain. Thus, DSL usually takes advantage of these domain models in order to obtain information and knowledge about concrete domains.

The DSLs, especially textual DSLs, may offer the several valuable properties being minimal, complete and expressive. Thus, this thesis takes these coherence into account and utilizes DSL for representing model changes (discussed in Chapter 5). This thesis expects the following advantages of using DSL in model difference representations:

– It allows to enclose and hide as much information as needed using DSL notations. This property of DSL makes representations complete. In turns, all change information can be enclosed behind the DSL syntax.
– It may form the directly executable and textual descriptions of model changes which also provides the efficient storage of model changes.
– It is expressive, extensible in a technical manner, and well-defined encompassing sufficient, yet minimal information about model changes.

This thesis takes advantage of DSL for representing model changes in Chapter 5 and developing several collaborative modeling applications on the top in Part IV.

2.3 Model Transformations

As already described in Section 2.1, software models are used to conceptually define and represent behavioral and structural information related to a particular domain including real-world objects, processes, work-flows, etc. In order to achieve the highest level of the effectiveness in dealing with the model-driven software projects, a novel means is required to transform models between modeling languages, manipulate models within the same modeling languages (e.g., *in-place transformations*), or generate source code from software models. In the MDSE paradigm, these features are provided by *model transformations*. Model transformations are the set of *transformation rules* specified by *transformation definitions*. The transformation rules are then executed by *transformation tools*.

Definition 2.5. **Model Transformation.**
[Kleppe et al., 2003, pp. 23ff] defines model transformations, transformation rules, transformation definitions, and transformation tools in the following way:

– *Model transformations* refer to the transformation of one model (source) into another (target) in general.
– *Transformation rules* define associations/mappings between the elements of source and target models, i.e., how the elements of a source model affect the elements of a target model.
– *Transformation definitions* are the aggregations of transformation rules to fully specify the transformation activities of models conforming to a source meta-model into models conforming to a target meta-model.
– *Transformation tools* receive a (source) model and relevant transformation definitions, then produces a new (target) model by performing the transformation process.

As depicted in Figure 2.2, the source and target models conform to their corresponding meta-models and transformation conforms to its definition. By looking

at the meta-models of models, the transformation definitions detect which elements of the source language have to be transferred to which elements of the target language. The model elements which have to be involved in transformation are specified in the transformation rules.

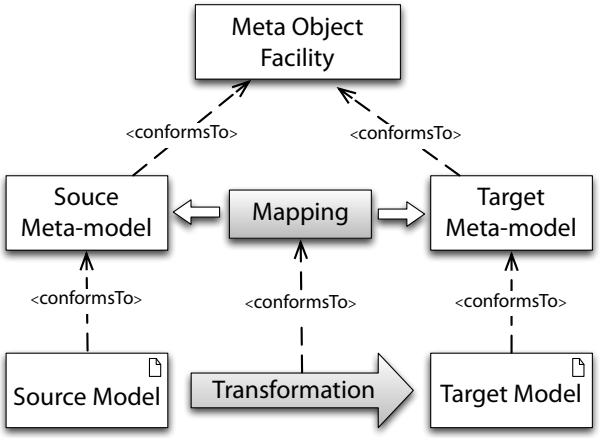

FIGURE 2.2: Architecture of Model Transformation [Jouault et al., 2008]

The transformation might occur between different modeling languages or within the same modeling language. If both, source and target, models conform to the same meta-model, it is called *endogenous* transformation, whereas transformations occur within one modeling language. If the source and target models conform to varying meta-models, it is referred to as *exogenous* transformations, whereas it occurs between different modeling languages. This thesis takes advantage of the endogenous transformations in the realization of its DSL.

"In-place" Transformations. This is the special form of model transformations and is investigated in order to realize the DSL operations in this thesis. It does not require to have source and target models, but it operates on only one model instead. Eventually, it does what the ordinary model transformation does, i.e., manipulate models based on the described set of rules or operations and the resulting model is the same model but with modified modeling artifacts.

Model Manipulations. Model manipulations [Dahm and Widmann, 1998] are the special case of the general model transformations and one of the main foundations in MDSE. Model transformations are automatic generation of `target models` from `source models`. A source model is an existing model which is an input model to the transformation engine and supposed to be transformed into another model which is a target model, i.e., a new model after transformations are executed. The executable descriptions of model transformations permit to propagate the changes defined in transformation rules on a source model and results eventually in a new model version. The most transformation approaches usually provide APIs (Application Programming Interfaces) for performing "in-place" transformations (e.g., `JGRaLab`, `EMF` discussed in Section 2.4). This thesis takes advantage of the basic concepts and tooling support behind the "in-place" transformation approaches in realization of its services (Section 6) and applications (Part IV).

Model to Code (Text) Transformations. Model to text transformations focus on the generation of textual representations from input models. This kind of transformations usually employ the template-based approaches where the expected output text is parameterized with model elements in the input model. Thus, model-to-text transformations can also be used to generate DSL from a meta-model provided. In Chapter 5, this thesis utilizes the *model-to-code* transformations for generating its DSLs (for describing model changes) from the meta-models of the appropriate modeling languages. In DSL generation, this thesis imports the meta-models of modeling languages to obtain transformation information such as language references, transformation parameters (e.g., variables), the set of the named elements of models, language conditions.

2.4 Technical Spaces

So far, this chapter has given a brief insight into the general concepts and principles of abstraction levels of MDSE (Section 2.1), Domain Specific Languages – DSL (Section 2.2) and Model Transformations (Section 2.3). These are the core concepts that provide the necessary knowledge to comprehend the basic concepts and foundations in this thesis. In order to realize and validate the research ideas, the concrete technical environment including tool support is required to define abstraction levels and model transformations. [Kurtev et al., 2002] has introduced the concept of *technical spaces* for generalizing associated technical frameworks into a single technical space (or environment) and compare it with other similar technical spaces.

Definition 2.6. **Technical Space.**
"A technical space is a working context with a set of associated concepts, the body of knowledge, tools, required skills, and possibilities" [Kurtev et al., 2002].
"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].
Technical spaces are also characterized by exhibiting a three-level arrangement of meta-metamodel, meta-models, and models [Bézivin, 2006].

This thesis defines the notion of technical space by the combination of the definitions in Definition 2.6.

Definition 2.7. **Adopted Definition of Technical Space.**
This thesis defines a technical space as a technological environment to realize its theoretical research ideas. A technical space provides a set of associated realization concepts, technologies and tools for *meta-modeling*, *modeling*, *DSL engineering*, *model manipulations* and many more.

There are diverse techniques and tools to be chosen from and combine into a single technical space which is required for realization and validation of this thesis. For instance, the `EMF` and `JGraLab` technical spaces can be referred to as the precise technical spaces of the MDSE paradigm. This thesis takes advantage of the `JGraLab` and `EMF` technical spaces for realizing its research ideas and prototypes. Section 2.4.1 discusses the `JGraLab` technical space and Section 2.4.2 reviews the main concepts of the `EMF` technical space. Section 2.4.3 investigates the further concepts and underlying technologies that are utilized in realizing the research idea behind this thesis.

2.4.1 JGraLab – Java Graph Laboratory

The `JGraLab` (Java Graph Laboratory) [Dahm and Widmann, 1998] technical space is based on strong theoretical foundations and continuous research. TGraphs [Ebert, 1987, Ebert et al., 2008] are used as data structure to represent software models internally. The TGraph schema is used to define meta-models and the TGraph meta-schema [Ebert and Franzke, 1995] defines meta-metamodels. These three technologies establish the M1, M2 and M3 layers of this technical space. In order to define a M3 meta-metamodel layer, grUML (graph UML), a profiled version of UML class diagrams can be used. This technical space can also be used to bridge between spaces [Bézivin, 2006]. The grUML modeling facilitates bidirectional navigations. The JGaLab technical space provides a query language, GReQL (Graph Repository Query Language) [Kullbach et al., 1998], a model transformation language GReTL (Graph Repository Transformation Language) [Ebert and Horn, 2014] and FunnyQT [Horn, 2013] that can also be bridged with the EMF technical space. `JGraLab` further provides an API and code generation facilities. It has adapters to import/export models from/to the EMF technical space (Section 2.4.2).

The JGraLab environment provides generic features for defining meta-models, representing models, defining "in-place" model manipulations and many more. It is independent of certain modeling languages, model designing tools and model manipulation approaches. Considering these features provided by JGraLab, this thesis takes advantage of the `JGraLab` technical space as the technical support for realizing its research prototypes. This section discusses the features provided by the JGraLab technical space.

Meta-Modeling.
This thesis uses TGraph schemas to define its meta-models in the `JGraLab` technical space. A schema can be given in the modeling language grUML or designed in Rational Software Architect (RSA) [Leroux et al., 2006] and imported into the `JGraLab` environment. It is a tool-ready subset of CMOF (Complete Meta-Object Facility) [MOF, 2003]. A grUML diagram (or grUML schema) defines a graph class which specifies the structure and certain constraints for the set of graphs that are instances of this graph class. A schema can be imported into the `JGraLab` environment from a XMI file exported from RSA, by the JGraLab API or by

writing directly into a file using graph formats. The only exception is that every schema needs exactly one class with the stereotype *«graphclass»*. The name of a model becomes the qualified name of the resulting schema after importing it into JGraLab. The JGraLab environment can also handle meta-models defined by EMF Ecore meta-models [Steinberg et al., 2008].

Graph UML (grUML) is the profiled version of UML2 class diagrams. The grUML diagrams inherit most of the notation elements of UML2 class diagrams with some additional stereotypes. The classes of a grUML diagram define vertex classes in the grUML schema. The attributes of a class define the attributes of the vertex class. The associations and association classes of a grUML diagram define edge classes in the grUML schema. The specialized aggregations and compositions define aggregation classes and composition classes accordingly. A TGraph schema (meta-model) defines which types and attributes are allowed in the graph of a certain graph class. The meta-models in Section 7.2 are designed using RSA modeling tool and imported into the JGraLab environment.

Modeling (Data Structures).
The graph-like representations of software models [Ebert et al., 2008] is the most popular form of internal model representations (i.e., data structures for models). All design concepts, rules, and theories, as well as the graphical notations of modeling languages, can easily be represented by graph structures. Software models designed using several modeling tools can also be converted to and represented by graph structures. This thesis partially uses the generic graph-based technique TGraph [Ebert, 1987] [Ebert et al., 2008] which is the specific sub-class of UML class diagram for representing models. In TGraphs, all modeling artifacts are defined by `vertexes` and vertexes are connected using `edges`. Graphs conform to their grUML schema, whereas vertexes are the instances of `vertex classes` in the schema and edges are the instance of `edge classes` in the schema.

The TGraph approach is fully capable of representing software models in a wide range of modeling languages. It is the generic class of graphs which can be informally described by their principal properties. The TGraph schema defines the set of possible vertex and edge classes, and it associates the set of attributes and their domains to each graph element class. TGraph has the following principal properties:

- The vertexes and edges are identifiable first-class, independently.
- All graph elements including graph itself, its vertexes and edges can be attributed and typed.
- TGraph edges are directed and each edge is navigable in both directions.
- The type system provides multiple inheritances for vertex and edge classes.
- The set of vertexes and edges in TGraph are ordered.
- Incident edges are ordered for each vertex.

Figure 2.3 depicts the small excerpt of a TGraph conforming to the meta-model depicted in Figure 7.2. The TGraph describes a UML class diagram and consists

of vertex *v1: KClass* containing vertex *v8: KAttribute* (connected by edge *e6: ContainsAttribute*) and vertex *v4: KAssociation* containing vertexes *v5: KAssociationEndPoint* (connected by edge *e3: ContainsSource*) and *v7: KAssociationEndPoint* (connected by edge *e5: ContainsSource*).

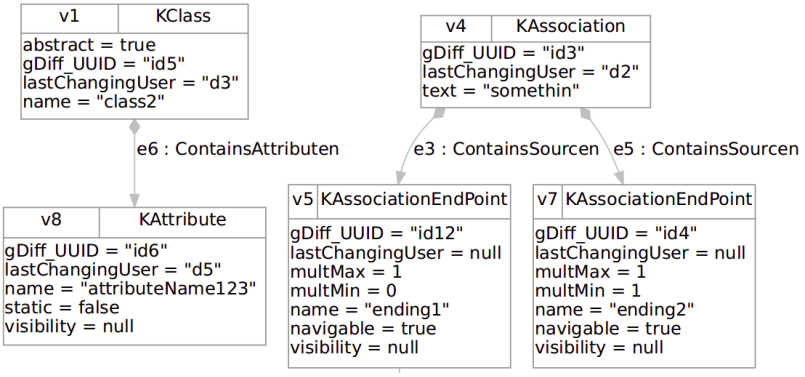

FIGURE 2.3: Example TGraph

The TGraph-based model representation is utilized in implementing supplementary services provided by this thesis as discussed in Chapter 6. Moreover, the collaborative modeling application (discussed in Chapter 7) is partially developed using the same model representation approach.

Model Manipulations.
As mentioned above, the `JGraLab` technical space provides a schema-based meta-modeling feature for defining the modeling concepts of modeling languages. In addition to the meta-modeling and internal data structures, `JGraLab` provides API (Application Programming Interface) for managing and manipulating the TGraph-based models. Hence, meta-models and the TGraph-based models can be created, saved, loaded, as well as the modeling artifacts of meta-models and models can be manipulated and accessed by the JGraLab API. For instance, new artifacts can be created in TGraphs, existing ones can be deleted and the attribute values of existing artifacts can be changed, etc.

In Section 6.4, the applier service provided by this thesis is realized using the model manipulation features of JGaLab API. The applier service is then employed in developing the collaborative modeling applications in Part IV).

2.4.2 EMF – Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [Steinberg et al., 2008] is another specific technical space and ECore is its meta-modeling technique. There are several EMF

modeling tools and a lot of research is dedicated to EMF. The EMF technical space is based on ECore and Eclipse platform for meta-modeling and model-driven software development. The Graphical Editing Framework (GEF) [Rubel et al., 2011], Graphical Modeling Framework (GMF) [GMF, 2018] and Sirius [Viyović et al., 2014] can be used to create the EMF-hosted domain-specific graphical modeling tools. Sirius is widely used and very popular among the Eclipse community and tool developers for developing domain-specific modeling tools.

This thesis takes advantage of the GEF features in creating graphical editor in Section 7.2. It further applies its collaborative modeling approach to the existing Sirius-based domain-specific tool UML Designer in Section 7.3. This section discusses the main concepts of the `EMF` technical space.

Meta-Modeling.
EMF is a modeling framework which is built based on *ECore* and the Eclipse platform. In the `EMF` technical space, *ECore* is used for meta-modeling, and for developing model-driven applications and tools. The *ECore* meta-modeling approach can be used to define custom meta-models if needed. Alternatively, due to ease of meta-modeling effort, UML2 plugins can be installed if only UML profiles are required. The UML2 plugins consist of standard UML profiled notation provided by MOF (Meta Object Facilities) standards [MOF, 2003], i.e., UML meta-models. The resulting Ecore meta-models can be the basis for EMF projects. EMF facilitates code generation (i.e., Java code) that provides APIs to work with the models conforming to the given meta-models.

Modeling (Data Structures).
The EMF-based modeling applications and tools usually store instance models as a collection of *resources* which is called *Resource Set* [Steinberg et al., 2008]. Each resource consists of language concepts (i.e., abstract syntax) defined by ECore meta-models and custom graphical representations (i.e., concrete syntax) if exists. For instance, in case of UML-based modeling, modeling artifacts are stored in the **.uml* files. The graphical concrete syntax of EMF-based modeling applications can be defined in various ways such as *GEF*, *GMF*, etc.

Graphical Editing Framework (GEF) [Rubel et al., 2011] can be used to display the concrete syntax of models in the EMF-based modeling editors. GEF is a graphical editing solution based on the Eclipse framework for creating modeling editors. It provides a model-view-controller architecture. To visualize the graphical views, GEF and Draw2D [Modica et al., 2009] provide the foundations for building graphical views for EMF and other model types. *Graphical Modeling Framework (GMF)* [GMF, 2018] is a tool which encapsulates GEF and Draw2D for creating a graphical editor. GMF also provides an abstract notation for developing EMF-based graphical modeling tools. The Sirius framework [Viyović et al., 2014] extends the GMF notations for developing domain-specific modeling tools, e.g., UML Designer [Obeo Network, 2017]. Draw2d [Rubel et al., 2011] is an object-oriented drawing framework associated to GEF, which the concrete representation takes place. In Draw2d, each diagram has tree-like structures which consist of shapes (figures) and their children as well as edges between the shapes.

UML Designer [Obeo Network, 2017] is one of the popular EMF-hosted domain-specific and open-source modeling tools developed based on the Sirius framework. It provides a graphical modeling editors for designing the most UML diagrams. A collaborative modeling infrastructure introduced in this thesis is also applied to UML Designer in Section 7.3 for designing UML activity diagrams.

Model Manipulations.
In Sirius-based modeling tools (e.g., UML Designer), each modeling project belongs to exactly one session, and each session has its own editing domain and resource set. The resource set contains semantic resources (semantic models), representation resources (representation data such as layout information) and the viewpoint specification models.

A class which extends the *Command Stack Listener* can identify new commands and can get details about them. These commands then allows for performing various actions (e.g., deleting a modeling artifact, creating a new ones, etc). These commands are then collected and processed in *Command Stack*. If a change is made to a *Command Stack*, the *command Stack Changed* method is called in the corresponding *Commander Stack Listener*. The interface of the *Command Stack Listener* can thus be implemented for specific cases of changes.

A class that extends the *Resource Set* listener or the *Resource Set Listener Implementation* class is notifies about changes to the resource set and can thus receive information about the updated artifacts. A class that overwrites the *resource Set Changed* method can respond to these changes. To apply changes to models, a *Recording Command* for the *Transactional Editing Domain* of the session can be created and executed by *Command Stack*. With this command, the elements of a model can be changed. The use of *Transactional Editing Domain* allows for performing *redo/undo* operations (on modeling editors) without any further implementation effort.

In Chapter 6, several services provided by this thesis are also realized using the `EMF` technical space. These services are then utilized to extend collaborative modeling infrastructure for the domain-specific, open-source, EMF and Sirius-based modeling tool UML Designer in Section 7.3

2.4.3 Further Technologies

Several additional underlying implementation technologies are utilized for developing the collaborative modeling infrastructure. These technologies are briefly discussed in this section.

Communication/Synchronization.
In order to provide communication (synchronization) among collaborators in collaborative modeling, various frameworks are studied to build this communication network. Two most popular communication frameworks namely KryoNet [Esoteric Software, 2018] and Netty [Maurer and Wolfthal, 2016] are inspected.

Netty is a network application framework for developing maintainable, high performance protocol servers and clients. It is characterized by being asynchronous and event-driven. When an event occurs, such as if a change occurs, a message can be sent directly via the network. This makes working with TCP and UDP easier. In addition, Netty is a non-blocking framework, which eliminates the need to wait for a different dispatch process to complete when sending messages.

In order to transfer data (model changes, user logging, etc.) between server and client, a fast and efficient graph serialization framework for Java, *KryoNet* [Esoteric Software, 2018] is used that makes the network usable from Java applications. KryoNet offers the possibility to send Java objects from compiled classes via TCP or UDP connections. The serialization of objects is handled automatically but can be exchanged completely or even for individual classes, if necessary. KryoNet API consists of several methods for establishing and interrupting connections as well as exchanging object serializations. It further provides the capability to use Remote Method Invocation (RMI). The goals of the KryoNet project are its speed, efficiency, and an easy technique to use its API. Thus, it is utilized in developing collaborative modeling applications (Chapter 7). During these development, KryoNet has provided sufficient performance and conveniences. Although, there is no other particular reason to choose KryoNet over Netty.

SWT and JFace.
SWT (Standard Widget Toolkit) and JFace are the part of the standard GUI framework used in Eclipse products [McAffer et al., 2010]. SWT provides standard control widgets such as buttons, input fields, labels, etc. SWT uses the native elements of the operating system with its own "look-and-feel" principle, i.e., the controls have the usual layout used by Eclipse instead of the operating system-specific appearance. The Eclipse Plugin API works with SWT and provides (top) classes to create views, editors, and menu items. In Eclipse, views are windows that visualize information, e.g., the Project Explorer or the Console window.

The GUI widgets (including tree views, diagram editor, users list, log messages window, dialog) in collaborative modeling application (in Chapter 7) are implemented using GEF which is based on SWT. SWT is also used for the rest of the GUI for developing user interaction widgets.

These underlying implementation technologies are investigated for only realization, application and validation purposes of this thesis as proof of the concept. However, the theoretical foundations of this thesis does not rely on any underlying implementation techniques and technologies.

2.5 Summary

This chapter has briefly surveyed the MDSE concepts, DSL, Model transformations, as well as potential technical spaces that are involved in realization, implementation, and validation of this thesis. Section 2.1 has clarified the main

principles, the core concepts and abstraction levels of MSDE which helps to comprehend clear distinction between the source code-driven and model-driven software development paradigms. Section 2.2 has briefly reviewed some principles and contributions of DSLs that this thesis can benefit from. Section 2.3 has discussed some core concepts of model transformations that can be used to realize representation descriptions in this thesis.

Finally, Section 2.4 has discussed some potential technical spaces. These technical spaces are inevitably required for the sake of development of the research prototypes as proof of the concept. The JGraLab [Dahm and Widmann, 1998] and EMF technical spaces are chosen as the main underlying implementation environments. Because they support the most essential and popular implementation technologies that are necessary for realization. Moreover, JGraLab does not rely on any specific modeling language, internal model representation or model transformation techniques and it provides the more technical environment which is capable of handling the wide range of modeling languages.

In order to achieve efficient results in model difference representation, this thesis takes advantage of the underlying syntactical principles of domain-specific languages (DSL). The utilization of DSL in model difference representation brings several advantages; (1) as much information as needed information can be enclosed using DSL, (2) the compact syntax of DSL provides efficient representation of model differences in collaborative modeling, (3) DSL can easily be designed using expressive syntax, as well as can be realized with less implementation effort.

The reviewed potential technical spaces and underlying implementation technologies are investigated for only realization, application and validation purposes of this thesis as the proof of the concept. However, the theoretical foundations of this thesis does not rely on any technical spaces or implementation technologies.

Chapter 3

Collaborative Development Use Cases

Software production activities usually undergo two sub-processes such as the *initial development* and *maintenance* (i.e., evolution). The initial development phase may also be considered as the special part of overall software evolution, whereas only more features are added to software systems or existing features are optimized. It is the development phase which takes place till the initial deployment of software projects.

All development and maintenance activities contribute to the evolution of software systems. After initial deployment, software systems are further subjected to constant changes. Software systems are changed because of various reasons such as extensions, corrections, optimizations, adaptations and other improvements. All these changes are usually entitled as *software maintenance* [Lientz et al., 1978], [Chikofsky and Cross, 1990]. Lientz and Swanson [Lientz and Swanson, 1980] classify software maintenance into four basic classes.

- *Adaptive maintenance* considers changing software systems to cope with changes in the software environment, e.g., adapt software systems to new hardware environments or operation system environments.
- *Perfective maintenance* covers all change activities implementing new or changed user requirements which are concerned with the non-functional enhancements of software systems, e.g., adding new features or structural changes.
- *Corrective maintenance* is usually dedicated to detecting and fixing errors in software systems, usually detected by end users, e.g., detecting and fixing bugs, correcting errors.
- *Extensive maintenance* is dedicated to changes only in functional requirements, e.g., adding new functionality or behavioral changes like optimizing implementation algorithms.

Development and maintenance is needed to ensure that the software systems still satisfy user requirements. Software systems are changed due to correction of faults, improvement of design, implementation of enhancements, interface with other software systems, adaptation of programs. So that different hardware, software, system features, and telecommunication facilities can be used, migrate legacy software and retire software [Bourque et al., 2014, p. 106].

As software systems become huge and complex with the several thousands of software artifacts, development and maintenance of the large-scale, evolving software systems require collaborative work of several stakeholders, project managers, developers, designers, testers and others (i.e., collaborators) on the shared software projects.

Definition 3.1. **Collaborative Development.**

Ellis et. al. [Clarence et al., 1991] defines *collaborative development* as computer-based collaborative development that supports the groups of people engaged in a common task (or goal) and that provides an interface to a shared interface.

According to [John, 2010], collaboration involves two or more people editing the same document at the same time.

Borestein [Borenstein, 1992] and Schooler [Schooler, 1996] broadly defines the field of collaborative work, encompasses the use of computers to support coordination and cooperation of two or more people who attempt to perform a task or solve a problem together.

As software systems evolve over time undergoing changes, constantly changing software systems during development and maintenance results in the multiple revisions of the same software artifacts. Collaborative software development allows several people for editing software artifacts using different computers, a practice called collaborative editing. There are two scenarios of collaborative development that are considered in this thesis. They are distinguished by *when* and *where* the interaction takes place [Clarence et al., 1991]. In this context, two primary dimensions are identified in Figure 3.1. The figure classifies collaboration scenarios in two dimensions: *Time* and *Space/Place*. The *Time* dimension is divided into two phases namely the *Same Time* and *Different Times*. In this thesis, collaboration which occurs in the *same time* is referred to as *concurrent collaboration*, and collaboration which occurs in the *different times* is referred to as *sequential collaboration* regardless the place they are located.

The *time/space* taxonomy consists of the four main forms of interaction. They are discussed below, in detail.

Face-to-Face Interaction. If the collaborators are located at the same place at the same time, it is called as *face-to-face collaboration*. In this kind of interaction, neither electronic devices nor communication networks are involved. Thus, this form of interaction is out of the scope in this thesis.

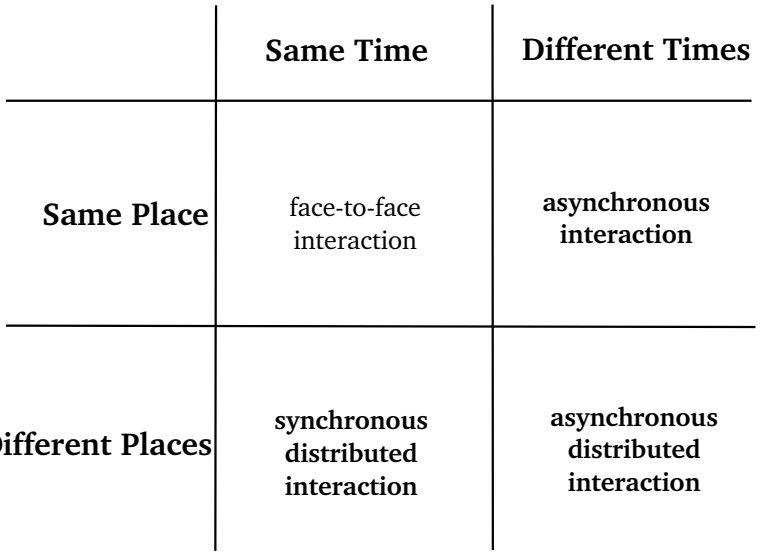

FIGURE 3.1: The Time/Space Taxonomy [Clarence et al., 1991]

Synchronous Distributed Interaction. This type of interaction occurs when collaborators are located in different places and communicate at the same time. This kind of collaboration is entitled as *concurrent collaboration/versioning* which happens in *real-time*. Synchronous interaction architectures are already realized in software development and evolution activities, as well as textual document editing (e.g., Google Docs [Google Inc., 2017], Etherpad [AppJet Inc., 2017]).

Asynchronous Interaction. If the collaborators are located in the same place but at different times, it is entitled to be *asynchronous interaction*. This form of interaction is the special scenario of *asynchronous distributed interaction* which is discussed, below.

Asynchronous Distributed Interaction. This interaction involves collaborators in different times and located in different places. There are several implementations of asynchronous distributed interaction principles and already applied to software development and evolution activities. Some of such systems like Subversion [Collins-Sussman et al., 2004], Git [Swicegood, 2008], CVS [Baudivs, 2014], and many more are the examples for this class of interaction tools. These approaches are usually referred to as *version control systems*. This form of collaboration is consider as the *sequential collaboration/versioning* scenario of collaborative development in this thesis.

As this thesis focuses on the concurrent and sequential scenarios of collaborative development, Figure 3.2 depicts the general architecture combining these two scenarios in a single architecture. This architecture demonstrates *main software development line* as a central software project that is being developed and evolved, thus, under collaboration. This central software project is being developed by several collaborators (*modify*). All activities (except *modify*) in the figure have two parts that are separated by *forward slash*.

Concurrent Collaboration. The vertical line in Figure 3.2 indicates that the collaborative work takes place on the shared software system in parallel, i.e., synchronous distributed interaction. Above the forward slashes, the figure depicts

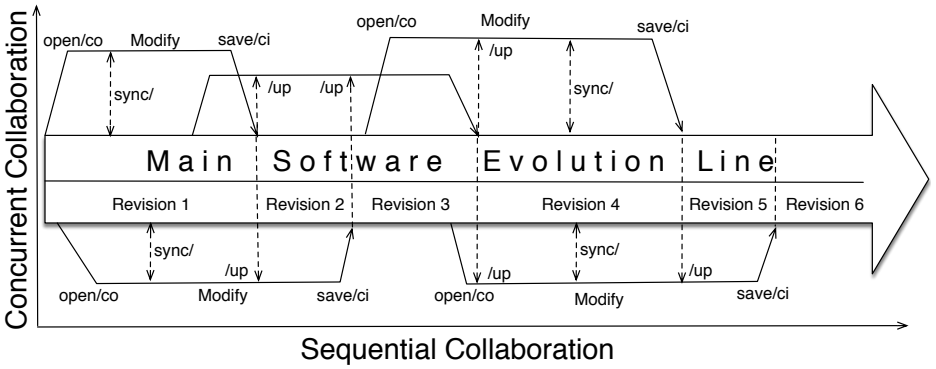

FIGURE 3.2: Combined Architecture of Concurrent and Sequential Collaboration

several activities of concurrent collaborative development. For instance, collaborators join the collaboration process by *opening* the central software project and they continue *modifying* it in their environment. While modifying the opened copies of the project, their changes/modifications are constantly *synchronized* with the *main software evolution line*, so that their changes (i.e., *concurrent revisions*) are constantly synchronized with other parallel collaborations and vise verse. After completing their work for that time period, they may save their software project which results in a new revision (i.e., *sequential revisions*) in the central development line. Next days or weeks, collaborators may continue developing their project opening the latest saved revision. As long as they usually save the correct and complete revisions of their project, they also can revert these revisions in cases that their project face failure, error or loss. This overall scenario is denoted as *concurrent collaboration* in the remaining part of this thesis.

Sequential Collaboration. The other scenario depicted in Figure 3.2 is *sequential collaboration* where each collaborator checks out (*co*, below the forward slashes) a copy of the central project, *modifies* it in their distributed environment, and checks in (*ci*) their changes into the central development line. But, before they check in their changes, they still have to update (*up*) their local copies with the central project. Because other collaborators might already have checked in other changes into the central project. Thus they have to first fetch changes from the central project in order to keep their copies up-to-date. Constantly changing and checking in the shared projects results in the several subsequent revisions of the same artifact differing from each other. As discussed above, collaborators are involved in the sequential collaboration in different times from different places, i.e., *asynchronous distributed interaction*. This use case is referred to as *sequential collaboration* in the remaining part of this thesis.

There are several concurrent and sequential collaboration tools for the source code-driven software systems and textual documents. The approaches, dedicated to source code-driven projects and textual document writing, can not efficiently be used in model-driven software projects because of the paradigm shift between the code-driven and model-driven concepts. Similar to source code-driven software projects and textual documents, MDSE is the subject to continuous development and maintenance. Thus, the similar collaborative development support (i.e., *collaborative modeling*) with the concurrent and sequential scenarios is inevitably

required for collaborative MDSE. Although the MDSE concepts are not the same as the concepts of source code-driven software projects and textual documents, the underlying architectures and terminologies of source code-driven collaborative development and textual editing approaches can still be taken over and reused in collaborative MDSE.

The most collaborative software development and text editing approaches provide some kind of analysis features in order to compare different revisions or analyzing the histories of software systems and textual documents under development. These comparison and analysis features are usually provided by tracing and visualizing the histories of changes in the software systems and textual documents. Similarly, the history of evolving software models is interesting for collaborators in order to track and analyze the evolutionary life-cycle of software models. By providing history analysis support for MDSE, collaborators can easily analyze and manage the development process and make further decisions about the evolution process of their software models. To this end, advanced history analysis support for MDSE is needed to *analyze the histories of software models* under collaborative development.

In the follow-up subsections, this chapter gives the brief descriptions for concurrent and sequential collaboration as well as the history analysis use cases of this thesis. These use cases are discussed distinguishing between source code-driven/textual and MDSE paradigms. Section 3.1 reviews concurrent collaboration principles and techniques for source code-driven/textual and MDSE. In Section 3.2, this chapter investigates the core concepts behind the sequential collaboration scenario for source code-driven and MDSE. Section 3.3 inspects the software history analysis use case for source code-driven/textual and MDSE paradigm.

These sections further motivate and derive required support for concurrent and sequential collaboration, as well as the history analysis use cases for model-driven software development and evolution. Additionally, these sections specify expected benefits from the difference representation for collaborative MDSE.

3.1 Concurrent Collaboration

As technology develops, there has already been a growing need for the computer supported cooperative work. Diverse approaches and technologies have been proposed to provide computer supported collaborative artifact editing in real-time [Rama and Bishop, 2006], i.e., *concurrent collaboration*. The early ideas of concurrent collaboration was demonstrated by Douglas Engelbart in his "mother of all demos" presentation [Engelbart, 1968] in 1968 where he demonstrated remote collaboration on a shared screen among other foundational computer science aspects. Formerly, the idea of concurrent collaboration was also entitled as the *Instant Update* [Gupta, 2000], *computer supported cooperative work* [Yang et al., 2000] and *groupware* [Clarence et al., 1991] in some literature.

Definition 3.2. **Concurrent Collaboration.**
According to [John, 2010], concurrent collaboration systems are software systems including hardware technologies that allow multiple users to cooperatively work on projects at the same time.

So far, there exist different concurrent collaboration approaches having their own sets of services for accommodating interactions among users, involving instant messaging and file sharing. Some of these approaches offer concurrent collaboration, where files can be amended or altered jointly in real-time. Concurrent collaboration systems can be facilitated by shared access to a centralized server and jointly, constantly editing the shared software artifacts, textual documents, or other form of information [Rama and Bishop, 2006].

Like any other form of interaction, the development and evolution of software systems require collaborative development of several team members enabled by concurrent collaboration. In concurrent collaboration, collaborators apply various changes to the shared software projects in parallel. Since several team members collaboratively work on the shared artifacts, the collaborative development approaches need to provide sharing software artifacts and the synchronization of the changes (i.e., concurrent revisions) among collaborators when new artifacts are created or existing ones are deleted or changed, i.e., instant updates. Instantly editing the parallel copies of the shared software artifacts results in the several parallel revisions of the same artifact differentiated by *concurrent revisions*. These concurrent revisions are then instantly identified and synchronized between collaborators in real-time. With this scenario, the concurrent revisions of the huge and complex software systems and documents are easily and constantly shared among several collaborators.

This section further investigates a brief list of the concurrent collaboration approaches for textual documents and source code-driven software projects.

3.1.1 Concurrent Text-Driven Collaboration

This section discusses a group of concurrent collaboration principles that are already investigated in textual document editing, and in source code-driven software development. These two groups of concurrent collaboration systems are investigated in order to comprehend their basics, study common underlying principles and technologies that might contribute to develop concurrent collaboration systems for MDSE.

In text-based concurrent collaboration, the two or more authors of shared documents can jointly create, update and revise textual documents. Through collaborative writing systems, identification of modifications are provided, i.e, the authors are able to keep the track of the evolving documents and identify who changed the document and to what extent [Bafoutsou and Mentzas, 2002]. For instance,

Google Docs [Google Inc., 2017], Etherpad [AppJet Inc., 2017], Firepad [Firebase Inc., 2017] and many more are widely used in document creation and editing in real-time.

The concurrent collaboration systems are usually dedicated to creating, modifying and maintaining huge, shared and centralized documents. The collaborators of shared artifacts can access the centralized and shared software project either having or without having the copies or branches of artifacts on their local machines and directly modify the software project, document or model on the server side. Once each collaborator modifies artifacts, the changes are automatically interpreted on the central project and are synchronized with the other parallel copies of that project.

Collaboration Architectures. The most existing concurrent collaborative systems for textual documents are built using one of two different types of following architectures:

- *Centralized.* In the centralized architecture [Nichols et al., 1995], the shared documents are located in a shared server or machine and collaborators are authorized to access these documents. Only one centralized copy of the shared documents is available in this type of architecture. For example, Google Docs [Google Inc., 2017], Etherpad [AppJet Inc., 2017], Firepad [Firebase Inc., 2017] are built on the top of centralized architectures and they usually use browser-based document editing environment.
- *Decentralized.* The decentralized architecture [Buchegger et al., 2009] is also referred to as the *peer-to-peer* architecture [Ahmed-Nacer et al., 2011]. Like the centralized architecture, this architecture uses a centralized server for synchronization purposes, but it further provides standalone client side tools which store the copies of the centralized, shared documents and changes are continually synchronized among these copies. For instance, Microsoft SharePoint Workspace [English et al., 2010] (formerly known as Groove) and TeamEdit [TeamEdit, 2011] are built on the top of the centralized architectures and they usually use standalone document editing environments.

Synchronization Algorithms. The synchronization algorithms are classified into the state-based and operation-based groups. These algorithms are used to synchronize the artifact changes through the collaboration architectures. Currently, these two algorithms are widely utilized in concurrent collaborative systems:

- *State-based (Differential Synchronization).* This group of algorithms takes advantage of only different states of the shared documents. The examples for state-based approaches are three-way merges adopted by version control systems such as Subversion [Berlin and Rooney, 2006] and differential synchronization approaches [Fraser, 2009]. Since performance is a key factor in concurrent collaboration, the state-based synchronization technique is considered to be inefficient in case of low-speed network connection as the

differences between states have to be computed every time after the documents are edited resulting in increased time complexity [Ahmed-Nacer et al., 2011].

– *Operation-based (Operational Synchronization).* Operational synchronization technique is supposed to be the suitable approach and widely used by concurrent collaborative systems [Clarence et al., 1991], [Ressel et al., 1996], [Sun and Ellis, 1998], etc. Modifications on the shared documents are presented by means of operations such as insertions and deletions of a sequence of characters. In this case, changes made by collaborators are instantly recorded (or listened for), defined as *concurrent revisions*, and synchronized in real-time.

Besides, there are several algorithms known as CRDT (commutative replicated data types) [Oster et al., 2006], [Preguica et al., 2009], [Roh et al., 2011], [Weiss et al., 2009]. The operational synchronization algorithms are used as the underlying concept for CRDT. Below, the core concepts of some operational synchronization and CRDT algorithms are briefly reviewed:

– *WOOT Algorithm* [Oster et al., 2006]. WOOT uses very basic change operations such as insertion and deletion of elements in a linear structure and these elements are uniquely identified. All kind of change operations are represented by the combination of these two atomic operations.

– *Replicated Growable Array (RGA)* [Roh et al., 2011]. This approach supports not only insertion and deletion but also update operations which replace the content of elements.

– *Logoot Algorithm* [Weiss et al., 2009]. Logoot is another CRDT approach that ensures the consistency of linear structures. Logoot associates to the list of elements of the structure, an ordered list of identifiers. Identifiers are composed by a list of positions. Positions are 3-tuples formed with a digit in specific numeric base, a unique site identifier and a clock value.

The most existing concurrent collaborative systems follow the collaboration architectures and synchronization algorithms as discussed above. These are the core techniques which applied to concurrent collaboration tools (e.g., [Google Inc., 2017], Etherpad [AppJet Inc., 2017], Firepad [Firebase Inc., 2017] and many more). Furthermore, these tools provide some additional tool specific features as follows:

– *Editor GUI.* The most concurrent collaboration tools usually provide an editor feature for instantly updating the textual document in real-time. Modifications are visible in real-time enabling support for undo/redo operations.

– *User Authentication.* This feature is referred to as *collaborative code ownership.* The user needs to be authorized to edit shared documents. Each user might be authenticated with a specific right such as write, read only, update or the combination of these. Eventually, the list of users registered (authorized) in the system is usually displayed on the editors' graphical user interfaces.

– *Sharing.* Collaboration can be organized in different ways. For instance, one document might be distributed among several editors dividing it into several parts, one document might be locked for other users while it is being edited by one user, each user might edit an independent document in parallel and so on.
– *Chatting.* The most concurrent collaboration systems usually provide chat features among the users of a system. The chat feature might be in different forms such as textual (instant messaging), audio (voice conversations), video chatting (screen sharing) features.
– *Highlighting.* This feature supports the highlighting of textual documents and users, i.e., each user is given a particular color so that the changes he/she made are highlighted with his/her color.

These features are only short list of the common services provided by the existing concurrent collaboration systems especially for textual documents. Besides, there are several concurrent collaboration systems for source code-driven software projects [Heinrich, 2013], [Goldman et al., 2011]. Concurrent collaboration systems follow similar architectures, principles, synchronization techniques and provide features as discussed above. They further support some additional features which are specific for the syntax of programming languages. Because, programming languages follow concrete grammar (i.e., syntax) rather than just plain-text [Dillenbourg, 1999]. This class of tools is also known as *web-based and cloud-based IDE (Integrated Development Environment)* [Hausladen et al., 2014], [Eclipse Orion, 2014], [Fluidbyte, 2014].

– *Language-specific syntax highlighting.* In concurrent collaboration, the syntax of written text on the editor is highlighted with different colors according to the syntax definition of a programming language.
– *Single click deployment.* As provided by standard IDEs like Netbeans, Eclipse, or many others, concurrent collaboration tools provide the deployment of the shared software systems by single click. This deployment usually occurs on the server side where the shared software systems reside.
– *Error handling.* Whenever, developers compile, run, or deploy their software systems under collaboration, the aforementioned concurrent collaboration systems support error handling feature whenever they arise. The error handling is usually capable of catching, highlighting and handling error as soon as a piece of source code is written on the editor.
– *Auto-completion.* All modern IDEs provide automatic code completion while writing the source code of software systems. The web-based concurrent collaboration systems also provide such features on the editors.

Discussions in this section indicate that there are several synchronization algorithms and collaboration architectures for textual document editing and source code-driven concurrent collaboration. These approaches exchange the sequences or lines of characters by means of operations such as insertions, removals or updates of the sequences of characters.

These synchronization algorithms can not be directly used in concurrent collaboration scenarios for MDSE and evolution activities because of the paradigm shift between code-driven and model-driven software engineering concepts (as discussed in Chapter 2.1). As long as software models have associated and composite data structures [Kleppe et al., 2003], existing concurrent collaboration technologies for source code and textual document editing can not be directly applied to model-driven concurrent collaboration scenario (explained in Chapter 3.1.2). However, the architectural principles, concepts and terminologies on the higher level may be taken over and reused for developing advanced concurrent collaboration support for MDSE and its evolution. To sum up, there is a strong need for further extensive research in the field of concurrent collaborative MDSE.

3.1.2 Concurrent Model-Driven Collaboration

Like the source code of software systems and textual documents, software models are also subjected to constant changes because of development in novel technologies, increasing user requirements, improvements, and corrections. Development and maintenance of software models require the collaborative development in parallel, i.e., in real-time because of their complexity and large size with several thousands of modeling artifacts. Concurrent collaboration is quite significant for creating and maintaining large-scale and complex software models. Concurrent collaboration might provide on-line designing support for models and instantly exchanging concurrent revisions between collaborators in real-time.

In parallel development of software models, several modelers design the same shared software models. But, each collaborator has his/her own copy of models and individually designs the own part of models. While designing models, changes need to be identified, synchronized and instantly propagated on other parallel copies. In comparison to textual document editing and software source code development, identification and synchronization of instant changes in MDSE is a more challenging task because of the graph-like structures of software models. Thereby, a number of technical challenges may arise in establishing concurrent collaborative modeling approaches.

Technical Challenges. The concurrent collaboration challenge stems from making software models commonly available to multiple users in different locations and communication issues among these users. Especially, the term *real-time* in concurrent collaboration requires extra research efforts and consideration in developing concurrent collaboration systems for MDSE rather than plain text-based documents. Because, changes on shared software models have to be detected and synchronized in real-time enabling collaborators to communicate without delays. However, the speed of communication is limited by network speed latency which creates a fundamental dilemma: collaborators need to be capable of synchronizing their changes with the centralized project instantly, regardless of the network speed and the complexity, large-scale of their shared software models.

There are several additional technical challenges associated with developing concurrent collaboration systems for MDSE and evolution. It has to cover different kinds of modeling artifacts enabling re-usability of different modeling languages, tools, model transformations, etc. Modeling and model management tools are commonly distributed as software packages, often on top of the complex software development IDEs (for example, Eclipse). This can be a burden, particularly for nontechnical experts [Rocco et al., 2015].

Existing Approaches. There are several *industrial/commercial* model-driven concurrent collaboration approaches and tools [Tolvanen, 2016], [Franzago et al., 2018]. Only few of them are open source and explicitly documented. This section excludes the commercial tools and approaches because they are not open source and explicitly documented making them difficult to study and extend. However, there are several web tools and approaches like GenMyModel [GenMyModel, 2015] [Dirix et al., 2013], Creately [Cinergix Pty., 2015], Gliffy [Gliffy, 2017], which exchange changes over WebSockets using web browsers. Their underlying ideas and implementation technologies like handling techniques of modeling concepts, change representations and synchronization are not explicitly documented.

The open source and well-documented approach EMFStore is introduced by Koegel and Helming [Helming and Koegel, 2013]. It provides a collaborative modeling infrastructure hosted in Eclipse for EMF based software models. Another EMF-based collaborative modeling infrastructure CDO (Connected Data Objects) [Stepper, 2018] is well-suited for developing collaborative modeling tools in the EMF technical space. As long as these approaches operate on the tree-like structures of software models, they need graphical editors for designing software models. Furthermore, these approaches store the histories of software models relational databases which might cause a problem as their model repositories become very huge over time. Chapter 4 investigates the aforementioned and other related approaches in detail.

As discussed in Section 3.1.1, there are several *collaborative development architectures* and *synchronization algorithms* for textual and code-driven concurrent collaboration. As result of studying literature, concurrent collaborative modeling approaches, especially web-based approaches, take advantage of the *centralized* collaborative development architectures. However, it is not explicitly documented which synchronization algorithm they utilize for representing and synchronizing changes.

Change Representation in Concurrent Collaboration. According to discussions in this chapter, the essential and widely used scenario of collaborative development is concurrent collaboration of several collaborators in real-time. As concurrent collaboration is dedicated to instantly creating, modifying and maintaining huge, shared and centralized projects (i.e., of textual documents, source code, models), the changes made by collaborators have to be continually detected and synchronized among the several concurrent copies of that projects. Unlike textual documents or source code, increased performance of change synchronization matters in collaborative modeling because of the graph-like composite structures

of software models. Thus, *model changes* have to be identified, represented and synchronized instantly using very compact notations. Typically, concurrent revisions are usually not stored, yet synchronized between the parallel working copies of software models.

Collaborative development approaches for MDSE have to consider the modeling concepts of model-driven software projects. Mens [Mens, 2002] discusses the limitations of textual differentiation and merging: *"[...] It does not take the specific semantics of software artifacts into account because everything is treated as an ordinary piece of text"*. Textual document and source code editing systems operate under a linear sequence of characters; file-based paradigm that is purely textual. Although since models are not sequential text lines, they could be represented either as trees or graphs, these systems cannot use the formal structure of models to achieve correct compare or merge results concerning syntax and semantics [Kofman and Perjons, 2004], [Lin et al., 2004].

3.1.3 Required Support

Discussions on concurrent text/code-driven collaboration approaches show that there are several common concurrent collaboration and synchronization techniques for textual documents and source code-driven projects, but very few concurrent collaboration approaches exist for MDSE. However, research on concurrent collaborative modeling can slightly take advantage of some general concepts of concurrent text-driven collaboration. For instance, the centralized architecture and operational synchronization algorithms are perfectly suitable and the very effective in case of concurrent collaboration for MDSE. Though, operational representation of concurrent revisions is quite different than the operational representation of textual documents and software source code. Operations for model changes embody more information about differences/changes, whereas operations for textual document changes consist of only lines of text or the sequence of characters as motivated in Section 1 and Section 3.1.2.

In concurrent collaboration, all modeling artifacts can be created, removed or their attribute values can be changed during initial development and evolution. All these modifications are referred to as *changes*. Changes forming *concurrent revisions* are the central artifacts in concurrent collaborative modeling. Representation of these changes is the core challenge in developing model-driven concurrent collaborative systems. Starting from the initial creation, each modeling artifact undergoes various aforementioned changes. These changes have to be identified and represented using efficient and suitable techniques.

There is a need for extensive research in a generic change representation approach for concurrent collaborative modeling regardless of modeling languages and modeling tools. It has to satisfy the technical challenges defined in Section 3.1.2. A *generic change representation approach* is needed for concurrent collaborative MDSE that can be used as a common underlying change representation in concurrent collaboration environment for MDSE. It should provide core functionality

and extension mechanisms enabling developers to develop concurrent collaborative modeling systems and infrastructures for different domain-specific modeling languages. Doubtless, the underlying technologies, concepts, architectures and mainly terminologies of concurrent textual/code-driven collaboration approaches can be reused in developing collaborative modeling approaches.

3.1.4 Expected Benefits by Difference Language

As discussed in Section 3.1.3, a generic change representation approach is required for concurrent collaborative modeling. The technical challenges described in Section 3.1.2 and required support defined in Section 3.1.3 can be satisfied by a generic *difference language (DL)*. DL should be generic with respect to the meta-models of modeling languages. Any service, component or plug-in required for concurrent collaborative modeling can then be developed on the top of that language definition enabling generality, tool independence, extendability and re-usability properties.

The concurrent revisions represented by DL enables scalability of larger, more complex model changes. DL supports several following advantages in concurrent collaborative modeling:

1. The model changes represented by DL may completely satisfy *operational synchronization* principles which provide high performance in change synchronization in real-time.
2. The high performance by DL-based change representations may allow for avoiding possible change conflicts in real-time (more evidence about validation results can be found in Chapter 10).
3. According to general language design, DL may serve as a common change representation and exchange format for various modeling languages, modeling tools and among various components and services of concurrent collaborative modeling, sequential collaborative modeling (discussed in Section 3.2) and history analysis (discussed in Section 3.3) use cases.
4. The DL-based changes can easily be detected from and applied to shared models regardless of their large-scale and complexity.
5. The histories, changes on concurrent revisions or between the different states of shared software models can be represented and stored using the same DL.
6. Any revision of shared models or their parts can be traced and browsed for further analysis.
7. It can easily be transferred over the network since it consists of only changed parts of software models.

These benefits are the main contributions of DL in case of concurrent collaborative modeling, particularly. Section 4.3 generalizes these expected benefits in the overall requirements for DL.

3.2 Sequential Collaboration

During initial development and maintenance of software systems, new software artifacts are constantly created, existing ones changed or deleted. For the sake of minimizing conflicts and developing large-scale software projects, they are often strictly separated into different parts and assigned to collaborators. Each collaborator usually feels responsible for his/her part of work. According to the *collective code ownership* manifesto of agile software development, an entire team is responsible for the shared software system and everyone works together to produce software artifacts [Beck et al., 2001]. Software artifacts are produced by the teams of several software developers, designers, project managers, and testers. According to [Beck et al., 2001], collective software artifact ownership allows for sharing knowledge, developing software projects with good quality and better style, enabling independence on other individuals, reviewing code efficiently, and good learning scope.

Constantly changing software systems results in several different revisions of the same software artifact. Thereby, software developers wish to store the different revisions of the same software artifacts including their changes. They intend to manage software revisions so that the previous software system revisions can be reverted or the change histories can be traced when needed.

Software projects usually undergo the initial development and evolution (i.e., maintenance) phases forming their evolution in general. A set of changes transfers software projects into new states resulting in new revisions. Eventually, there are several revisions of the same software projects under evolution. The multi-version software projects are usually managed using *version control systems*, i.e., *sequential collaboration*. These revisions are usually used to identify, store and reuse changed software artifacts [Conradi and Westfechtel, 1998].

Definition 3.3. **Sequential Collaboration.**
According to Glasser [Glasser, 1978], a sequential collaboration system is a system for controlling changes to files of text (typically, the source code or documentation of software systems). It provides facilities for
– storing, updating, and retrieving all revisions of files;
– controlling and updating privileges;
– identifying the revisions of the retrieved files;
– recording who made each change, when and where it was made, and why.

In order to provide the stable evolution of software systems, user changes are usually identified and stored in the software repositories using *difference documents*.

These difference documents are reused in the further manipulations and analysis of the base software system. Changes in difference documents embed adequate knowledge about the history of artifact modifications.

The objective behind sequential collaboration is to preserve the history of software projects. Sequential collaboration systems are used to store and handle the histories of evolving software systems. They offer features such as adding a software project to sequential collaboration, updating the latest changes from the main repository, merging local branches to the main development trunk by committing local changes to the main repository. All these version management features are built on the top of software repositories taking advantage of difference documents for storing software change histories. With the help of sequential collaboration, software product owners and developers can easily develop and maintain their large-scale software projects with the teams of developers.

Sequential collaboration plays an essential role in the most of the development processes. Ranging from simple undo/redo capabilities for software changes to complicated model management, branching and collaboration of several developers. This section focuses on the novel approaches for sequential collaboration. Section 3.2.1 inspects sequential text-driven collaboration approaches. Section 3.2.2 studies the state of the art in sequential collaborative modeling approaches. Learned lessons and required support are explained in Section 3.2.3. Section 3.2.4 portrays expected benefits of suitable difference representation in sequential collaborative modeling.

3.2.1 Sequential Text-Driven Collaboration

There are several source code-driven sequential collaboration approaches and techniques consisting of advanced concepts and foundations. This section explains the current state of the art in sequential text/code-driven collaboration approaches. They are investigated in order to study the core architectures, concepts and terminologies that can be conducted in sequential collaboration for MDSE.

Software Repositories. Software repositories are used to store software projects that are being developed or maintained, their revisions and all other necessary data related to their development and evolution. For storing software projects and their revisions in repositories, one naive approach is to duplicate the entire software project after each modification. Instead, repositories may store the latest revisions of software systems and the differences between their revisions. Because the latest revisions are the most frequently accessed during collaborative development.

There are two forms of software repository architectures [Altmanninger et al., 2009].

– *Centralized.* This is a classical approach to sequential collaboration which has a single central repository storing the project history and clients can have the working copies cached locally, and possibly also (parts of) the history. This

repository paradigm requires connection with the central server to perform any change operations (e.g., committing new revisions). The centralized collaboration approach requires to gain appropriate commit permissions before being able to work. The work and performance done by all developers can easily be tracked [Baudivs, 2014]. For instance, Subversion [Collins-Sussman et al., 2004] and Perforce [Wingerd and Seiwald, 1998] are built based on this kind of repository architecture.

– *Distributed.* In this form, users have their own repositories on their local machines in addition to their working copies (*pulled*). Repositories can be optionally synchronized with other repositories. There is no need to establish the connection to server. Operations such as commit, browsing the history and check out, are fast. Later, these changes can be sent (*pushed*) to the other repositories in order to synchronize changes with other developers' repositories. For example, Monotone [Hoare et al., 2005], Git [Swicegood, 2008], Darcs [Roundy, 2005] and Mercurial [Mackall, 2006] follow this technique.

Repository Storage Models. In source code-driven collaboration, there are two models for storing software repositories:

– *Snapshot-oriented.* In this repository storage model, the first-class objects are usually the particular revisions (*snapshots*) of projects at a particular moment. The traceability links between revisions defining artifact changes are detected and identified by comparing the given two revisions (e.g., Monotone [Hoare et al., 2005], Git [Swicegood, 2008], Mercurial [Mackall, 2006]).

– *Changeset-oriented (Delta-based).* This technique focuses on the only changed artifacts of software projects. The new revisions of projects mean the *differences* between two; old and new revisions of projects (e.g., Darcs [Roundy, 2005], Subversion [Collins-Sussman et al., 2004]). As long as the differences consist of only changed artifacts, this category of approaches can be referred to as *delta-based* approaches.

Delta Representation Approaches. An artifact change defines any kind of modification made to software artifacts. The set of artifact changes can be aggregated to a group which is called *differences* and stored in *difference documents*. Difference documents are referred to as *deltas*. Consequently, deltas consist of the group of changes. The most repository storage models take advantage of deltas to store change information, i.e., differences.

> ***Definition*** 3.4. **Delta.**
> The aggregated group of artifact changes is defined in terms of *differences* and represented in difference documents, referred to as *deltas*.

The existing sequential text-driven collaboration systems use diverse delta representation approaches depending on their implementation techniques. Thus, this section focuses only on the most popular approaches.

- **Delta Algorithms.** This is dedicated to compare two objects and create a *delta*. There are several delta algorithms. Some algorithms provide suitable and sensible output for human review and others focus on finding the smallest set of differences and providing minimal required output for storage [Baudivs, 2014].

  – *Delta Combination* [Hudson, 2002], [Proceedings, 2006]. While applying deltas, a set of deltas is emerged to a large singe delta, then applied. This approach dramatically increases the performance of delta application process. Similar technique entitled *skip-delta* [Hudson, 2002] is used in Subversion [Collins-Sussman et al., 2004].

  – *Myer's Longest Common Subsequence* [Myers, 1986]. Another widely used algorithm is based on recursively finding the longest sequence of common lines in the list of lines of compared objects. This algorithm is also used by *GNU diff* tool [MacKenzie et al., 2003]. This algorithm performs two breadth-first searches searching for modifications (line addition/removals/keeps) on the compared files. The shortest modification sequence is found when these two searches meet. The output of the algorithm is the additions, removals and keeps of textual lines. Another optimized version *Patience Diff* of Myer's LCS algorithm was introduced by Bram Cohen as the default difference tool to Bazaar version control system [Cohen, 2003].

  – *BDiff* [Proceedings, 2006]. The BDiff algorithm is the part of the *Python difflib* library [Hellmann, 2011] and used by Mercurial for both delta storage and difference representation. The BDiff algorithm searches for the longest common continuous substring within compared files and recursively in the part preceding/succeeding it.

  – *XDelta* [MacDonald, 2000]. The XDelta algorithm produces *copy* and *insert* instructions as the output. The first of compared revisions is splitted into the smaller blocks and each block is put on a hash table. The same hashing is done for the second object as well. Then, all possible matches are found in the first hash table. The *copy* instruction is derived for the largest match. The *insert* instruction is then generated for unmatched data in the second hash table. This algorithm is used in Subversion [Collins-Sussman et al., 2004], Monotone [Hoare et al., 2005] and Git [Swicegood, 2008] for internal storage, i.e., generating one-way binary differences between arbitrary non-textual blobs.

- **Delta Formats.** The delta algorithms discussed above are used to compare the revisions of the same software artifacts and produce difference documents (deltas), whereas *delta formats* are used for storing deltas produced by the delta algorithms. Principally, delta algorithms and delta formats are strongly tied to each other.

  – *Unified Diff* [Davison, 1990]. The Unified diff format is a standard used in "patch" files. The differences consist of the *change chunks* for each

file considering only changed artifacts. The change types are distinguished by three operations. These are additions; marked with "+" sign, removals; marked with "-" sign, and modifications; marked as "+/-" sign. Git takes advantage of the extended form of the unified diff in representing merges.

- *RCS Delta* [Tichy, 1985]. The RCS approach represents changes in deltas using textual formats. In case of the development branches, deltas are stored in reversed order and newer revisions are represented by *backward deltas*. Deltas consist of lines and each line contain the letter `a` for additions, `d` for deletions followed by line numbers, and the enumerations of lines which have to be added or deleted.
- *Weave* [Rochkind, 1975]. The weave format introduced by SCCS [Rochkind, 1975] is used in BitKeeper [Henson and Garzik, 2002]. This approach represents all revisions, listing all lines that appear in the file together with its revisions.

In addition to these approaches, there are several other repository formats such as *Git Packs* [Swicegood, 2008] which is very effective in size. It uses pack-specific objects for representing deltas in binary formats [Baudivs, 2014].

As these discussions show, there are several difference representation formats and repository models for source code-driven software systems, textual files, data objects or binary files using textual, binary or object formats. The most approach consider line-by-line differentiation techniques considering line additions, removals or line modifications, or consider object as a whole. These repository storage models and delta formats can not directly used in representing model differences in collaborative MDSE because of paradigm shift between code-driven and model-driven software development technologies.

3.2.2 Sequential Collaborative Modeling

According to discussions in Section 3.2.1, there are several advanced sequential collaboration approaches for source code-driven software projects, textual files, and object files. As MDSE is becoming widely accepted and used technology in current day's software development activities, there is a need for extended research on model difference representation, i.e., delta formats in sequential collaboration for MDSE. Generic delta format approaches for sequential collaboration support is required regardless of modeling languages.

Similar to the code-driven artifacts of software systems, the design level models of software systems are definitely the subject to changes during development and evolution. In case of code-driven software development and evolution, changes are mostly considered in the line-by-line forms, i.e., line additions, line removals, and line deletions. But, this approach of change identification and representation can not be utilized in case of sequential collaborative modeling. Because software

models have composite and graph-like structures representations (as discussed in Section 2.4).

Proposed sequential collaboration infrastructures for MDSE must implement mechanisms to deal with consistency in the shared modeling artifacts, to view data shared by the team members and consider the associated data structures of models. Thus, sequential collaboration systems have to be well-advanced enough for handling model-driven software development and evolution. The adapted process of sequential collaboration is needed that addresses the challenges arise by having software models as the subject of evolution.

There are already several sequential collaboration approaches and research prototypes for MDSE and evolution. Below, some of these existing approaches are partially discussed in order to study the state of the art and derive the related terminologies. The extended literature review and discussions about the underlying delta formats of existing sequential collaborative modeling approaches are given in Chapter 4.

Software Repositories. The software repository architectures such as *centralized* and *distributed* are usually used for both, source code-driven and model-driven sequential collaboration. As long as the software repository architectures are only the organizational subject, the same repository architectures can be reused for sequential collaborative modeling.

Repository Storage Models. The *snapshot-oriented* and *changeset-oriented* models are discussed as the repository storage models in Section 3.2.1. In case of the sequential collaboration, the most existing approaches take advantage of the *changeset-oriented* approach for identifying and representing their repositories (e.g., [Saeki, 2006], *CoObRA* [Schneider et al., 2004], *SMoVer* [Altmanninger et al., 2007], *AMOR* [Langer, 2011], etc). Because, the *changeset-oriented* model is more efficient than storing complete model revisions after every small change. Further motivation and discussions can be found in Section 4.2.6. As discussed in Section 8.1, this thesis also takes advantage of the changeset-oriented model for developing its sequential collaborative modeling application.

Delta Representation Techniques. Section 3.2.1 has reviewed two different parts of delta representation techniques such as *delta algorithms* and *delta formats*.

- *Delta Formats.* There are several delta formats for representing model differences in the field of sequential collaborative modeling. These are classified into four groups such as *text-based* [Appeldorn et al., 2018], *model-based* [Cicchetti et al., 2007, Taentzer et al., 2012], *graph-based* [Kehrer et al., 2013a], *Relational Database-based* [Altmanninger et al., 2007]. As long as the difference representation for collaborative MDSE is the core research objective of this thesis, Chapter 4.1 is dedicated to extended literature review in model difference representation approaches (i.e., delta formats).
- *Delta Algorithms.* In addition to delta formats, providing a catalog of supplementary services is further research objective of this thesis as explained in

Section 1.1. Thus, Section 4.2 provides extended discussions on the existing delta algorithms for collaborative modeling.

As described in the content of this chapter, collaborators checkout the shared, central software models into their distributed environments, modify their copies of models, and eventually check in (merge) their changes into the central model. Usually, this scenario is constantly performed by several collaborators. It results in several different subsequent revisions of the same modeling artifact. Alternatively, during concurrent collaborative modeling, collaborators may save their models under development whenever they are correct and complete which eventually results in the new subsequent revisions of the same modeling artifacts. According to *changeset-oriented (delta-based)* difference representation, only changed modeling artifacts are identified and represented in deltas, instead of storing complete model as sequential revisions. In order to represent model differences in deltas, there is a strong need for a suitable and appropriate difference representation notation.

Difference Representation in Sequential Collaboration. According to discussions in this section, changes are the first-class citizens in sequential collaborative modeling, as well. Representation of changes plays an essential role in developing advanced sequential collaborative modeling systems. As discussed in Section 3.2.1, there are multiple change representation approaches (i.e., delta formats) for source code-driven sequential collaboration. Software models can not be treated as neither source code nor blob objects because of their graph-like structures and syntax [Kleppe et al., 2003]. Software models can be represented in textual formats using XMI exchange formats, but it is commonly agreed that differentiating the textual representations of software models do not provide sufficient information for storing the model histories in sequential collaboration [Cicchetti, 2008], [Steinberg et al., 2008]. Nevertheless, the existing technologies of sequential text-driven collaboration systems can be utilized on the architectural level. Moreover, similar principles, concepts and terminologies on the higher level may be taken over and reused for developing advanced sequential collaboration support for MDSE and evolution. There is a strong need for a generic and extensible difference representation language regardless of modeling languages, modeling designing tools and other underlying technical spaces.

3.2.3 Required Support

According to discussions so far, sequential collaborative modeling approaches are not competitively well advanced in comparison to sequential text-driven collaboration approaches. Like concurrent collaboration, difference representation (delta formats) lies at the core of sequential collaboration. Therefore, there is a strong need for an extended research on a novel and solid means for model difference representation for sequential collaborative modeling.

Section 3.1.3 requests a *difference representation notation* for concurrent collaborative modeling. The same underlying difference representation notation can also

be used for sequential collaborative modeling providing the combination of both collaborative modeling scenarios. Likewise, the same underlying representation notation can be suitable for sequential collaboration. A common underlying notation can serve as a generic delta representation and change exchange format for both concurrent and sequential collaborative modeling scenarios enabling development of any component or service on the top.

In the field of concurrent and sequential text-driven collaboration, the delta formats for representing differences do not rely on any source code syntax, textual file types or object types. Moreover, this kind of features are not sufficiently covered by the existing difference representation approaches for collaborative MDSE as discussed in Chapter 4. Like the delta formats for concurrent and sequential text-driven collaboration, an efficient delta notation is required for representing differences and changes in collaborative MDSE. It should not rely on a particular modeling language, modeling tools, delta algorithms or other underlying technical spaces. It has to be generic allowing for development of further operative services and components on the top, yet serve as the common underlying delta representation format for both, concurrent and sequential, collaborative modeling. To this end, this thesis strongly requests a generic *difference language* for representing changes/differences in deltas for the both scenarios of collaborative MDSE.

3.2.4 Expected Benefits by Difference Language

In case of sequential collaborative modeling, change representation should definitely be based on *changeset-oriented storage* technique, i.e., only changed modeling artifacts have to be considered in delta documents. As long as modeling concepts follow associated and composite graph-like structures, considering only changed modeling artifacts has a great deal of time and storage memory efficiency. Eventually, it allows for storing only the changes in deltas. It plays an essential role in storing the small set of changes in case of a huge amount of model revisions. A difference language for representing model differences yields several significant contributions to sequential collaborative modeling, as well.

1. It may facilitate tool developers' productivity with precise, concise and clear descriptions (extensible).
2. It can be declarative enough by making any concepts or mechanisms implicit that can be intuitively interpreted from the context.
3. It may convert a software model from one revision to another by being directly executable descriptions of model changes.
4. It can be fully expressive, yet unambiguous and provide necessary knowledge about each change.
5. It may embody the only changed parts of software models saving memory and time.

Model changes are quite crucial in case of the large-scale software models designed by several collaborators where models have several sequential revisions and several parallel development branches. Consequently, difference language brings several advantages to sequential collaborative modeling, particularly. By using simple difference representation notations, sequential collaborative modeling may reduce the difference storage space and improves simplicity by storing only changed artifacts. Changes can be efficiently exchanged among various development branches of software models under development and evolution. If a difference representation notation is generic with respect to the meta-models of modeling languages and independent of underlying technical spaces, it can be applied to the wide range of modeling languages and model designing tools.

These are the expected contributions of difference representation notation provided by difference language. These benefits are generalized in Section 4.3 as the set of requirements for overall difference representation language.

3.3 History Analysis

Since software systems evolve undergoing different changes resulting in several different revisions, the history of evolving software systems is quite interesting for project managers, designers, developers, and other stakeholders. Development team members are usually interested in which collaborator made particular changes, how the development and evolution processes are going on. They aim at being able to keep the development and evolution processes under their control, as well as make the further appropriate decisions on improvements of their software project.

Often the present of models is understandable by looking at their past. Moreover, in concurrent collaboration, history analysis enables users to visually compare their current revision with previous ones or the other parallel revisions of the same modeling artifacts. The users can specify changes they want to save and discard.

In software evolution, [Ducasse et al., 2005] defines three main terms: *revision*, *evolution* and *history*. A *revision* is a snapshot of a software artifact at a particular moment of time. The *evolution* is the process that leads from one revision to another. *History* is the reunification which encapsulates knowledge about evolution and revision information. According to these definitions, the history is used to understand the software evolution.

During development and evolution processes, software systems are usually stored in *software repositories* [Arnold, 1996]. Analyzing the evolutionary life-cycle and history of software systems is the part of the general software engineering activity entitled *mining software repositories (MSR)* [Godfrey and Tu, 2002, Kagdi et al., 2007].

Definition 3.5. **Mining Software Repository.**
[Kagdi et al., 2007] defines Mining Software Repository (MSR) as follows:
"The term mining software repositories (MSR) has been coined to describe a broad class of investigations into the examination of software repositories. Here software repositories refer to artifacts that are produced and archived during software evolution. They include sources such as the information stored in source code version-control systems (e.g., Concurrent Versions System(CVS)) requirements/bug-tracking systems (e.g., Bugzilla), and communication archives (e.g., e-mail)."

The term *MSR* has a very broad meaning and this thesis refers to it as *history analysis* of software systems, later on, software models. Thus, this section limits the scope of MSR which covers examining the multiple revisions or change-sets of software artifacts during their development and evolution. The revisions of software systems or evolutionary change-sets are specifically investigated in order to analyze the evolutionary life-cycle of software systems with the particular purpose asking questions such as *why? who? and when?*. The people involved in development and evolution of software systems obviously want to analyze how changes they made impact on their software systems under evolution and development.

Analyzing and viewing the histories of software systems is significant support for software stakeholders in order to comprehend necessary knowledge about the revision histories. The advanced querying systems might bring several conveniences to the software collaborators. These conveniences are discussed in Section 3.3.4.

Information Resources (Software Repositories). As discussed so far, multi-versioned and shared software artifacts are usually stored, archived in software repositories which serve as **information resources** for analyzing the evolutionary history of software artifacts. Both, concurrent and sequential, versioning systems utilize software repositories for storing and archiving their software artifacts. The software repository types vary in their exploitation, information contents, and storage formats. Furthermore, these repositories are managed and operated in isolation by the tools and approaches built on the top of these repositories [Kagdi et al., 2007]. Nevertheless, the common goal of these repositories is to support software evolution by managing the life-cycle of **software change** [Robbes, 2007].

[Kagdi et al., 2007] classifies three basic categories of information in software repositories that can be mined:

– *Differences between Artifact Revisions.* As long as software changes are the first-class citizens in development and evolution of software systems, the differences between artifact revisions are always the main subject in analyzing the evolutionary life-cycle of software artifacts.
– *Software Artifact Revisions.* Software artifact revisions are mainly the states of software systems before and after making changes. Information about different revisions can be queried from software repositories. Information about

revisions might be different according to the purpose of mining, analysis and even the type of software artifacts.

– *Meta-Data.* In addition to software revisions and the differences between these revisions, meta-data such as commit comments, user names, timestamps, and other similar data is also necessary information in history analysis. These meta-data describe, respectively, why, who and when the context of software artifacts change [Kagdi et al., 2007].

Purpose. Software repositories are mined for different purposes such as extracting necessary, useful information and/or detect relationships, consistencies according to particular evolutionary characteristics or metrics [Bieman et al., 2003]. These characteristics and metrics vary based on what kind of software artifact is being analyzed, e.g., textual, source code, object type, graph or tree-like structures, etc. Mining purposes might vary based on the purpose of stakeholders, for instance, one may be interested in the growth of systems, i.e., the amount of software artifacts, change relationships between software entities, or most instantly changed artifacts and so on. Due to studying particular characteristics, and define the scope and context of mined information, the purpose is typically expressed as a set of questions. Thus, the purpose of mining reduces to what questions can be answered by software system analysis [Kagdi et al., 2007].

[Wenzel, 2008, Kagdi et al., 2007] broadly define two classes of software system analysis questions. The first class is the *market-basket questions* formulated as: if something happens then what else occurs on a regular basis? The answer for this kind of questions is the set of rules or guidelines describing situations and their relationships. The second type of analysis purpose is related to *prevalence questions*. Thereby, questions include metrics and boolean queries. For instance, was a particular software artifact created/deleted/changed? These questions indicate the purpose of the analysis scenario.

History Analysis Steps. In general, the process of the history analysis of software systems undergoes several sub-steps [Robbes, 2007]:

– *Artifact Representation.* Different repositories store or archive software artifacts using different techniques such as source code, abstract syntax trees (AST) [Robbes, 2007], control-flow graphs [Kim and Notkin, 2006], software models [Raumbaugh et al., 2004].

– *Change Operation Definition.* Change operations represent the actual evolution of software artifacts. Change operations are usually defined in two forms: *atomic operations* such as creations/additions, deletions/removals, property changes, and *composite operations* such as moves [Robbes, 2007]. Change operations are usually stored in difference documents sometimes referred to as *delta documents* [Zimmermann et al., 2005] using *log records* [Hindle and German, 2005], *relational databases* [Robles et al., 2004], *file systems* [Berlin and Rooney, 2006], *software models* [Cicchetti, 2008], *graph-like structures* [Kehrer et al., 2013a], etc.

– *Data Extraction/Retrieval.* This is the process of querying and fetching analysis information from software repositories. This step is the main operation which facilitates software history analysis with retrieval of necessary and pertinent information from software repositories. Different techniques are used in data retrieval (sometimes referred to as *repository mining*) phase according to data formats stored in software repositories. For instance, *graph queries* [Kullbach et al., 1998] are used to extract information from the graph-like structures, SQL queries [Date and Darwen, 1987] are used to query relational databases, etc.

– *Data Browsing, Visualization.* Data browsing and visualization are the central ingredients for any software history analysis solution. Difference information obtained from the data retrieval phase needs to be properly visualized in readable and convenient ways so that it can be used for subsequent analysis and manipulations. According to literature, visualization of software repositories and/or their differences can be differentiated in the following ways:

  Text-based presentation lists the differences in form of plain text or in some structured formats, e.g., XML. Modern text-based visualization techniques, for example, the technique described in [Suvanaphen and Roberts, 2004] provides a means for the overview, zooming, filtering. However, the overview, zooming, and filtering, in this case, are syntax-based and do not provide much insight into the meaning of differences.

  Graph or tree-based visualizations (e.g., [Wenzel, 2008], [Collberg et al., 2003]) show all model or source code artifacts in lists or in trees if they have hierarchical structures. But getting the overview of the differences is not so easy for the larger amount of software artifacts because the size of the visible part of trees is limited by the size of displays. The details of differences are still not easy to comprehend, since the users need to interpret the tree representation of differences.

  Diagrammatic visualization [Schipper et al., 2009] offers the diagrammatic view for visualizing differences. In this approach, it is hard to extract all artifact types having a certain property, because these artifact types might be in different parts of software system, and thus they might only be visualized in different diagrams.

  Difference highlighting approach [Wenzel, 2008] highlights the differentiated software artifacts with different colors. For instance, creations might be highlighted with green color, deletions with red color and changes might be marked with another color. In this approach, moves cannot be displayed sufficiently. However, the approach does not scale up to the large amount of software artifacts due to display limitations.

  Tabular view gives the first overview about those software objects complying with the appropriate query. These query results are directly linked to the source code if possible [Ebert et al., 2002].

Section 3.3.1 reviews the core ideas and basic concepts behind the history analysis for source code-driven software development. Section 3.3.2 derives some common

basic concepts and terminologies from the source code-driven software history analysis and inspects the history analysis for MDSE and evolution.

3.3.1 Text-Driven History Analysis

There have been several approaches to analyze long-term software-project data in order to understand software evolution. [Fernández-Ramil and Lehman, 2000] has reported various results on the changes in the software and nature of software evolution based on long-term studies such as the laws of software evolution [Lehman, 1996], metrics of software evolution [Fernández-Ramil and Lehman, 2000] and classification of programs [Lehman and Fernández-Ramil, 2001]. These reports serve as theoretical grounds for several repository mining approaches.

Hassan [Hassan, 2008] distinguishes between two kind of repositories:

Historical repositories are usually utilized in case of the sequential collaboration scenario such as source code control repositories (as explained in Section 3.2.1) (the source code of various applications developed by several developers), bug repositories, and archived communication record information about the evolution and progress of projects. Monotone [Hoare et al., 2005], Git [Swicegood, 2008], Mercurial [Mackall, 2006], Darcs [Roundy, 2005], Subversion [Collins-Sussman et al., 2004] sequential collaboration tools are the examples for these repositories.

Run-time repositories such as real-time editing logs contain information on the execution and development of applications by multiple developers. This kind of repositories are used in case of the concurrent collaboration scenario. For instance, Google Docs [Google Inc., 2017], Etherpad [AppJet Inc., 2017], Firepad [Firebase Inc., 2017] are widely used in document creation and editing in real-time.

Since the historical repositories store the subsequent revisions of software artifacts, the repository mining approaches based on the historical repositories usually provide history information about the subsequent revisions of software source code, the differences between subsequent revisions and meta-data about their commits. The repository mining approaches built on the top of real-time repositories usually contribute to the comparison of parallel revisions, displaying instant updates, displaying real-time log messages, user lists and other related data. In this thesis, the both concurrent and sequential collaborative modeling scenarios are built on the top of the same underlying repository.

Repository mining and history analysis approaches used in concurrent and sequential collaboration systems for source code-driven software development and evolution can extract information from their repository formats. However, they can not fully handle repositories for MDSE and evolution. They do not take the specific semantics of software artifacts into account because everything is treated as the ordinary piece of text by them [Mens, 2002]. These techniques established

on top of the concurrent and sequential text versioning systems operate on textual documents and source code under the linear, the sequence of characters, file-based paradigm that is purely textual. As long as models are not sequential text lines, but could be represented as trees or/and graphs, these systems cannot use the formal structures of software models to provide operating data extraction and suitable analysis features concerning syntax and semantics [Kofman and Perjons, 2004], [Lin et al., 2004].

3.3.2 Model History Analysis

During the evolution and maintenance process of software models, model designers feel a need for history analysis support for tracing and comprehending the evolution history of models in general and their particular artifacts. In order to analyze the histories or trace a particular artifact of evolving models, designers need to determine answers to several questions such as:

1. How often does a modeling artifact change?
2. When was a modeling artifact created?
3. When was a modeling artifact deleted?
4. Which modeling artifacts are constantly changing?
5. How does the history of a modeling artifact look like?
6. How was the state of a whole model in earlier revisions?
7. What are the differences between any two revisions of a model?

For answering these questions, the change histories of modeling artifacts have to be identified and stored in appropriate ways for further analysis and manipulation [Kuryazov and Winter, 2015b], [Wenzel, 2008], [Wenzel, 2010].

There are only few approaches dedicated to history analysis for model-driven software development and maintenance. There are approaches providing software model repositories such as [Schneider et al., 2004] and [Oliveira et al., 2005], but they do not provide change tracing and history analysis features. The terminologies and techniques described in the content of this chapter are suitable for model history analysis, as well. Thus, below, few approaches [Xing and Stroulia, 2005a], [Godfrey and Tu, 2002], [Wenzel and Kelter, 2008] and [Kehrer et al., 2012] are discussed by matching these terminologies.

An approach addressing to model change tracing and history analysis is introduced by Sven Wenzel in [Wenzel, 2010] and [Wenzel and Kelter, 2008]. This approach uses EMF-based models as its *data repositories/resources*. In the first stage of *data retrieval*, the approach creates the history repository from the model revisions in repository. It detects model *change operations* using model comparison as the

part of data extraction. In the history creation phase, the approach extracts revision information and one graph representation for each *model reversion* in the repository. It then stores *traceability data and evolution data* by adding fine-grained element information to revision information and graph representations. By this step, the approach creates mapping between the model repository and the history repository. This is also referred to as traceability information. Traceability information helps (1) to identify modeling artifacts across their evolution and (2) to follow the modeling artifacts of model revisions to the corresponding artifact in the ancestor and descendant revisions respectively, if the element exists in that revision of the model.

Traceability information can be calculated incrementally whenever the new revisions of models are created. The identification links are computed by the model matching approach of SiDiff algorithm [Treude et al., 2007]. The approach further provides the *data browsing* feature for detected history information by the change tracer. It takes advantage of *coloring* and *graph-based* visualizations and data browsing techniques. Moreover, model revisions can be opened in *textual forms* using *tree-like visualizations* with multiple revisions. The occurrences of the selected artifacts can be traced, and the traced modeling artifacts and their occurrences are visualized in *different colors*. In the visualization component, modeling artifacts and their correspondences are visualized in different colors and notations.

The SiLift approach [Kehrer et al., 2012] aims at comprehending model evolution through semantic lifting model differences from the low-level graph-based representations to the higher-level, composite difference representations. As the extension of the SiDiff [Schmidt and Gloetzner, 2008] algorithm, SiLift uses low-level graph-based difference representations produced by the SiDiff algorithm as its *data sources*. In the *data extraction* phase, the approach defines the mappings between the low-level and high-level differences using the Henshin graph transformation rules. Eventually, the low-level differences are then transformed into the high-level, human-readable difference representations and visualized using *model-based* and *coloring* techniques.

Xing and Stroulia [Xing and Stroulia, 2005a] presents an evolution analysis approach for object-oriented software systems. The evolution of source code is translated into models. This particular evolution is not comparable to the evolution of models in MDSE. Another approach by Godfrey and Tu [Godfrey and Tu, 2002] deals with the problem of tracing source code entities over time. The main objective of the so-called origin analysis is the structural evolution of software systems. It addresses renaming or moving of code artifacts. These approaches reside in the domain of source code evolution, the parts of these concepts can be transferred to the MDSE context. However, they are neither investigated nor applied in MDSE.

3.3.3 Required Support

The main challenge in software model history analysis is an advanced query mechanism, which is crucial for retrieving artifacts from model repositories according to different criteria. For instance, software models can be searched by considering the corresponding modeling concepts, notations, domain type, or development phases. To achieve successful results in model history analysis, the representation formats of software model repositories must be easily accessible, reusable and extensible with compact syntax and notations. Repository mining should not avoid the model-driven concepts and the syntax of software models when the extracted data is displayed to collaborators. This allows for analyzing the model histories by referring to these modeling concepts.

To sum up, model history analysis is still in its infancy. In Section 3.2 and Section 3.1, a *difference language* is requested for difference representation in MDSE and evolution. As long as the concurrent and sequential collaborative modeling scenarios can be built on top of the same underlying difference representation language, the model history analysis scenario can also use the same common repositories. Eventually, these repositories can be used as *information source* for the history analysis applications in the context of MDSE.

3.3.4 Expected Benefits by Difference Language

Since software models are the visual form of software system design and consider all aspects of design level concepts, a proposed difference language can serve as suitable information resource for advanced querying, browsing and visualization features for collaborative MDSE.

1. It may improve the performance of data extraction by difference language notations.
2. It may allow for tracking model changes by following actual modeling concepts.
3. Whole models or their certain aspects can be queried and query results can be browsed and visualized in different ways.
4. The history of whole models or their particular artifacts can be traced without loosing relationships between sequential and concurrent revisions.

The list of expected benefits are the contributions of a difference language to particularly model history analysis use case. This thesis aims at achieving aforementioned expected benefits in model evolution history analysis by applying a proposed *difference language*. The model history analysis application of this thesis takes advantage of the tabular view and graph visualization techniques for visualizing its repository query results.

3.4 Summary

The actual state of the art in concurrent, sequential collaboration and history analysis use cases for source code-driven/textual and model-driven software development are briefly discussed in this chapter. These discussions have shown that having a suitable model difference representation might significantly contribute to concurrent, sequential collaborative modeling as well as model history analysis. These discussions helps to learn several research lessons as follows:

- There are several efficient and suitable difference or change representation approaches for source code-driven software development and evolution.
- A number of solid and sophisticated concurrent, sequential collaboration and history analysis applications and tools are established on the top of the existing difference representation approaches for source code-driven software development and evolution.
- The source code and text-based difference representation approaches can not be applied to the representation of model-driven software differences and changes because of paradigm shift between *linear* source code-driven and *graph-based* model-driven software development concepts.
- Difference representation lies at the core of both collaborative modeling scenarios and history analysis support. Thus, there is a strong need for extended research in sophisticated, extensible, reusable, applicable and efficient ways of *difference language* for collaborative MDSE. It can be a common underlying representation technique for the concurrent and sequential scenarios of collaborative modeling as well as model history analysis.
- A catalog of supplementary services can be developed on the top of a proposed difference representation approach which can help to extend the application areas of difference representation.

The research and applications in model difference representation field are still in early ages. Consequently, this thesis addresses model difference representation by *a difference language* in collaborative MDSE as its primary research question. Furthermore, it aims at providing *a catalog of services* which can produce, reuse and manipulate model differences represented using a proposed difference language. Eventually, the *concurrent and sequential collaborative modeling scenarios* as well as *model history analysis* in MDSE can be elaborated by the specific orchestration of the provided services.

Chapter 4

Related Approaches To Difference Representation

The problem of model difference representation is the actively discussed and extensively addressed topic among the research community in software engineering and modeling field. There is a large number of research papers addressing the problem of model difference representation and its certain aspects. In order to study the existing approaches and the state of the art, an extensive literature review is done in the framework of this thesis. This chapter gives a brief overview about the state of the art in the field of model difference representation. Several existing model difference representation approaches are studied and analyzed in order to clarify what aspects of model difference representation are already covered and which ones are still remaining to be solved.

The related approaches employ various techniques for describing and storing model differences and changes in deltas. For instance, some approaches use the *model-based* way of difference representation, i.e., the differences between model revisions are described in individual models so called the *difference models*. There are some approaches which use *textual operations* (similar to difference language in this thesis) as techniques to describe difference representation. Even, some approaches take advantage of *relational databases* to store model differences. In addition to difference representation techniques, these approaches further provide *supplementary services* to exploit and reuse their difference representation information dealing with certain aspects.

This chapter studies existing approaches according to, firstly, their model difference representation technique in Section 4.1, secondly, the supplementary services they provide to reuse and exploit their difference representation information, in Section 4.2. As the result of the literature study, the learned lessons are discussed in Section 4.2.6 so that the open research challenges are highlighted in the research field. Section 4.3 defines several requirements for model difference representation and its applications. Finally, this chapter ends up by drawing some conclusions in Section 4.4.

4.1 Model Difference Representation

In Section 3.2.1, multiple difference/change representation approaches are discussed for source code-driven software development. There exist several delta representation approaches in collaborative MDSE, as well. They employ various forms and techniques for describing and storing model differences. This section classifies and discusses the existing delta representation approaches for collaborative MDSE according to their model difference representation techniques.

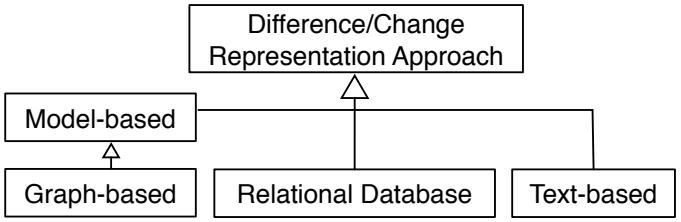

FIGURE 4.1: Related Approaches

Figure 4.1 depicts the classification of the most widespread difference/change representation approaches that are addressed in this thesis. This classification is the result of literature study in the framework of this research work.

- *Model-based.* Model differences or changes are represented in software models, e.g., usually using UML class diagrams. This is the specific and more abstract type of the graph-based representation. This class of approaches is discussed in Section 4.1.1.
- *Graph-based.* This technique is the special form of the model-based representation, but relying on the low-level graph-like structures. The graph-based model difference representation represent model differences in the graph-like structures on more detailed and technical ways. These approaches are enlightened in the same section with model-based approaches in Section 4.1.1.

- *Relational Database.* Several model difference representation approaches take advantage of the relational databases for storing/representing their model differences as discussed in Section 4.1.2.
- *Text-based.* Model differences or changes are represented by a sequence of edit operations in textual forms embedding change-related difference information. The delta representation approach proposed in this thesis also belong to this class of difference/change representation approaches. The text-based representation approaches are discussed in Section 4.1.3 in detail.

Operation-based. Before starting to inspect and study the related approaches, a clear definition of the term *operation-based* must be given. Most of the related approaches identify themselves as *operation-based difference representation* techniques. Because they usually utilize basic edit operations such as `create`, `delete` and `change` (or similar and more) which is a general concept being relevant to many difference representation approaches. Regardless its difference representation technique, these approaches use the aforementioned basic operations only to

recognize the types of changes. But, model changes are generally stored in various forms like models, graphs, relational databases or textual forms, as discussed in the remaining sections of this chapter.

4.1.1 Model- and Graph-based Difference Representation

In model- and graph-based difference representation approaches, model differences are represented again in software models so-called *difference models (graphs)*. As long as the model-based difference representation approach is the special form of the graph-based approaches, this section studies the model-based difference representation approaches together with the graph-based ones.

The graph-based approaches represent model differences or changes using internal graph-like structures. In graph-based approaches, model differences are represented by several low-level (i.e., graph-level) change operations, whereby they can be composed as only operation in case of model-based difference representation.

A meta-model independent approach to model difference representation. In [Cicchetti et al., 2007] and [Cicchetti, 2008], Cicchetti et al. introduced a meta-model independent approach to model difference representation which is agnostic of difference calculation. The approach uses software models for representing model differences. Model differences are described in difference models which conform to difference meta-models. The difference meta-models themselves are derived from the base meta-models of modeling languages by automatic transformations applying three atomic operations `add, change, delete` to the concepts of the given meta-models. These basic operations are attached to each concept of difference meta-models so that the differences between instance models can be represented by the change operations derived from that difference meta-models.

In the approach by [Cicchetti et al., 2007], the derived difference meta-models are utilized for representing model differences on the instance level. The approach distinguishes three types of change operations as follows:

- `Additions.` New modeling artifacts that are not present in the previous revision of a model are added to the next revision.
- `Deletions.` Existing modeling artifacts are deleted from the previous revision that should not exist in the final revision of a model.
- `Changes.` Existing modeling artifacts are updated when a model is evolved from the previous revision to the next revision.

The approach is applied to several modeling languages with respect to their meta-models. The authors intend to find a suitable representation for model differences which is agnostic of the calculation method and permits to harness the potential offered by generic modeling platforms. Thus, they identify several natural properties their representation technique should provide. They also argue that

the outcome of difference calculations must be represented using models to conform to the spirit of "everything is a model" principle. Therefore, in order to find a solid and suitable model difference representation approach, they state several prerequisites that the model difference representation technique should satisfy:

- *minimalistic*, difference models must contain only the necessary information to represent model differences/changes, without duplicating the parts of those models that are not changed;
- *transformative*, difference models must induce model transformations, such that whenever they are applied to the initial model, it yields the final one. Moreover, the transformations must be applicable to any other model which is possibly left unchanged, if the modeling artifacts specified in the difference models are not contained in it;
- *compositional*, the result of subsequent or parallel modifications is a difference model whose definition depends only on difference models being composed and is compatible with the induced transformations;
- *meta-model independent*, the representation techniques must be agnostic of the respective meta-models. In other words, it must be not limited to specific meta-models.

The discussions above outlines the minimal set of requirements which should be taken into account in order to let a generic modeling platform to deal with advanced model version control facilities. Additionally, the approach provides several supplementary services that are discussed in the follow up sections of this chapter.

AMOR – Adaptable Model Versioning System. A fundamental approach to model version control based on graph modifications is introduced in [Taentzer et al., 2012]. It represents model differences by *difference models*. The approach is validated in the Adaptable Model Versioning System (AMOR) [Langer, 2011] for EMF models [Steinberg et al., 2008]. The major focus of the approach is the differentiation and merging of software models that serve as the main foundations for sequential collaborative modeling. The approach employs model matching algorithms to produce mappings between modeling artifacts in the original and revised versions of models. The identified correspondences by model matching are described by match models conforming to the match meta-model. For each pair of matched modeling artifacts, a match model consists of the instance of the class `Match` connecting the corresponding model artifacts in the original and the revised versions. If the model elements are not matched, the instance of the class `Unmatch` referring to the unmatched model artifacts is created. The attributes of meta-classes indicate whether the unmatched object resides in the original or revised model.

According to graph modifications in [Taentzer et al., 2012], nodes and edges may be inserted or deleted. For expressing such changes in EMF models, the approach uses two concrete sub-classes of `FeatureChange` in the difference meta-model, namely `InsertFeatureValue` and `DeleteFeatureValue`. Feature changes are referred to as `changedObject`, to the changed feature in the meta-model. In case of a

reference, this value is an object and in case of attributes, values are the primitive value of type String, or Boolean, etc. The meta-model also contains the two classes `InsertObject` and `DeleteObject`, that are the sub-classes of the abstract class `ObjectChange`. The container of inserted or removed objects is specified through the reference `changedObject` and the containment feature by the reference `changedFeature`.

As long as the approach proposes the solid foundations for building sequential model version control systems, it intends to consider every detail in model differentiation and model difference merging. Thus, the approach extends the kernel difference meta-model. The single-valued and multi-valued multiplicities of features are extended as well. They introduce `FeatureUpdate` that represents the change of a single-valued attribute or reference in addition to `InsertFeatureValue` and `DeleteFeatureValue` for multi-valued features. They further define ordered features that enrich the kernel difference meta-model by the classes `InsertOrdered FeatureValue` and `DeleteOrdered FeatureValue`. By defining these kind of operations, the approach intends to cover all possible conflict scenarios which can occur during merge process. Furthermore, it considers the special combination of two `FeatureChanges`, which is insertion and deletion of one and the same object in different containers. In this case, the object is moved from one container to another, Thus, the `Move` operation is derived consisting of two feature changes: one `InsertFeatureValue` and one `DeleteFeatureValue`.

SiDiff Approach. A generic model difference representation approach using edit scripts is introduced in SiDiff [Treude et al., 2007]. The SiDiff approach consists of a chain of model differentiating processes for EMF models which include correspondence matching, difference derivation, and semantic lifting phases. Initially, the SiDiff approach represents model differences by the *low-level change sets* considering the graph-like internal representations of software models. As long as software models are usually represented by the low-level graph-like structures, it makes more sense and quite essential to consider the internal representation techniques of software models. The low-level differences are the outcome of the difference computation phase.

In the framework of the SiDiff approach, the complete chain of data structures and activities performed can be distinguished in the following steps:

- *Low-Level Differences.* The low-level differences are the outcome of model matching represented by the graph-like structures. Model differences consist of correspondences between the modeling artifacts of two revisions, as well as changes like additions and removals of modeling artifacts. The low-level changes are represented using EMF models and the related low-level changes are discovered by the pattern matching engine of Henshin transformations [Arendt et al., 2010].
- *Semantic Lifting.* The low-level model differences are then lifted to the semantic change sets by the SiLift approach [Kehrer et al., 2012] using *recognition rules*. The basic idea behind the SiLift is to group low-level changes into semantic change operations according to change patterns. The resulting

semantic change sets are then more expressive notations for comprehending changes such as additions, removal and re-locations of modeling artifacts. The recognition rules always operate on the low-level differences.

– *Operation Specification.* The formal parameters of the lifted change operations in the semantic change set are retrieved from the operation specifications of the edit operations by analyzing the dependencies among edit scripts.

The approach distinguishes the following types of change operations:

– `Attribute Change.` Two corresponding artifacts differ in their attribute values.

– `Reference Change.` The references between two corresponding modeling artifacts are changed.

– `Move.` References connecting parent and child artifacts are reconnected to other parents so that the child artifacts are considered as moved (relocated) artifacts.

– `Structural Change.` Artifacts that have no entry in the correspondence table are considered to be structurally different, i.e., due to insertion or deletion operations.

In the SiDiff approach, software models are represented by the graph-like structures of EMF and aforementioned mentioned steps are implemented using Henshin model transformation rules [Arendt et al., 2010].

The SiDiff approach offers two options of representing model differences. The first option is to serialize the internal EMF representation of the difference data structures. The second option is to employ textual notations. The sequence of operation invocations can represent the list of edit scripts. There, the formal parameters are given as objects. The first option can be used for loading edit scripts which is useful for later reuse in difference application or merge tools. Eventually, the difference representation approach of SiDiff can be considered both model-based using the graph-like serializations and text-based (textual) approach.

The SiDiff approach defines the complete (mandatory) set of edit rules for the considered modeling languages. The mandatory edit rules are generated from the meta-models of modeling languages by SiDiff Edit Rule Generator (SERGe) [Kehrer et al., 2013b]. The edit scripts, matching patterns, recognition rules, parameter retrieval and dependency analysis components are implemented in Henshin transformation engine [Arendt et al., 2010] for EMF models. The approach is applied to Ecore-based UML [Stephan and Antkiewicz, 2008] models and Matlab/Simulink [Ong, 1998] models. Due to the representation of model differences, the authors elaborated the 41 different types of edit operations for UML class diagrams and 16 different edit operations for Matlab/Simulink models. The authors further identify 148 mandatory edit rules for Ecore models and 134 of them is generated by SERGe rule generator and 14 mandatory rules are engineered manually.

4.1.2 Database-based Difference Representation

In this class of approaches, model differences are stored (represented) in relational databases. The database-based model difference representation approaches simply store model differences in relational databases in a way that is convenient for reusing difference information in developing additional services on the top.

SMOVER. Semantically enhanced Model Version Control System (SMOVER) [Altmanninger et al., 2007] is a sequential model version control for software models offering the conflict resolution of model differences. The approach relies on the EMF Ecore meta-models and derives the view definitions meta-model from the meta-models of modeling languages. The meta-models of modeling languages are used for representing model differences, whereas the view definitions meta-model is used for representing semantic model differences. Both, the creation of the view definitions meta-models from meta-models and creation of model revisions in view from model revisions are realized using the "Atlas Transformation Language" (ATL) [Jouault et al., 2008]. The approach detects the model differences from the representation of models and the model differences from the representations of software models, then these two kinds of model differences are mapped to each other on the conflict resolution phase.

SMOVER stores modeling artifacts and their differences in standard MS SQL databases and it creates the working copies of models in the initialization phase. The EMF-based graph-like structures of the working copies of models are created from the artifacts stored in databases using ATL transformations.

In particular, SMOVER distinguishes four different operations in order to describe structural changes in graphs that are useful for conflict detection.

- *Attribute Update:* The values of artifact attributes have been changed.
- *Reference Update:* The set of referenced modeling artifacts have been changed. For example, new modeling artifacts can be added or removed.
- *Role Update:* Modeling artifacts are referenced or de-referenced by other modeling artifacts.
- *Referenced Element Update:* The referenced modeling artifacts have been updated.

SMOVER provides sequential version control feature for software models. The system can be integrated with external modeling tools with additionally developed adapters which convert the XMI serialization of models into SMOVER internal structures and vice verse.

4.1.3 Text-based Difference Representation

In text-based approaches, model differences are represented by means of the change operations (change statements) in textual forms. This section investigates the

text-based model difference representation approaches. Some of these approaches refer to themselves as *operation-based* because they use basic change operations. However, they are represented in form of textual differences.

Alanen and Porres. An early approach in text-based difference representation was introduced by [Alanen and Porres, 2003]. In this approach, model differences are represented by the sequence of modification (transformation) operations in difference documents which consist of the sequence of change operations such as additions, removals of modeling artifacts and the changes of the attribute values of modeling artifacts. These transformation operations are described in textual forms. The authors define seven elementary transformation operations that are used as the basis for defining difference documents. The approach refers to modeling artifacts with their UUIDs (universally unique identifier).

Alanen and Porres identify the following two groups of transformation operations in difference documents:

– The approach provides creation operations for creating new modeling artifacts and deletion operations for deleting existing ones.
  - `New`: operations create new modeling artifacts with particular types and UUIDs. All features of newly created elements are set to their default values.
  - `Del`: operations delete existing modeling artifacts with particular types and UUIDs.

– The approach offers the following modification operations for changing the attribute values of modeling artifacts and associations among modeling artifacts.
  - `Set`: The set operations change the attribute values of particular artifacts.
  - `Insert`: The insert operations add links between the unordered features of particular artifacts and other artifacts.
  - `Remove`: The remove operations delete links between the unordered features of particular artifacts and other artifacts.
  - `InsertAt`: These operations add links between the ordered features of particular artifacts and other artifacts at particular indexes.
  - `RemoveAt`: These operations delete links between the ordered features of particular artifacts and other artifacts at particular indexes.

Additionally, each operation has a dual operation with the opposite effect which is needed to calculate the inverse of model changes. Finally, the new and delete (del) operations do not contain references to other elements simplifying the construction of algorithms that work with model differences.

DeltaEcore. DeltaEcore is a delta language generation framework [Seidl et al., 2014] and addresses to the problem of generating delta modeling languages for software product lines and software ecosystems. The approach presents a framework to derive custom delta languages for the source languages given as EMOF-based

meta-models. The approach further defines six types of standard delta operations and generates the syntax, semantics and tooling for generated delta languages including editor support, parser and interpreter. The framework uses the meta-models of source languages to derive the syntax and semantics of delta languages with concrete textual syntax for model representation.

The *DeltaEcore* approach employs two languages represented by meta-models with concrete textual syntax: (1) the common base delta language, which provides functionality common to all delta languages such as creating and referencing elements and (2) a delta dialect, which provides delta operations specific to the source language. The delta language is then generated by combination of the common base delta language with a delta dialect specific to the respective source language. The common base delta language is utilized on the meta-meta-level and used to represent the structure of the custom delta language that is to be created on the meta-level. The delta dialect represents the delta operations suitable for expressing variability for a particular source language.

On the instance level, the following six delta operations are defined as the change semantics of standard delta operations for variability modeling with EMOF-based models:

- `Set/Unset` operations modify the value of a single-valued reference. A set delta operation assigns a new value to a specified single-valued reference, whereas an unset delta operation replaces the current value with the default value for that reference as defined in the meta-model.
- `Add/Insert/Remove` operations manipulate a set of the values of many-valued references. An add operation appends a given element to the set of values and a remove operation detaches it from the set. An insert operation places the element at a certain position within the set of values, which is only sensible if the set is ordered.
- `Modify` operations are used to alter the values of an attribute.
- `Custom` operations are used to declare delta operations with user-defined domain-specific semantics that could not be expressed using the generated delta operations. This enables the creators of delta languages to utilize knowledge of the semantics of the source languages to provide specifically tailored operations, e.g., to avoid dangling references according to the constraints of the source language.

DeltaEcore provides *editor support* for the derived delta languages including syntax highlighting and auto completion. It also supports a *delta parser* to create the model representation of the textual syntax of delta languages as well as the *delta completer* that collects all models to be manipulated. The approach provides *delta sorter* which performs topological sorting of delta to establish a suitable application order for the delta modules, the delta interpreter executes delta modules and their delta operations with the help of the generated delta dialect specific interpreters and the variant derivation assembles all affected models to store them as variant of the SPL (Software Product Lines) or SECO (Software Ecosystems).

EMFStore. EMF Store framework [Helming and Koegel, 2013] is a model and data repository for the EMF-based software models [Steinberg et al., 2008]. The framework enables collaborative work of several modelers directly via centralized and peer-to-peer connections. EMF Store represents model changes by the created and deleted modeling artifacts, feature operations and composite operations. The feature operations themselves consist of the references and attribute operations. The composite operations are the compositions of other atomic operations.

The EMF Store framework provides the state-based model matching for merging model development branches after designing in the off-line mode and the run-time operation recording feature in modeling in real-time, e.g., like in the concurrent collaborative modeling scenario. The outcome of both change detection features is the package of operation-based change log. Eventually, model changes are represented in the package of change logs using aforementioned change operations and these packages consisting of the operation-based change logs are stored in the file system of server repository.

EMF Store provides features to retrieve any state of models that was ever stored on the EMF Store repository at the any point of time. The approach provides several additional sequential collaboration, repository browsing, model editing, difference merging features which are discussed in Section 4.2 in detail.

Custom Serialization Approaches. The collaborative modeling approaches emfCollab [A. Schmidt, 2018] and Dawn [Fluegge, 2009] (the sub-component of CDO - Connected Data Objects [Stepper, 2018]) provide collaborative development features for EMF models. However, they use custom serialization of changes for synchronization in concurrent collaborative modeling. For sequential collaborative modeling, Dawn utilizes standard relational databases to persist models under development.

4.1.4 Lessons Learned

This chapter has discussed the existing related approaches based on the criteria if representation by these approaches can serve as a common underlying representation and provide efficient difference representation for both sequential and concurrent collaborative modeling scenarios. Besides, there are several approaches focusing only on some aspects of these collaborative modeling, and are not discussed in this thesis. This section briefly summarizes the outcome of this overall chapter by drawing the result of literature study.

Model- and Graph-based Difference Representation.
The graph- and model-based representations of modeling deltas are more effective in case of the sequential collaboration and distributed concurrent collaboration. The modeling deltas represented by models or graphs usually consist of additional conceptual information for representing its modeling or graph concepts alongside actual change information. Because these approaches require some sort of technical spaces with data structures for model difference representation. Thus, the model-

and graph-based representations might not be as small as text-based representations. In case of concurrent collaborative modeling, difference representation has to be as small as possible to achieve higher performance (by rapid synchronization of changes) in real-time.

The model- and graph-based model difference representation approaches are very generic in a sense for further reuse if domain experts have sufficient technical knowledge about graph theories. However, this category of approaches might possibly require more implementation effort, whereas text-based representations can be executable by simply developing textual parser. These argumentation are addressed throughout this thesis.

Relational Database-based Difference Representation.
During the evolutionary life-cycle, if all difference information is stored in relational databases, the databases might become complex and large with an associated data set. Thus, the database-based representation approaches may require more implementation effort in identification and reuse of model differences.

In concurrent collaborative modeling, this category of approaches can be quite efficient if only concurrent revisions are temporarily stored in relational databases using custom serializations. In practice, this scenario imply that the only working copies of models are stored in relational databases and the model change/differences in concurrent collaboration are synchronized via the central database server.

In sequential collaborative modeling, if the delta-based representation of model differences is required, storing all differences between the subsequent revisions of models might not be more efficient than storing these differences in separate delta documents. Because, if developers want to revert the older revisions out of the base revision which is quite usual case for the most collaborative systems, this operation might become complicated. Because, collaborative system has to query its relational database for each reversion. For instance, if the working copy of a model is in revision twenty and developers want to revert the tenth revision, then a collaborative system has to query its relational database ten times to revert the model from revision twenty to revision ten.

Text-based Difference Representation.
Model difference representation using text-based forms is more likely to be small which is well-suited to the both, concurrent and sequential, collaborative modeling scenarios. The textual deltas are (1) directly executable descriptions of model changes; (2) expressive, yet unambiguous providing necessary knowledge; (3) easy to synchronize with high performance in concurrent collaborative modeling; (4) easy serialization and deserialization by textual parser.

The text-based representation of model differences is the coded form of model differences. It is very practical and provides the compact structures of differences for tool developers in further extensions and implementations. The operation-based textual representation of model differences in difference documents offer several valuable properties:

- Difference representations directly describe the executable descriptions of model differences which can easily be implemented by simple text parser.
- The textual differences are fully expressive and unambiguous, as well as necessary knowledge about each change can easily be gained.
- The textual differences in deltas consist of only change-related information without requiring additional modeling or graph content information, i.e., without relying on any technical spaces. For instance, model- or graph-based approaches require some kind of modeling content and internal model representation structure to represent differences, whereas textual deltas can simply be stored in textual files.
- Only sequence of change operations can be exchanged over the network for synchronizing model changes in the concurrent collaborative modeling scenario. It makes the real-time synchronization of model changes very fast and contributes to avoid change conflict between the parallel revisions of models. This argumentation is evaluated in Chapter 10.
- The textual representations can be declarative by making implicit any concepts or mechanisms that can be intuitively interpreted from the context.

The operation-based textual representation approach introduced by Alanen and Porres [Alanen and Porres, 2003] aims at detecting the differences and union of model revisions with the same ancestor. The approach provides a feature for differentiating and merging models in the framework of sequential model version control. However, the approach is still remaining as a research prototype.

As presented in this chapter, until now, several approaches are introduced to difference representation and build up sequential collaborative modeling scenarios on the top. But, these approaches still can be elaborated, extended and improved. Research in difference representation for collaborative MDSE is still on its early stages. As the result of this literature study, the text-based representation (in deltas) technique is found as a compatible means to representation of model differences in this thesis.

4.2 Services

In order to achieve the higher degrees of productivity, flexibility and reusability in software system development, software engineering offers *service-oriented* and *component-based* software engineering paradigms [Breivold and Larsson, 2007]. This thesis also takes advantage of these software engineering paradigms in studying related approaches in this chapter and developing its services in Chapter 6.

The definition of the term *service* is used in a very broad manner throughout several literature. But, this thesis relies heavily on a strict definition of the term *service*.

Definition 4.1. **Software Service.**
According to [Jelschen, 2014], a **service** is defined as a unit of functionality.

A service is an abstract definition of a functionality, i.e., the service defines what functionality is provided, including required inputs and expected outputs in abstract forms regardless of how the functionality is realized or what representation the in- and output parameters are used [Jelschen, 2014]. The concrete implementations of services are usually defined as *components*.

Definition 4.2. **Software Component.**
"Software components enable the practical reuse of software parts and amortization of investments over multiple applications. There are other units of reuse, such as source code libraries, design, or architectures. Therefore, to be specific, software components are the binary units of independent production, acquisition, and deployment that interact to form a functioning system" [Szyperski, 2000].

Component-based software engineering allows for increasing the reuse of individual software units. It provides weak coupling between software units exposing the clearly defined interfaces of software services. Service-oriented software engineering can be considered as an extension of component-based ideas, yet enabling integration of distributed software systems. This thesis intends to take advantage of these software engineering paradigms in inspecting services and components provided by existing related collaborative modeling approaches, as well as in developing its additional supplementary collaborative modeling services in Chapter 6.

Existing related approaches are discussed in Section 4.1 according to their model difference representation techniques. They employ various techniques for describing and storing model differences. These and other related approaches also provide additional *supplementary services* that aim at producing, using and manipulating their difference information. This section reviews these approaches based on the supplementary services they provide.

In general, the most model difference representation approaches aim at solving the particular sub-problem of or offering an essential grounds for the certain aspects of overall collaborative modeling or model history analysis. In order to build these applications, the existing approaches intend to provide additional services so that they can cover the particular problem of overall application with the respective research ideas. The collaborative modeling scenarios require several additional services and the orchestration of these services in order to accomplish the successful collaborative work on software modeling artifacts.

The list of additional services/components can also be derived from the basic principles of source code-driven collaborative development and analysis tools, i.e.,

the same terminologies and common grounds of source code-based collaboration are applicable to MDSE, as well. The main additional services are classified as follows:

Difference Calculator. This service compares two given revisions of the same base model and detect differences between them using model matching approaches [Kolovos et al., 2009]. The difference calculator receives two revisions of the same model and produces difference documents in some representation formats (cf., Section 4.1). The special form of the difference calculator service is referred to as *change recorder* or *change listener* [Hermannsdoerfer and Koegel, 2010]. The change recorder might be embedded behind modeling editors. It does not necessarily require the changed and unchanged revisions of the same model, instead it operates directly on the base model. Thereby, the change recorder listens for models in order to observe user changes. Detailed discussions are given in Section 4.2.1.

Difference Applier and Merger. Difference applier offers essential advantages to the concurrent and sequential collaborative scenarios independently. In case of concurrent collaborative modeling, collaborators work on the different copies of the same model. The changes made by collaborators are synchronized with all other collaborators. These changes are then applied to the parallel copies by the applier to propagate changes on models. The sequential collaborative modeling scenario is another main use case of the difference applier. In case of loss or damage of models, models can be reverted to older revisions by applying deltas to models. The difference applier receives a base model and several deltas, and produces the requested revision of the model. In some literature, difference application is viewed as the part of *model merging*. Model merging aims to combine several different revisions of a model into a main or base model. In this case, model merging is required to support for conflict resolution in case conflicts arise. This service is discussed in Section 4.2.2 in detail.

Synchronizer. In concurrent collaborative modeling, collaborators preserve the copies of the same central model and modify them in real-time. This scenario requires the synchronization of model changes among these copies through the central model repositories. In concurrent collaborative modeling, these copies are usually designed using the different tool instances. In this case, the changes made by collaborators are synchronized among these tool instances in order to keep all copies up-to-date with latest changes. More discussions are given in Section 4.2.3.

Model Manager. The model management feature of collaborative modeling is one of the most addressed aspects by research works. It is meant to be one of the key characteristics of any collaborative modeling infrastructure. A model manager service should provide several activities for managing the life-cycle of software models and their revisions which are discussed in Section 4.2.4.

Change Tracer. Change tracer helps to trace the model changes and gives detailed history information about these model changes. The change tracer might focus on particular modeling artifacts and detect the histories of

changes in each revision of a model. Detected change information contributes to model history analysis in order to understand the evolutionary life-cycle of models and their artifacts. The change tracer translates the technical representations of differences into human-readable formats and helps to extract necessary and useful knowledge about the ongoing evolution process. The change tracers provide to query software model repositories and visualize the results in human-understandable ways as discussed in Section 4.2.5.

There are several approaches focusing on the certain aspects of overall collaborative modeling by providing particular services from the list above. This section revisits the approaches discussed in Section 4.1 and other approaches according to the additional supplementary services they offer. Moreover, the aforementioned services are addressed by this thesis in Chapter 6.

4.2.1 Difference Calculator

Section 3.2.1 has discussed several difference calculation (i.e., delta algorithms) approaches for collaborative source code-driven/textual development. This section investigates difference calculation approaches for collaborative MDSE.

The difference calculator (Definition 4.3) calculates model differences usually in two steps: (1) model matching and (2) change/difference production. In the model matching step, the difference calculator detects the mappings between the modeling artifacts of the two compared revisions of models. In the change/difference production step, the difference calculator detects the type of changes (creations, deletions of modeling artifacts, and changes of modeling artifact attributes) between model revisions using the mappings detected in the model matching step and produces the model differences in some forms as described in Section 4.1.

Definition 4.3. **Difference Calculator.**
A *difference calculator* is used to compare two different revisions of the same modeling artifacts and compute differences between them. It calculates differences between software artifact(s) and one of its revisions [Mens, 2002].

Several model matching techniques exist that focus on detecting differences between any given two model revisions. [Kolovos et al., 2009] gives classification of model matching approaches. In particular, [Kolovos et al., 2009] distinguishes four types of matching techniques, namely *static identity-based matching, signature-based matching, similarity-based matching*, and *custom language-specific matching.* The static identity-based matching relies on persistent universally unique identifiers (UUID) attached to modeling artifacts, whereas the signature-based matching compares modeling artifacts based on the computed combination of the feature values (its signature) of the corresponding modeling artifacts. The similarity-based matching calculates the aggregated similarities between two modeling artifacts

based on their feature values. As not all feature values of artifacts are always considerable for matching, they often can be configured in terms of weights attached to the respective features. Ultimately, the custom language-specific matching allows users to define dedicated match rules considering the actual semantics of the respective modeling languages for matching.

Below, several existing model difference calculation approaches are discussed. The difference representation approaches discussed in Section 4.1 provide the difference calculation services which are discussed, below.

Alanen and Porres. [Alanen and Porres, 2003] employs a simple model matching technique using universally unique identifiers (UUID) for all modeling artifacts. The approach differentiates the following two phases of difference calculation.

- *Mappings.* This phase creates mappings between the elements in the old and new revisions of models. The unique identifiers of modeling artifacts in both revisions are used to build mappings between these two revisions. Eventually, the created map helps to detect three types of changes; creation, deletion and change.
- *Calculations.* This phase discovers creation, deletion and change operations. As the result of this phase, difference documents are produced consisting of operations to create and delete modeling artifacts, as well as changing the values of attributes.

The complete difference documents between two model revisions are then specified by the sequence of operations as described in Section 4.1. This is the sequence of artifact creations, deletions and feature modifications.

EMF Compare. Another model comparison algorithm *EMF Compare* [EMFCompare, 2017] is introduced to detect changes in EMF models by comparing model revisions based on *structural matching*. The approach proposes the total comparison of the attributes of modeling artifacts. The approach considers that if all attributes of the two revisions of a modeling artifact are near, then these artifacts are near. The EMF compare approach detects the following six types of model changes:

- Modeling artifact differences:
  - *Creations:* new modeling artifacts are created;
  - *Deletions:* existing modeling artifacts are deleted;
  - *Order:* the order of modeling artifacts are changed.
- Attribute differences:
  - *Set:* an attribute value is changed;
  - *Add:* an attribute value is added;
  - *Remove:* an attribute value is removed.

The EMF compare tool undergoes the following steps in the process of comparing two given models in any order:

- *Comparing attributes.* The comparison of attributes itself consists of several sub-steps:
  - The computation of the distance between attributes – they define the distance between attributes as the difference between the values of the two attributes. The distance is a metric which value is comprised between 0 (attributes have the same value) and 1 (attributes have totally different values);
  - Comparing the values of boolean attributes – the values have an identical value, the distance is 0 else it is 1;
  - Comparing character strings – the algorithm returns the number of manipulations for transforming one of the character strings to the other, where a manipulation is one of those operations: deleting a character, adding a character, modifying a character;
  - Comparing numbers – numbers have a very large sense.
- *Attribute weights.* An attribute has a weight depending on the volume of information it carries. It can be computed by two factors: (1) the number of times the attribute is met in the model; the more – an attribute is present, the less – it is important; (2) the number of values – an attribute takes; if it has only two possible values, like a Boolean, it is very unlikely to be a good identifier.
- *Comparing two elements.* The difference rate in this algorithm is the value of the difference between two elements. If the distance is less than an arbitrary value, the elements are considered to be a potential couple.
- *Local weights.* The attribute weights are constricted between 0 and 1 by dividing their value by the sum of the weight values for the attributes present on two elements.

In EMF compare, models are compared recursively. Modeling artifacts for every level are compared with every artifact for their level. Once artifacts are compared, they are ordered depending on their distance in an increasing order. Couples are then computed starting with artifacts with the shortest distance. Potential couples which contain at least one artifact are deleted. The remaining single artifacts are added to the list of couples by specifying that difference between the two artifacts is addition or deletion. The type of operations depends on if modeling artifact is in the local model or in the compared model.

AMOR. The adaptable model version control system (AMOR) [Taentzer et al., 2012], [Brosch et al., 2010] takes advantage of the EMF compare approach for comparing differentiated model revisions and detect differences between them. AMOR employs the *UUID-based* and *structural* model matching features of EMF compare for detecting mappings (i.e., match models) between the original and revised revisions of models. Implementation of AMOR is partially based on EMF compare so that EMF models can be matched using either heuristics or UUIDs. However,

AMOR improves the model matching algorithm of EMF compare by the specific implementations. The model differences produced by EMF compare are optimized and translated into the own model-based representation of AMOR. The difference meta-models of EMF compare and the extended difference models of AMOR are similar, but some explicit information and the several types of conflicts are not supported by EMF compare in case of AMOR. Eventually, they develop extra implementation for detecting conflicts based on two difference models conforming to their extended difference meta-model.

SiDiff. One of the most popular and generic approaches is the *SiDiff* approach [Treude et al., 2007] which is utilized by several other model comparison approaches, as well as in this thesis. SiDiff is a framework for comparing and detecting model differences. It is modeling language generic, configurable and provides several model matching algorithms. It uses the graph-like structures to represent software models internally and exploits graph isomorphism algorithms to discover differences between differentiated graphs. The SiDiff approach can be used for many modeling languages if their models can be represented as graphs and accessible by the SiDiff kernel. This approach provides three main matching strategies, i.e., *ID-based, signature-based* and *similarity-based* using name and structural similarity metrics.

SiDiff processes two input models that are given in a textual or binary format which is either complying to XMI (XML Metadata Interchange) [MOF, 2003] or a proprietary format. It supports a parser to transform these models into internal representation format which is typed, attributed and directed graphs. Every node corresponds to a modeling artifact in the original model and edges represent existing connections between those artifacts. Every node in the graph is additionally connected to a type that represents its artifact type and may have a set of attributes consisting of the pairs of keys and values. Likewise edges are connected to the type representing connections in models and have boolean attributes whether connections are the references or nesting edges of model.

In order to decrease the computational cost on finding corresponding elements from the two revisions of a model, a special high-dimensional search tree, the S3V (Similarity searches parse vector) tree, is used to store modeling artifacts in memory. To find possible matching candidates from another model, all of the modeling artifacts usually must be checked for similarity. Instead of comparing all artifacts, they are stored in the S3V tree according to their similarity to each other. Every artifact is therefore interpreted as a numerical vector where every index represent the certain characteristic of artifact. The similarity of two artifacts is then defined as the euclidean distance with smaller distances resulting in higher similarity. The tree can then be used to find a set of the most similar artifacts of a given artifact by doing a range query on the tree which returns all similar candidate artifacts within a subspace of the tree defined by the specified range or distance of the query. One S3V tree is created for each compared models and for each occurring artifact type before beginning comparison and these trees remain in memory while comparing [Treude et al., 2007].

The SiDiff algorithm firstly detects correspondences between compared model elements by calculating similarities between these elements. The similarity between two elements is given as a float value between 0 (no similarity) and 1 (equality). Similarity is usually determined by local attributes or elements in the near proximity of the investigated elements. SiDiff provides a set of standard functions for such similarity measurements. However, it allows for adding new similarity functions if required.

Whilst only corresponding matched artifacts have been identified, changes to artifacts, additions and removals still have to be detected and stored for later reuse. This is done within the unified document which contains relevant information about difference calculation. The unified document contains all artifacts of both compared models and difference information. Already matched artifacts are however only listed once in the document. SiDiff classifies differences into four categories:

- *Structural differences:* artifacts that have been added or deleted;
- *Attribute differences:* artifacts that are corresponding but that have changed attributes;
- *Reference differences:* artifacts that are corresponding but that have changed references;
- *Move differences:* artifacts that are corresponding but that have a changed parent artifact.

SiDiff provides similarity computation flexibility to define a set of similarity functions which allow customization of the calculation process on a more detailed level. However, it is necessary to define such sets for every processed model type separately according to the available artifacts. Transformation of the processed models into a directed graph provides the single algorithmic approach. Thereby, any model type can be processed without modifying the algorithm. Transformation of the processed models into the internal graph structure must still be defined.

Concluding the SiDiff approach, it is highly configurable even it is time-consuming in case of the large-scale graphs. Hence, the approach is much more productive if the given models consist only of several hundreds of artifacts. Even though, the approach is investigated in several model comparison and difference calculation approaches. The approach in this thesis takes advantage of the difference calculation feature of the SiDiff algorithm to develop its difference calculator service in Section 6.3.

SMoVer. The *SMOVER* (Semantically enhanced Model Version Control System) [Altmanninger et al., 2007] is dedicated to the sequential model version control of EMF-based software models relying on the Ecore meta-models. In order to detect conflicts between the two concurrent revisions of models, the approach takes advantage of the EMF Compare model comparison approach based on a *graph-based structural difference computation* between the model revisions. Actual comparison of modeling artifacts is based on persistent unique identifiers designated in meta-models.

EMFStore. EMF Store framework [Helming and Koegel, 2013] addresses to the problem of sequential collaborative modeling. It uses the *state-based model matching* technique for comparing models in order to merge the various development branches into the main repository. This case usually considered in the off-line model designing mode. But, the approach also provides the *change-recorder* service for recording changes on models.

UMLDiff. UMLDiff [Xing and Stroulia, 2005b] is another difference computation approach that specifically tailored to UML modeling language. The approach computes similarity metrics based on the names of modeling artifacts and the structure of models. UMLDiff applies the string-based matching technique at the element level, as well as the graph-based matching at the structure level, and internally combines the obtained similarity measures. These features eventually makes UMLDiff a hybrid matching approach.

Nejati et. al. Another approach particularly dedicated to model difference computation is introduced in [Nejati et al., 2007]. The approach is used to match only UML state machines. Its matching is based on static similarity measures such as the typographic, linguistic, and depth properties of modeling artifacts, as well as behavioral similarity measures.

DSMDiff. DSMDiff [Lin et al., 2007] is model matching approach for domain-specific modeling languages. The approach compares modeling artifacts based on their computed signature and considers the relationships between modeling artifacts previously matched by signatures.

To summarize, DSMDiff and EMF Compare aim at obtaining optimal results and mostly dedicated to particular modeling language relying on the specific type of meta-model. In contrast to these modeling language-specific approaches, SiDiff provides an adaptable model comparison framework, which can be configured for specific modeling languages. In addition to these, there are diverse approaches for model matching and difference calculation. Among these approaches, the generic and configurable algorithm SiDiff is considered to be the most reasonable fit to difference calculation in this thesis. Thus, this thesis takes advantage of the SiDiff algorithm for implementing the *state-based comparison* feature of its difference calculator service. The detailed discussion of the rest of existing difference calculation approaches is out of the scope in this thesis.

4.2.2 Difference Applier and Merger

Difference appliers perform the process of difference application which refers to manipulation of a model according to the set of predefined change operations. The term *difference application* is also referred to as *patching* in some literature and text-based version control systems [Roundy, 2009], [MacKenzie et al., 2003]. It considered to be the special case of general software revision *merging* scenario [Bailor et al., 2011], [MacKenzie et al., 2003], [Brunet et al., 2006].

Definition 4.4. **Difference Application.**
Difference application is the process of transforming software systems from one state (version) to another by creating new artifacts, removing existing ones or changing the properties of existing artifacts.

There are several use cases of the difference applier service:

- *Reverting.* In concurrent or sequential collaborative modeling systems, software models (or any software system) usually have several revisions starting from the initial development till its current state (revision). Thereby, collaborators feel a need for reverting their models to earlier revisions in case of the damage or loss of data. In this case, several difference documents have to be applied to the base revision of models in order to obtain the intended revision of the same model.
- *Merging.* Software projects under evolution and maintenance usually have several development branches manipulated by collaborators using sequential model version control tools. Collaborators always have to keep their branches up to date with the main development trunk to obtain changes made by other collaborators. Likewise, each collaborator has to commit his/her local changes to the main trunk. Updating and committing operations are eventually application of model differences. But, it usually invokes the model matching in order to detect model differences between two revisions. The difference application scenario in sequential collaborative modeling is also referred to as *difference merging* in several literature. The difference merging itself is largely discussed research subject alongside the problem of *conflict resolution* by several literature. Because, difference conflicts are more likely to occur in software model evolution with several development branches and raises the problem of conflict resolution. Merging revisions or differences is required mostly in the sequential collaborative modeling scenario. The related approaches focusing particularly on this kind of difference application are discussed in this section.
- *Conflict Resolution.* In merging different revisions or differences, change conflicts might arise. These conflicts should be somehow resolved either *automatically* if possible or with human-interaction [Altmanninger et al., 2009]. Change conflicts can not be fully resolved automatically but sometimes requires human interaction which is referred to as *semi-automated* merging. The most merge techniques perform possible merges. In unsolvable cases, they switch to interaction mode with collaborators for providing an interactive resolver feature. The unsolvable conflicts can then be presented to collaborators in form of the recommendations by browsing conflicts [Koegel et al., 2010].

Below, this section investigates existing difference application services provided by the difference representation approaches discussed in Section 4.1 and other approaches. There are several approaches addressing the model difference application

service related to their conflict resolution features. As long as this thesis does not directly addresses the problem of conflict resolution, this section does not provide the detailed discussions on the conflict resolution of the covered approaches.

Alanen and Porres. [Alanen and Porres, 2003] offers a merge algorithm for the sequential collaborative modeling scenario using their operation-based difference representation techniques as discussed in Section 4.1. Their merge algorithm is basically used to apply the transformations contained in difference documents to models and obtain the new revisions of these models, eventually. Since the difference representation uses the operation-based approach, the difference information is directly the transformation (executable) descriptions of model differences. The approach addresses the model elements by their UUIDs. The difference application process is performed in the following order:

- `new(e,t)`: creates a new modeling artifact of type t with the UUID of e.
- `change(o)`: makes the feature change o; a change operation might be one of the change operations defined in Section 4.1.
- `del(e,t)`: deletes a modeling artifact of type t with the UUID of e.

By ordering difference operations in this way, creation operations are executed firstly, change operations secondly, and deletion operations in the end.

Cicchetti et al. In [Cicchetti et al., 2007] and [Cicchetti, 2008], *Cicchetti et al.* uses the *model-based* difference representation approach as stated in Section 4.1, i.e., model differences are represented using *difference models*. Additionally, the approach provides the difference application and conflict resolution features. The approach addresses applying difference models to the differentiated models and obtain other revisions. It further offers the merging techniques. Difference application involves several sub-operations to be performed. These are *model matching*, *weaving* and *merging*. To apply difference models to the base models, difference model and base model are matched for detecting references between modeling artifacts. These references are designated to detect which change should be applied to which modeling artifact. References are stored in a *weaving model* that is utilized while applying difference models to base models. The approach considers the two ways of difference application: (1) forward application, i.e., application of difference models to given models results in newer revisions, (2) backward application, i.e., application of difference models to given models results in older revisions.

According to Cicchetti et. al., conflicts usually arise during the difference application process. For instance, a modeling artifact might be deleted from model by a collaborator, whereas the same artifact is changed in difference model. In this case, a `delete/change` conflict occurs. Thus, the approach focuses on handling conflicts in advance and resolve by the combination of atomic operations. The authors have elaborated a list of conflict types which are derived from operation combinations.

AMOR. The Adaptable Model Versioning System (AMOR) introduced in [Brosch et al., 2012] is a modeling tool independent approach which works with the state-based model differences. The major focus of AMOR is model differentiation and

model merging. AMOR does model matching in order to detect difference models including atomic and composite operations between compared models. Then, it produces a conflict model after detecting conflicts based on difference models. It detects inconsistencies and annotates them before passing conflict models to the conflict resolution step.

The work-flow of the AMOR approach is as follows:

- *Operation Definition.* AMOR provides modeling language-generic sequential model version control support. The quality of conflict detection and resolution may be considerably improved when language-specific knowledge is incorporated in the merge process. Therefore, AMOR provides an extension point to integrate composite, user-defined operations like refactorings. Once this is done, application of such refactorings is detectable. This additional information is the basis for a precise conflict detection.
- *Conflict Detection.* The conflict detection component detects not only the generic atomic changes like insert, update, delete, but also composite operations stored in operations repositories.
- *Resolution Lookup.* In this step, the Resolution Recommender of AMOR checks whether there is solution for the reported conflict in the Resolution Pattern Storage.
- *Conflict Resolution.* Collaborators have to decide how to resolve conflicts. Collaborators may either resolve the conflict completely manually or choose one of the recommendations made by the Resolution Recommender.
- *Resolution Reasoning.* In the resolution phase, the Resolution Reasoner analyzes collaborators decisions in order to derive general resolution patterns for conflicts between the two conflicting operations. The patterns are stored in the Resolution Pattern Storage for application in similar situations in future.

- *Merge.* Finally, all previously chosen resolution recommendations are applied and the resulting model is saved into repository as a new revision.

AMOR provides different mechanisms to support the conflict resolution:

- *Semi-Automatic vs. Manual Conflict Resolution.* In semi-automatic conflict resolution users select one of suggested recommendations offered by the Resolution Recommender. The resolution rule of the resolution recommendation is then applied to the maximally merged revision. In certain cases, user input is required, e.g., if the name of an element has to be introduced. If none of the given suggestions fit to user's need, then it is still possible to resolve the conflict manually either alone or in collaboration with others.
- *Collaborative vs. Single User Conflict Resolution.* If no adequate recommendations are found, then semi-automatic resolution is not possible. Therefore, AMOR offers a validation of the merged revision, on the one hand, and an opportunity to resolve conflicts in a collaborative way. In this context, collaboration refers to communicating the intentions behind changes and trying to

combine these intentions in a new revision. Thus, AMOR provides an extension called Collaborative Conflict Resolver to overcome the aforementioned challenges by orchestrating collaborators while resolving conflicts. The Collaborative Conflict Resolver offers a communication platform to exchange intentions behind their conflicting changes. Collaborators may manually create the merged revision together by partly remodeling a scenario to obtain the final model with high quality. Afterwards, both collaborators have to accept the new revision before committing it to the central model repository. This ensures that an approved and consolidated revision is checked in. To sum up, the Collaborative Conflict Resolver allows for distributing responsibility in this critical and error-prone merge phase.

EMF Store. The EMFStore framework [Helming and Koegel, 2013] can be used in *off-line* (i.e., without being connected to the server/repository) and *on-line* (i.e., being connected to the server/repository) collaboration modes, as well as it provides the storage of the model histories. Thus, the approach provides synchronization of model changes in form of the change packages consisting of operation-based change logs. In model designing in off-line mode, model revisions are compared using the state-based comparison technique in order to detect whether there are conflicts between main development trunk and its branches. If there are conflicts in model matching, they are resolved by the semi-automated conflict resolution. In EMF Store, any revision of models can be reverted by applying the set of change packages to the base model.

EMF Diff/Merge. The EMF diff/merge framework [Constant, 2012] offers a feature for differentiating and merging EMF-based software models. This approach targets Ecore-based meta-models. The approach is capable of comparing and merging EMF models in semi-automated ways. It provides a graphical interface highlighting model changes in conflict of compared models. This enables users to manually merge these conflicts.

Application of differences or changes to models under development is one of the important and fundamental activities in collaborative modeling. It contributes to usability of difference information by propagating changes represented in difference documents on models under development and evolution. As discussed above, realization of the difference applier service depends heavily on how model differences are calculated and represented in difference documents. As long as difference applier performs what is defined in difference documents, the representation technique of differences has to provide difference information about operations it has to perform (e.g., create, delete, or change), references to modeling artifacts that operations have to be applied to, and other necessary information.

4.2.3 Synchronization

In concurrent collaborative modeling, collaborators preserve the copies of the same central model and modify them in real-time. This scenario requires the synchronization of model changes among these instances through the central model repository. In real-time collaborative modeling, these instances are usually designed using the different tool instances and change synchronization should occur among these tool instances.

> ***Definition*** 4.5. **Synchronization.**
> *Synchronization* is constantly keeping all connected collaborators up-to-date with instant changes made by any of these collaborators.

EMFCollab. EMFCollab [A. Schmidt, 2018] is an EMF-based collaborative modeling approach to allow multiple collaborators to edit shared EMF models, concurrently. The approach utilizes a client-server architecture. The client can be integrated into EMF-based Eclipse editors. The *emfCollab* approach offers a client-server model to let the collaborators for editing the shared models at the same time. It stores the master copies of the models on the server side and the slave copies of the master models on the client side. These models are *synchronized* over the network, by serializing and distributing the commands affecting the model. The communication layer it uses is *CoolRMI* [QGears, 2018]. *Cool RMI* makes the implementation of client-server application prototypes easy using pure TCP on the transport layer.

EMFStore. EMFStore [Koegel et al., 2010] provides a repository for storing, distributing and collaborating on EMF-based entities. Its server is a headless application that is usually run as a daemon or service on physical server machines or virtual machines. Its clients are typically embedded in existing applications that relies on *EMFStore* to store entities and *synchronize* its entities among the different collaborators. In *EMFStore*, changes on shared or checked-out projects can be synchronized with *commit* and *update* operations. However, it is not explicitly documented if these changes can be synchronized in real-time.

Dawn. Dawn [Fluegge, 2009] is a sub-component of Connected Data Objects (CDO) project [Stepper, 2018] and achieves to create collaborative network solutions for user interfaces basing on CDO. It uses the *Net4j Signalling* platform which is an extensible client/server communications framework.

Web-based Tools. There are also several web-based modeling tools like GenMyModel [GenMyModel, 2015], Creately [Cinergix Pty., 2015] which exchange changes over WebSockets. As long as they are not open-source approaches, their core ideas and underlying implementation technologies are not explicitly documented. Their modeling notations and other services are not accessible making them difficult to study and extend.

Generally, there are several synchronization services that provide synchronization of changes in concurrent collaboration. In most cases, synchronization of changes among several collaborators is accomplished by defining custom serialization of model changes. Moreover, the existing approaches take advantage of different techniques of realizing their change synchronization.

4.2.4 Model Manager

The model manager feature of collaborative modeling is one of the most addressed aspects by research works. Franzago et. al. [Franzago et al., 2018] defines model management as one of the key characteristics of any collaborative modeling infrastructure. A model manager may consist of a model repository and several activities for managing the life-cycle of software models; such as managing the persistence of models and their related meta-data, support for creating, editing, or deleting models and their revisions [Franzago et al., 2018].

Definition 4.6. **Model Manager.**
A model manager for managing the life-cycle of software models may contain a (possibly distributed) *repository* for managing the persistence of software models, their revisions, and their related meta-data, a *modeling tool* for creating, editing, or deleting software models and their revisions [Franzago et al., 2018].

EMFStore. In EMFStore [Koegel et al., 2010], modeling projects can be shared with the server by a client and *checked out* from the server to other clients. Changes on shared or checked-out projects can be synchronized with *commit* and *update* operations. Commit operations push local changes made by collaborators to the server while update operations pull changes made by other collaborators from the server to the client. EMFStore can store and manage models and their revisions.

SMoVer. The *SMOVER* (Semantically enhanced Model Version Control System) [Altmanninger et al., 2007] provides several model management activities such as *add* – to bring new models under version control, *checkout* – to obtain the copies of models in the central repository into the local working spaces, *commit* – to commit local changes to the central repositories and *update* – to update the local working spaces with the latest changes in the central repository. SMOVER mostly addresses flexible difference merging technique and uses *relational databases* to store the model differences. Hence, the approach requires an adapter for parsing models between external modeling tools and SMOVER.

4.2.5 Change Tracer

Model differences stored in model repositories must be reusable and traceable in further analysis of the evolutionary life-cycle of software models under development, evolution and maintenance. Collaborators can then extract necessary information about the change histories and different states of their models focusing on a particular artifact or state. Eventually, the histories of model changes and states of modeling artifacts can be traced and collaborators can gain knowledge about the history of their models in order to make further useful decisions about the future of their models.

According to Definition 3.5 of mining software repositories, the change tracing scenario is the part of overall repository mining.

> ***Definition*** 4.7. **Change Tracer.**
> Change tracing has been coined to describe a broad class of investigations into the examination of software repositories. Here, software repositories refer to modeling delta repositories representing modeling artifacts that are modified, produced and archived during model evolution [Kagdi et al., 2007].

In general, the problem of model history analysis can be leveled into two common objectives: (1) *extraction* of difference information from model repositories and (2) *browsing/visualization* of extracted information. The existing related approaches discussed below consider these two main objectives of model history analysis.

Wenzel et al. A solid approach addressing the problem of model change tracing and history analysis is introduced in [Wenzel and Kelter, 2008] and [Wenzel, 2010]. In the first stage, the approach creates the history repository from the model revisions repository. In the history creation phase, it extracts revision information and one graph representation for each model revision in the repository. It then stores traceability and evolution data by adding fine-grained element information to revision information and graph representations. With this step, the approach creates mappings between the model repository and the history repository.

The approach creates the mappings of persistent identifiers to store connections between original modeling artifacts in the repository and their revised artifacts. The persistent identifiers stored in that map are also assigned to the original modeling artifacts and their revised counterparts. This is referred to as traceability information [Meier and Winter, 2018]. Traceability information serves to identify modeling artifacts across its evolution and to follow the artifacts of model revisions to the corresponding artifacts in the ancestor revisions, and in the descendant revisions respectively, if the element exists in that revision of the model. Traceability information can be calculated incrementally whenever the new revisions of models are created.

Furthermore, the approach introduces identification links connecting the new revisions and the corresponding elements in the ancestor revisions. These identification

links are added as the identity of the respective versioned artifacts. The identification links are computed by the model comparison approach of SiDiff algorithm [Treude et al., 2007].

Due to the EMF-based internal graph representations, the approach distinguishes four types of changes: attribute change, reference change, moves, structural changes that denote inserted or deleted artifacts. It stores the artifact changes as objects assigned to the identification links that represent the correspondences of the changed artifacts. It does not store information about structural changes and it is implicitly given by the absence of identification links. The inserted artifacts have no incoming identification links and the deleted artifacts have no outgoing links respectively.

The approach provides a browsing feature of detected history information by the change tracer. Model revisions can be opened as textual tree representation with multiple revisions. User can trace the occurrences of the selected modeling artifacts. The traced elements and their occurrences are visualized in different colors. In the visualization component, modeling artifacts and correspondences among them are visualized in different colors and notations.

EMF Store. The EMFStore framework [Helming and Koegel, 2013] supports extensible default user interfaces for almost all its available functionality. By integrating the EMF-based client platform, clients connected the server can have a navigator, history view and repository browser. The repository browser (integrated from EMF Client Platform) enables the users to connect their client to the server and log in. The navigator view provides viewing all modeling entities in the client's local workspace.

Moreover, EMF Store provides several visual dialog to visualize synchronization with the server, to share and check out entities, as well as to commit and update entities. It offers an interactive merge dialog for resolving the change conflicts. Finally, the history of modeling projects or selected entities can be viewed in the history view including branches, tags and the changes between revisions.

Hawk. Hawk [García-Domínguez et al., 2018] is an indexing approach for fast querying over the fragment collections of software model repositories. It provides features for querying through work-spaces, indexing models under version control systems and creating NoSQL databases from these work-spaces. Eventually, it can query through NoSQL databases and return the results of queries. Hawk can run as Eclipse plug-in, Java library, or network service. It can work over various types of locations, e.g., version control systems (SVN/Git repositories) or file stores (local folders, Eclipse work-spaces, HTTP locations). The approach is still under development and extension. It is planned to extend its user interfaces, more back-ends, better Git connector (JGit-based) and visualizations based on time-aware queries.

MDSE provides a visual designing aid for software development and evolution activities by software models. Thus, analyzing the histories of evolving software models is quite crucial to understand and comprehend the evolutionary life-cycle

of software models. The change tracer, in turns, is usually viewed as the main underlying component for mining model repositories. The combination of repository mining and browsing forms the evolution history analysis. The current state of research in the field of evolution history analysis for MDSE demonstrates a strong need for research on advanced mining and change tracing approaches for model repositories.

4.2.6 Lessons Learned

Section 4.1 has reviewed the existing difference representation techniques for collaborative MDSE. Section 4.2 has investigated these and other approaches according to the supplementary services and components they support. This section discusses these approaches in order to highlight open challenges and missing features, as well as to indicate what can be improved in the research field.

As the result of literature study in Section 4.1 and Section 4.2, there are many approaches addressing the certain aspects of overall research challenge. However, these is still a strong need for extended research in difference representation for collaborative MDSE combining concurrent, sequential collaborative modeling scenarios and model history analysis.

A difference representation technique should not only be abstract, but practically useful for further tool developers. A difference representation approach has to provide several technical properties being executable, implementable in efficient ways, fully expressive, yet unambiguous, for transforming existing models to new models, as well as facilitate developer productivity with precise, concise and clear descriptions. If a difference representation approach provides aforementioned properties, it can then be applicable, extensible and further reusable.

This section gives a brief overview of *lessons learned* and *open issues* in the existing related approaches. Table 4.1 lists the related approaches discussed in the previous sections based on their representation techniques and services they provide.

The most approaches, listed in Table 4.1, aim at resolving challenges in the context of independent services or tools. Only few of current approaches aim at being generic, where they are surrounded or associated with other modeling tools or technical spaces. Apart from the list of reviewed approaches, there are several related approaches focusing only on the particular aspects of the overall model change representation and its use cases. Some of the existing approaches still remain as research prototypes.

This thesis intends to deliver a difference representation approach with several effective properties such as the usability, expressiveness, productivity, completeness, extend-ability, etc. Hence, a change representation approach has to be compact, enabling tool developers for developing further collaborative modeling tools on the top. Additionally, the technical implications play an essential role in difference representation in collaborative MDSE. Model differences/changes have to be

Approach	Representation	Services
Alanen and Porres [Alanen and Porres, 2003]	textual operations	calculator, applier (merge, conflict resolution)
DeltaEcore [Seidl et al., 2014]	textual operations	operation recorder, applier
EMF Store [Helming and Koegel, 2013]	textual operations	model management, calculator, applier (merging, conflict resolution), model editor, change tracer
Cicchetti et. al. [Cicchetti et al., 2007]	model-based	applier (merging, conflict resolution)
AMOR [Brosch et al., 2010]	model-based	calculator, applier (merging, conflict resolution)
SiDiff [Schmidt and Gloetzner, 2008]	graph-based, model-based	calculator, SiLift – semantic lifting, SERGi – rule generator
SMOVER [Altmanninger et al., 2007]	database-based	model management, calculator, applier, adapter
Wenzel [Wenzel and Kelter, 2008]	database, model-based	Evolution analysis – change tracer
Hawk [García-Domínguez et al., 2018]	database	query engine – change tracer

Table 4.1: Classification of Related Approaches

represented in forms that can be reused by other tools enabling the high-levels of interoperability.

As the result of discussions, it can be pointed out that more research work is necessary on generic *model difference/change representation* and its *supplementary services*. A novel means for model change representation and services have to consider MDSE concepts, their complexity and associated data structures. Moreover, the several existing approaches still remain as only research prototypes that result in the lake of implementation of prototypes in many cases. Hence, generic difference/change representation is required which allows for developing the chain of services and supports change synchronization among them. In turns, difference representation has to support the concurrent, sequential collaborative modeling scenarios and history analysis through centralized servers enabling *single point of truth*.

4.3 Requirements

After reviewing and analyzing several existing difference representation approaches in Chapter 4, as well as highlighting and discussing open research challenges, it can

be pointed out that the research in model difference representation and its applications is still in its infancy. To this end, this thesis addresses the problem of *model difference representation* as its primary research question. Furthermore, the thesis aims at providing a *catalog of supplementary services* which can produce, use and manipulate difference documents represented by a proposed difference representation approach. Eventually, concurrent and sequential collaborative modeling, as well as history analysis use cases of collaborative MDSE will be elaborated by the specific orchestrations of provided services.

A difference representation approach for software models has to satisfy a number of requirements regarding its re-usability, applicability and extendability. Difference information must be small, compact, complete, conveniently reusable, extensible and applicable. In order to have a solid difference representation approach in the end, several research approaches [Cicchetti et al., 2007], [Herrmannsdoerfer and Koegel, 2010], [Sriplakich et al., 2008] specify requirements for a model difference representation. In the same vein, this thesis also defines a list of requirements and properties following concrete principles and extending requirements stated by [Cicchetti et al., 2007], [Herrmannsdoerfer and Koegel, 2010] and [Sriplakich et al., 2008].

Below, these requirements are listed targeting efficiency, re-usability, applicability and extend-ability of model difference representations. Initially, the list of requirements is described that the overall approach has to satisfy. Thereafter, the list of requirements is given for the difference document, i.e., modeling delta and a difference representation (i.e., *Difference Language*) and its operations.

- **RQ1: Meta-model Generic.** There are several modeling languages following diverse formal specifications and concepts. The abstract syntax of modeling languages, i.e., the modeling concepts are defined by their corresponding meta-models. Models conforming to that meta-models are the subject to continuous changes and evolution. Thus, being generic with respect to meta-models makes a difference representation approach applicable to many modeling languages. A difference language should not relying on a particular domain-specific modeling language, enabling applicability in various domains. A generic difference language can then be tailored to the use of specific domain-specific modeling languages even more supporting the user in defining the models in easy ways. So that users may work with concepts they know and they only have to learn the limited set of concepts related to the domain they work.
- **RQ2: Modeling Tool Generic.** There are several modeling tools as well and they have own internal model representation techniques, i.e., concrete syntax (e.g., Sirius-based [Viyović et al., 2014] tools like UML Designer [Obeo Network, 2017], etc.). To be able to handle models designed in different modeling tools, a difference language must not rely on a specific model designing tool restricting itself to that tool. This feature of a difference language provides its integration and interoperability with existing CASE tools.
- **RQ3: Extensible.** In addition to being meta-model and tool generic, a proposed difference language must be flexible, i.e., it has to be open for further

improvements, extensions, adaptations and integration. Particularly, the provided services by an approach have to be available for further improvements, enrichment and replacements. For instance, the tool developers might intend to extend or improve existing services, or replace the existing services by other services. Eventually, further services, components and plug-ins can easily be developed on the top of a proposed difference language. This requirement is one of the most significant properties a difference language has to provide, which is not sufficiently addressed by the existing approaches.

- **RQ4: Operation-based.** The model differences are the collection of the changes of modeling concepts. Artifact changes can be viewed as operations using the simple sequence of edit operations (cf. Section 4.1). Using an operation for each change results in a low number of construction operations with simple syntax and enables effortless implementations, as well as follows actual modeling concepts. The operation-based difference language should also be completely independent from the underlying implementation techniques and technologies, i.e., technical spaces. Each change can slightly be represented by single line of operation. Change operation have to be sufficiently accurate and detailed for particular automatic transformations.
- **RQ5: Model Reference.** In order to refer to modeling artifacts from the change operations of a difference language, each change operation has to embody a reference to a modeling artifact which has to be edited by an initial operation. Such kind of references are essential grounds for applying changes to models and provide persistence of modeling artifacts and changes. It is the best aid for mining consistent information (i.e., change tracing in Section 4.2.5) for model history analysis (as discussed in Section 3.3).
- **RQ6: Expressive.** The syntax of the change operations of a difference language must be meaningful so that they can be easily understood by users or tool developers. The understandable syntax of change operations is quite easy to express changes in human-readable formats and is practical to implement by various technologies. Eventually, unlike model-based, graph-based or database-based approaches, the syntax of the operations of a difference language can be easily understood and extended by tool developers.
- **RQ7: Executable.** The change operations of a difference language have to form the executable descriptions of model differences so that it can then be applied to its differentiated models in order to transform the model from one revision to another. Difference documents have to represent directly the executable descriptions of model differences or at least they have to be easy to implement by model transformation or manipulation approaches. Applicability of difference documents allows for reverting the older or newer revisions of the working copy of a model. By being applicable, difference documents enable easy change propagation in case of the concurrent collaborative modeling scenario, as well.
- **RQ8: Delta-based.** Only changed modeling artifacts have to be referred (excluding unchanged artifacts) in difference documents. Difference documents should consist of a set of operations which refer to the only changed modeling concepts, i.e., a difference document covers only necessary set of data

about each artifact change. As long as a difference document consists of references and operations for only changed modeling artifacts, this requirement is also entitled as *minimalistic* by the some other related approaches [Cicchetti et al., 2007], [Herrmannsdoerfer and Koegel, 2010]. The smallness of difference documents brings a huge amount of advantages in terms of time and storage memory in sequential collaboration, as well as in the synchronization performance of model changes over network in concurrent collaborative modeling.

- **RQ9: Persistent.** The model differences in difference documents have to be persistent during the evolutionary life-cycle of models. Each difference document must consist of persistent change information with respect to its successor and predecessor elements. It allows for maintaining and persisting each modeling artifact together with the change history. The persistence of difference documents provides consistency of changes and modeling artifacts from the beginning to the end of the life-time of each modeling artifact. If change operations are persistent, they are traceable and applicable as well, i.e., the change histories of modeling artifacts can easily be traced for extracting necessary knowledge for further analysis.
- **RQ10: Traceable.** The model differences represented in difference documents have to be available for further reuse and exploitation. Only representing the model differences without being accessible and re-usable is ineffective and needless. Thus, difference representation must be straightforward and accessible for further analysis and manipulations enabling applicability to various application areas so that the change and model histories can be analyzed by mining necessary information from difference documents. A difference language should not rely on underlying implementation technologies and provide easy access to difference information by the change tracer service (Section 4.2.5) in model history analysis.
- **RQ11: Relevance.** The representation by a difference language must embed precise information about each change including the kind of change, the reference to modeling concept and modeling concept which has to be changed. The kind of change defines what kind of change is made (e.g., creation or deletion of a modeling artifact or change of attribute values), whereas the referenced modeling concept defines the changed modeling artifact. These operations have to allow for representing all model changes embodying all necessary information about each independent change. An operation must be a statement that is correct and relevant to a change. Its relevance and correctness can be checked against the meta-model of a modeling language.

These requirements are proper characteristics for emerging an appropriate approach for a model difference representation language for collaborative MDSE, as well as in order to make the data representation more efficient by choosing more suitable data structures. A difference language fulfilling these requirements provides more efficient ways of managing, manipulating and reusing difference information improving performance of data processing. The aforementioned properties

contribute to have a solid and common syntactic grounds for representing model differences in diverse domains and effortless development of further services on the top. A difference language satisfying the aforementioned properties is supposed to be executable, implementable in efficient ways, fully expressive, facilitate tool developers productivity with precise, concise and clear descriptions. It can easily be adopted to many domain-specific modeling languages regardless their graphical constructs. It is an efficient design of the data structure for representing model differences in repositories.

This thesis focuses on a generic *model difference representation technique* which provides *single point of truth* for concurrent, sequential collaborative modeling scenarios and model history analysis by supporting a common underlying *difference language* and a *list of supplementary services*. The main objective of this thesis is not focusing only on a single application, but being more generic with respect to the meta-models of modeling languages and model tools.

The significant principles listed in this section are the design foundations for a *difference language* that contribute to empower the qualification and solidity of difference representation. The proposed approach aims at fulfilling these requirements throughout this thesis.

4.4 Summary

The existing approaches offer various difference representation techniques such as *model/graph-based*, *text-based* and even *relational database-based*. The model-based and graph-based difference representation techniques are well-suited for the distributive sequential collaborative modeling scenario. However, they might not show high performance in case of concurrent collaborative modeling in real-time. These kind of representation approaches may require more effort and technical knowledge to develop collaborative modeling tools on the top. Because model- and graph-based difference representation requires technical spaces with sufficient data structures to represent models and graphs. There are several approaches focusing only on the particular aspects (e.g., sequential model version control by difference calculation and application) of the overall research challenge.

The existing approaches can be applicable to several application areas, but they can still be extended and improved with additional services and components to achieve successful results in terms of applicability. In this case, developing additional operative services require much effort and technical knowledge from the tool developers and domain experts. Some of the existing approaches discussed in this chapter usually rely on the meta-model of specific modeling languages, e.g., like EMF-based Ecore.

The result of literature study in this chapter indicates that there is still a need for extended research offering generic means to model difference representation in collaborative MDSE. A generic model difference representation approach is needed

which serves as the common grounds and generic difference representation for a wide range of modeling languages (w.r.t. meta-models) and modeling tools. It should provide *single point of truth* for concurrent and sequential collaborative MDSE by central repositories. A proposed model difference representation approach should satisfy several properties being executable, implementable, extensible, expressive and descriptive, generic with respect to modeling languages and tools (as defined in Section 4.3). These requirements are satisfied throughout this research work and evaluated in Section 10.2.

Part III

Approach

In Part I, the thesis has initially defined central research question and main objectives. Part II has reported on the core foundations of this thesis. These foundations are MDSE concepts, collaborative development use cases and related approaches to model difference representation and its operative services.

This part is dedicated to the core ideas behind this thesis. As this thesis focuses primarily on model difference representation, Chapter 5 presents *difference representation language* for model differences in modeling deltas. It firstly explains the conceptual idea behind the proposed difference language, then depicts a simplified motivating example for the approach. In addition to the model difference representation, this research work aims at providing several supplementary services which operate on the proposed difference representation approach. Chapter 6 explains these supplementary services in detail.

Chapter 5

Difference Language

As specified in Section 1.1, the *core objective* of this thesis is to introduce a generic difference language to model difference representations in collaborative MDSE. The discussions in Chapter 3 demonstrates that model difference representation is entirely crucial for use cases such as *concurrent collaborative modeling*, *sequential collaborative modeling* and *model evolution history analysis*. The extended literature review in Chapter 4 shows that there are several approaches dedicated to the problem of model difference representation. Most of these approaches can be well-suited for the particular part of the overall research challenge. However, these approaches can still be extended and improved covering a wide range of application areas and domains. To this end, this chapter presents a generic **Difference Language (DL)** to model difference (modeling delta) representation in collaborative MDSE.

Software models undergo various changes during the evolution process. These changes are the main reason for the evolution of software models. Thereby, the artifact changes are the first-class citizens in collaboration and the evolution process of software models. Constantly changing modeling artifacts results in the evolution of software models. Modeling artifacts can be created or deleted, or the attribute values of modeling artifacts can be changed when models evolve from one state to another.

As discussed in Chapter 3, there are several scenarios where the evolving artifacts of software models have to be properly identified and stored for further processing of software models. The most high-level scenarios are development and evolution (particularly maintenance) of model-driven software projects. Both scenarios require collaborative modeling. Overall collaborative modeling usually comprises the *concurrent collaboration* (Section 3.1) and *sequential collaboration* (Section 3.2) scenarios. In both scenarios, model changes are the first-class citizens.

The model changes define modifications made on the modeling artifacts. A list of changes can be aggregated as a collection which is called model differences and stored in *difference documents*, also referred to as **modeling deltas**. Consequently, each modeling delta consists of the collection of model changes. Model

changes are the first-class entities in this thesis for representing model differences (changes) in modeling deltas.

> ***Definition*** 5.1. **Model Change.**
> Any type of modification such as creation, deletion of modeling artifacts or changing the values of their attributes are referred to as *model changes*. Relationships among modeling artifacts are treated the attributes of respective meta-classes.

Model differences are usually detected and produced by comparing (or matching) the two states (revisions) of the same software models before and after change(s) are made. Sometimes, they are detected by recording (i.e., logging, listening) changes on modeling artifacts in modeling editors. The both approaches of computing modeling deltas are described as the *delta calculator* (discussed in Section 6.3 in detail) in this thesis.

> ***Definition*** 5.2. **Modeling Delta.**
> A collection of model changes is defined in terms of *model differences* and represented in difference documents which is referred to as **modeling deltas**.

As its core objectives, this thesis requests a notation, i.e., *Difference Language (DL)* to represent model differences in modeling deltas. Formally, DL is a family of domain-specific languages. Specific DLs are generated from the meta-models of modeling languages as depicted on the most upper level of Figure 5.1. The editing steps are described in terms of DL in modeling deltas. The general concepts of DL can be seen as a specialized form of domain-specific languages according to its properties. In contrast to general purpose languages, DL provides only selected modification operations required for expressing basic model differences/changes in modeling deltas. DL operations that should not be performed as the part of model changes are not explicitly considered by simply not expressing the relevant DL operations [Seidl et al., 2014]. Because, the model difference representation approach in this thesis considers only atomic changes like artifact creation, deletion and attribute values changes without reflecting composite changes. Section 5.2 provides more details about this notion and explains a brief example for DL syntax.

As depicted on the central part of Figure 5.1, the approach also provides several additional DL services to produce, manipulate and reuse the DL-based modeling deltas. Afterwards, the application areas of the DL approach are developed and extended by the special orchestrations of these DL services (the bottom part of Figure 5.1). As the proof of the concept, the DL approach is applied to the *concurrent collaborative modeling* in Chapter 7 and *sequential collaborative modeling* in Chapter 8, as well as *Model History Analysis* in Chapter 9.

Figure 5.1 depicts the overall architecture and main outline of the approach. It has three main levels such as *DL Generation* (explained in this chapter) as its conceptual idea, *DL Services* (discussed in Chapter 6) and *DL Applications* (explained in Part IV).

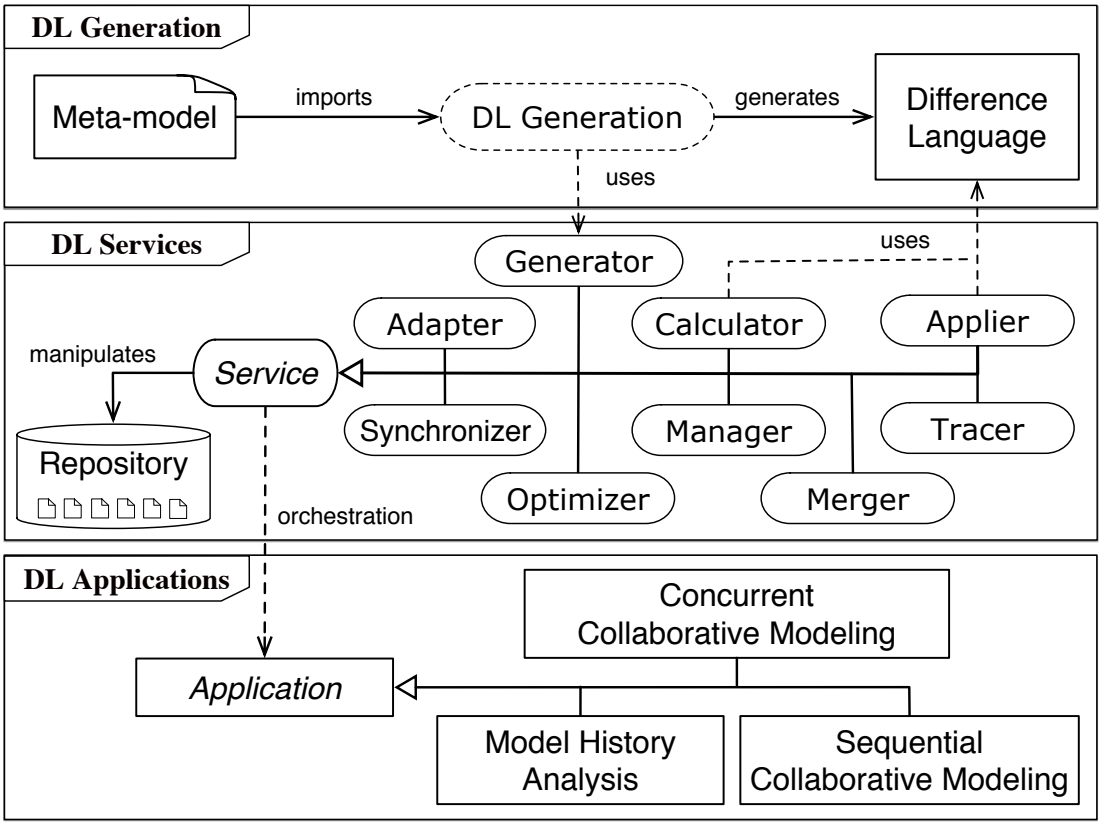

FIGURE 5.1: Overall Architecture of Approach

DL Generation. On the upper level, the architecture depicts the generation of specifics DL for specific modeling languages (explained in Section 5.1 in detail). The DL generator is meta-model generic. Generally, DL is a family of conceptual domain-specific languages, whereby specific DLs for specific modeling languages are generated from the meta-models of modeling languages (more details in Section 5.2). After generating specific DLs, all changes made on models conforming to the provided meta-models are represented in terms of the generated specific DLs. In general, the DL generator generates the DL syntax. The DL syntax is a special and self-defined form of general DSL. In the same vein, model changes are then represented by the DL operations that are referred to as *delta operations*. Each modeling delta conforming to a specific DL consists of at least one delta operation (cf. Section 5.2 for more details).

DL Services. The DL approach further provides several DL services (explained in Chapter 6 in detail) to produce, manipulate and reuse the DL-based modeling deltas stored in model repositories. Each DL service has a particular task and is involved in constructing the specific service orchestrations to

develop the DL applications. For instance, the delta *calculator* is used to compute differences between model revisions and produce modeling deltas. The delta calculator uses *optimizer* service to produce the optimized modeling deltas. After all, the other DL services such as delta *applier* and change *tracer* utilize the DL-based modeling deltas in further processing of models during their evolutionary life-cycle.

DL Applications. On the third level, Figure 5.1 depicts the DL applications developed by the specific orchestrations of the DL services. The DL services are orchestrated based on the specific operative scenarios of collaborative modeling applications such as *concurrent collaborative modeling* and *sequential collaborative modeling*, as well as *Model Evolution History Analysis* (described in Part IV).

Section 5.1 demonstrates the overall conceptual idea of the DL generation clarifying how to generate specific DLs for certain modeling languages in order to represent modeling deltas. Section 5.2 explains the DL-based model difference representation approach by a simplified running example. Section 5.3 sums up this chapter. While explaining the core ideas of the approach, the requirements defined in Section 4.3 are revisited how they are fulfilled by this approach.

5.1 Conceptual Idea: DL Generation

DL is conceptually a family of operation-based domain-specific languages dedicated to represent modeling deltas. Specific DLs for representing modeling deltas within the particular modeling languages are derived from the meta-models of these modeling languages. This thesis provides a *DL generator* (explained in Section 6.1) for automatically generating specific DLs from the given meta-models. This section describes how the generation process of specific DLs is performed by the DL generator. As proof of the concept, a specific DL is generated for UML activity diagrams in this section. But, the DL generator does not rely on a particular modeling language, i.e., it is modeling language generic with respect to the meta-models of modeling languages. The overall all DL-based collaborative modeling approach can be applied to a wide range of modeling languages by generating specific DLs for them (more applications in Part IV).

Metamodel.
As depicted on the upper part of Figure 5.1, specific DLs are generated from the meta-models of modeling languages as the initial step. According to the MDSE concepts explained in Section 2.1, the modeling concepts of any modeling language can be recognized by inspecting the meta-model of that language. Thus, a specific DL for UML activity diagram is generated by the DL generator service (Section 6.1) importing the UML activity diagram meta-model. Then, the model changes in modeling deltas can be represented in terms of DL on the instance activity diagrams.

Figure 5.2 depicts the structure of the UML activity diagram meta-model that is used as a running example. The meta-model is separated into two parts by a dashed line. Below the line, it depicts the content part (i.e., abstract syntax) which is adopted from the standard UML activity diagram meta-model for EMF (Eclipse Modeling Framework) [Steinberg et al., 2008].

In graphical modeling, each modeling object has design information such as ration, size, and position, also called layout information. Above the dashed line, Figure 5.2 portrays the layout part (i.e., concrete syntax) for the content part. The layout part of the meta-model depicts the substructure of Graphical Modeling Framework (GMF) notation [GMF, 2018] which supports notation for developing visual modeling editors based on EMF.

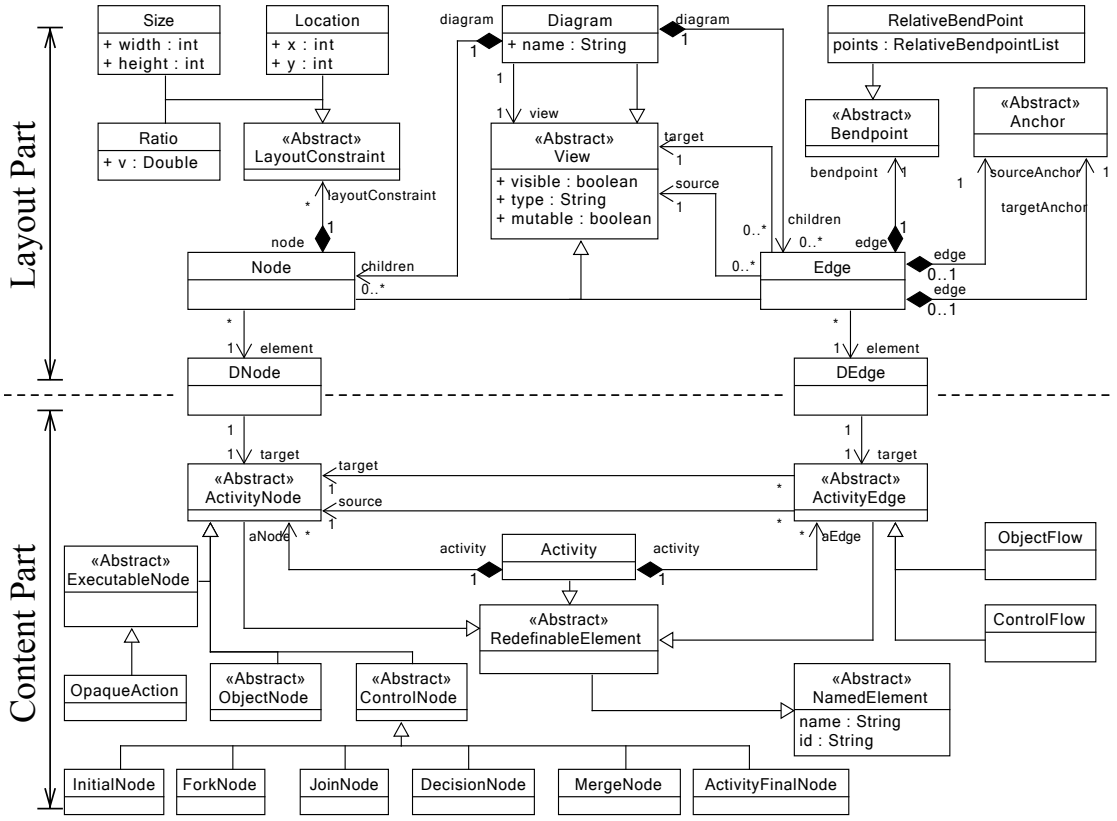

FIGURE 5.2: Meta-model for UML Activity Diagrams

The *ActivityNode* and *ActivityEdge* of the content part are connected to the *Node* and *Edge* of the layout part through the *DNode* and *DEdge* artifacts of the Sirius odesign notation [Viyović et al., 2014]. This simplified meta-model is utilized in applying the DL-based collaborative modeling infrastructure to a Sirius-based domain-specific modeling tool UML Designer in Section 7.3. *Sirius* is an EMF-based framework for developing domain-specific graphical modeling tools and it extends GMF notations for its visual representations. Therefore, the meta-model uses Sirius notations to combine GMF notation (layout part) and Ecore-based EMF meta-model for UML activity diagrams (content part).

A specific DL is generated for this combined meta-model of the UML activity diagram content (i.e., abstract syntax) and layout (i.e., concrete syntax) parts. Thus, a specific DL generated from this meta-model can also be used to represent changes in the layout data of instance models conforming to this meta-model. For instance, when the size, position or ratio of a model element is changed in modeling editors, these changes are synchronized among various parallel collaborators. Moreover, these changes are stored in repositories. This donates that the same underlying DL is used to represent model differences/changes for the layout and content of UML activity diagrams. Eventually, this way of designing meta-models allows for using the same collaborative modeling environment for different modeling contents.

Difference Language Generation.
The specific DL for the UML activity diagram meta-model is generated by the DL generator service explained in Section 6.1. While generating the specific DLs, the DL generator inspects all meta-classes. For each of these meta-classes, it iterates over the attributes and collects those that are changeable and not marked as the persistent identifier (e.g., the attribute *id* of *NamedElement* in Figure 5.2). Modification of the values of persistent identifiers are not allowed by default as an identifier tightly defines the identity of a modeling artifact and, therefore, should not be changed as the part of model changes. For each meta-class, the DL generator generates `creation` and `deletion` operations. For each attribute of these meta-classes, the DL generator generates a `change` operation. In general terms, the DL generator applies three basic operations, `create`, `delete` to each meta-class and `change` to each meta-attribute of a given meta-model. Relationships are treated as the meta-attributes of meta-classes. Thus, meta-associations and meta-attributes are referred to as *meta features* in Figure 5.3.

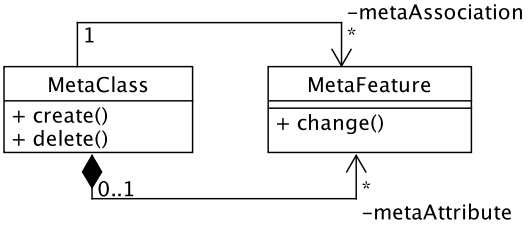

FIGURE 5.3: Conceptualization of DL Generation

For the sake of simplicity and to demonstrate the overall idea behind DL generation, this section presents generation of a specific DL for only UML activity diagrams including the content and layout parts. However, the DL generator is meta-model generic, i.e., it does not rely on a particular meta-model (modeling language). In the framework of this thesis, the DL generator is realized using JGraLab and EMF technical spaces (Section 2.4). Section 6.1 extensively discusses the further core principles of the DL generator and its realization in detail.

Difference Language.
A specific DL is generated in form of the *Java API* by the DL generator. Figure 5.4 depicts the abstract specification of the DL operations for the meta-model

in Figure 5.2. It is the combined abstraction of the *API (Application Programming Interface)* in Figure 5.5 and Figure 5.6. *It does not cover all possible cases*, yet depicts only abstract substructure just to show which operations can applied to each modeling artifact. According to abstraction in Figure 5.4, all modeling artifacts including flows and nodes can be deleted or created with parameters if they have attributes. Only attributes can be changed and all relationships are treated as attributes. Attributes are created when their container artifacts are created, and they are deleted when their container artifacts are deleted.

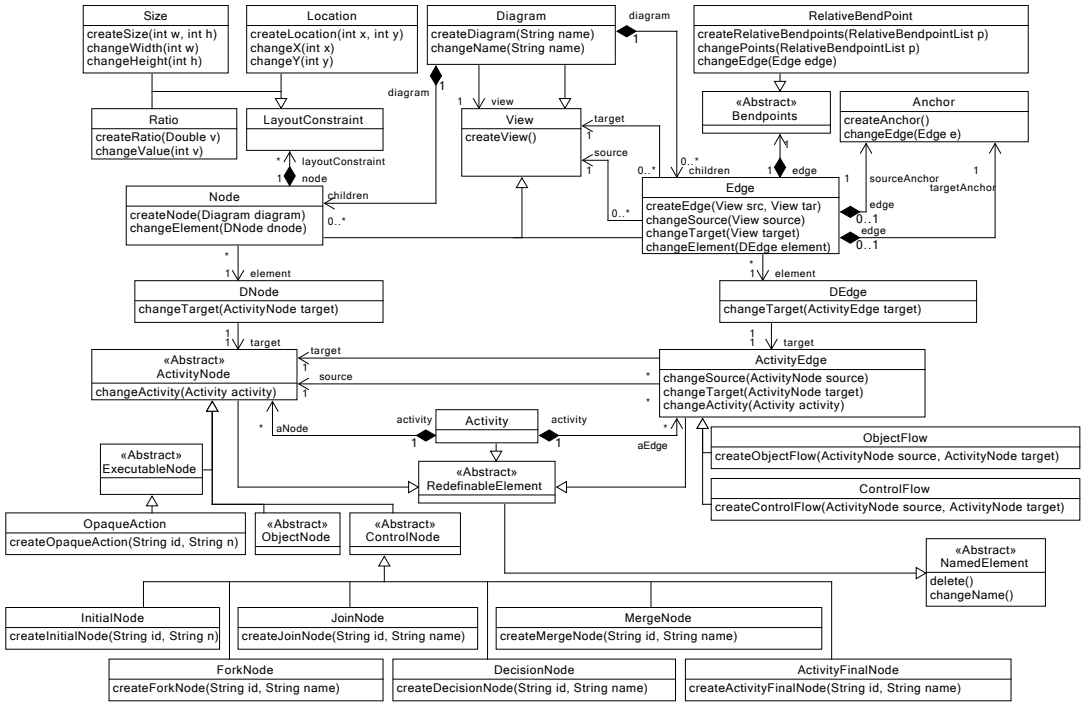

FIGURE 5.4: Abstraction of DL API

Figure 5.5 and Figure 5.6 depict only interfaces part of DL generated by the DL generator importing the UML activity diagram meta-model depicted in Figure 5.2. In general, DL consists of the interfaces (Figure 5.5 and Figure 5.6), their implementations to recognize modeling artifacts and a factory to operate on instance models, i.e., create, delete and change modeling artifacts. All methods of the resulting API are named and parameterized according to the modeling concepts described by the meta-model including the basic operations `create`, `delete` and `change`. The DL generator generates implementation of the interfaces in Figure 5.5 and Figure 5.6, as well.

For the sake of simplicity, the API in Figure 5.5 depicts the abstract representation of the most important change operations for the content part. The content part of API consists of creation operations for the meta-classes. These creation operations further contain operation parameters if the meta-classes have attributes or associations. The API further consists of change operations for all meta-attributes and associations. As long as all the meta-classes are of type *NamedElement*, the deletion operation is defined by deletion of that root element *NamedElement*.

```
1  //------------ Content Part (UML) in Content Package ------------
2  Activity createActivity();
3  InitialNode createInitialNode(String id, String name, Activity
       activity);
4  OpaqueActionNode createOpaqueAction(String id, String name, Activity
       activity);
5  ForkNode createForkNode(String id, String name, Activity activity);
6  JoinNode createJoinNode(String id, String name, Activity activity);
7  DecisionNode createDecisionNode(String id, String name, Activity
       activity);
8  MergeNode createMergeNode(String id, String name, Activity activity);
9  ActivityFinalNode createActivityFinalNode(String id, String name,
       Activity activity);
10 ControlFlow createControlFlow(String id, String name, ActivityNode
       source, ActivityNode target, Activity activity);
11 ObjectFlow createObjectFlow(String id, String name, ActivityNode
       source, ActivityNode target, Activity activity);
12 void changeName(String name); // for all meta-classes in content part
13 void changeSource(Node source); // for meta-classes of type Activity
       Edges only
14 void changeTarget(Node target); // for meta-classes of type Activity
       Edges only
15 void changeActivity(Activity activity); // for all meta-classes in
       content part
16 void delete(); // for all meta-classes in content part
```

FIGURE 5.5: Content Part of API Generated From Meta-model in Figure 5.2

As depicted in Figure 5.6, the same method of DL generation is applied to the layout part of the meta-model. *Nodes* or *Edges* can not exist without *element*, i.e., *DNode* or *DEdge*; as well as *DNode* or *DEdge* can not exist without their *target*, i.e., *ActivityNode* or *ActivityEdge*, respectively. In order to have possibility to manipulate *children* nodes that are also of type *View*, the *View* meta-class can be created and deleted including their *Layout Constraints*, *Bendpoints* and *Anchors*. Figure 5.6 does not consist of delete operation, because layout information is automatically deleted whenever its relevant element in the content part is deleted.

Once the DL delta calculator (or listener) service detects any modifications while calculating (listening for) model changes, it calls the relevant method of the DL interface providing the necessary parameters with their values. These parameters are the persistent identifier as a *reference*, *modeling artifact* and attribute values if necessary. Each call of these methods results in an analogous operation with relevant parameters. The methods of the interface depicted in Figure 5.5 and Figure 5.6 returns objects for creation operations and void for change and delete operations. These objects are considered as the part of *Delta Operations*. The list of objects of type Delta Operation can then be aggregated as *Delta Representation* to define a modeling delta. The DL API also supports a *serialization provider* which can serialize the objects of type Delta Operation into textual DL and vice verse.

The DL approach considers three atomic operations `create`, `delete` and `change` as a sufficient set of operations for representing all model differences. Below, the general syntax of these operations is explained in detail.

Creations. The create operations specify creation of new artifacts that exist in the new revision of a model and did not exist in the previous revision. The

```
1  //------------ Layout Part (GMF) in Layout Package ------------
2  View createView(boolean visible, String type, boolean mutable);
3  void changeVisible(boolean visible);
4  void changeType(String type);
5  void changeMutable(boolean mutable);
6
7  DNode createDNode(ActivityNode target);
8  void changeTarget(ActivityNode activityNode);
9
10 DEdge createDEdge(ActivityEdge target);
11 void changeTarget(ActivityEdge activityEdge);
12
13 Size createSize(int width, int height, Node node);
14 void changeWidth(int width);
15 void changeHeight(int height);
16 void changeNode(Node node);
17
18 Location createLocation(int x, int y, Node node);
19 void changeX(int x);
20 void changeY(int y);
21 void changeNode(Node node);
22
23 Ratio createRatio(Double v, Node node);
24 void changeValue(int v);
25 void changeNode(Node node);
26
27 Node createNode(DNode element, Diagram diagram);
28 void changeElement(DNode dnode);
29 void changeDiagram(Diagram diagram)
30
31 Diagram createDiagram(String name, View view);
32 void changeName(String name);
33 void changeView(View view);
34
35 Edge createEdge(View source, View target, DEdge element, Diagram
       diagram);
36 void changeSource(View source);
37 void changeTarget(View target);
38 void changeElement(DEdge dedge);
39 void changeDiagram(Diagram diagram);
40
41 RelativeBendPoint createRelativeBendPoint(RelativeBendpointList
       points, Edge edge);
42 void changePoints(RelativeBendpointList points)
43 void changeEdge(Edge edge);
44
45 Anchor createAnchor(Edge edge);
46 void changeEdge(Edge edge);
```

FIGURE 5.6: Layout Part of API Generated From Meta-model in Figure 5.2

overall syntax of the creation operations is as follows:
«persistentIdentifier» = create«ModelingArtifact»(value[0..]);*
As a whole, creation operations embody the basic *create* operator, *object* (modeling artifact that is taken from the meta-model) and the set of multiple parameters. The whole operation is assigned to a persistent *universally unique identifier*. If a modeling artifact has attribute(s), creation operations may have parameter(s) in the parentheses. If a child (contained) artifact should be created by a creation operation, the parent artifact should be provided to that creation operation (cf. Section 5.2 for more details).

Deletions. The deletion operations can be applied to all modeling artifacts. It defines a modeling artifact that does not exist in the new revision of a model and existed in the previous revision. With a delete operation, associations connected to a modeling artifact that should be deleted are automatically

deleted if they are not reconnected or deleted by any previous operations. If a modeling artifact is deleted, its attributes are also automatically deleted. The overall syntax of deletion operations is as follows:
«persistentIdentifier».delete();
Deletion operations consist only of the basic *delete* operator. The modeling artifacts that should be deleted are referred to by a persistent *persistent identifiers*.

Changes. The target or source end of meta-associations and the attributes of meta-classes can be changed. Modeling artifacts exist on the both old and new revisions of a model, but they might be changed from revision to revision. Changing the target or source end of any association is done by changing the attribute of a modeling artifact that is of type association. The overall syntax of change operation is as follows:
«persistentIdentifier».change«Feature»(newValue);
Syntactically, change operations depict the basic *change* operator, attribute name and one attribute value at most. The change operation can only be applied to the attributes of modeling artifacts to change their values on instance models. If any end of an association should be changed, the source or target end of that association has to be provided as a parameter *newValue*.

Section 5.2 illustrates a motivating example for the DL-based difference/change representation including all aforementioned operations.

Composite operations. When presenting the results of this thesis among research community, there has been always the question of composite operations like *move* operation, i.e., moving a part (a group of modeling artifacts) of a model from one part to another. But, this thesis considers that the *move* operation is necessary mostly in further analysis of the history of evolving models. From the technical point of view, the move operation can successfully be derived by the combination of the atomic DL operations. The move operation can be derived by changing association ends between the part that should be moved and the rest of a model. Detecting such composite operations from DL-based modeling deltas requires to define concrete rules and semantic lifting of atomic operations to higher/user level operations. Lifting atomic operations to composite ones is out of the scope in this thesis and one of such novel approaches is introduced in [Kehrer et al., 2011].

5.2 Motivating Example

Section 5.1 has explained how to generate specific DLs from the meta-models of modeling languages. The DL generator introduced in Section 6.1 is generic with respect to the meta-models of modeling languages, i.e., it does not rely on a particular modeling language. In order to explicitly explain how model differences are represented in terms of the newly generated DL in Section 6.1, this section

presents a simplified example of the DL-based difference representation approach. A very simple UML activity diagram [Raumbaugh et al., 2004, pp. 95ff] is chosen as a running example to apply the DL approach and to show how to represent model differences in terms of DL [Appeldorn et al., 2018]. Section 5.2.1 depicts a sample model of UML activity diagram. Section 5.2.2 gives the detailed descriptions of modeling deltas for the example models in Section 5.2.1. The DL syntax is explained in Section 5.2.3 in detail.

5.2.1 Sample Model and Model Changes

Figure 5.7 depicts three subsequent revisions namely `Rev_1`, `Rev_2` and `Rev_3` of the same UML activity model portraying an `"Order System"` example. All model revisions conform to the same simplified meta-model shown in Figure 5.2. Figure 5.7 further depicts two concurrent copies of the latest revision, in this case, `Rev_3`. Two designers, namely `Designer_1` and `Designer_2` are working on these parallel copies.

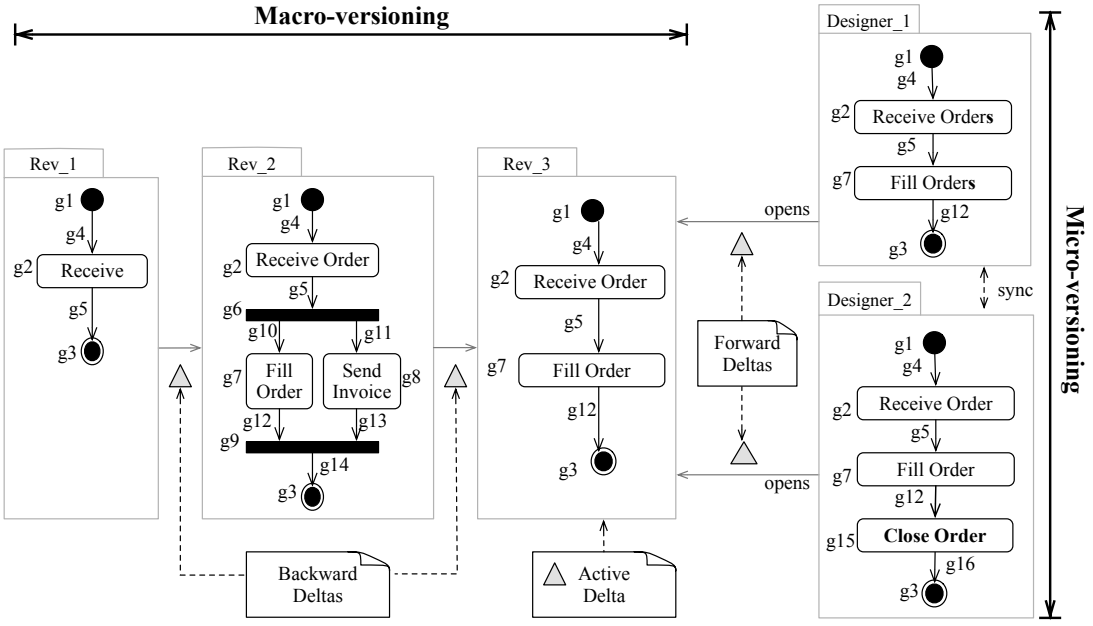

FIGURE 5.7: Example Activity Diagram under Collaborative Development

Model Changes.
In the first revision (`Rev_1`), the model consists only of one *Opaque Action* `g2` named `Receive` as well as the *Initial* `g1` and *Final* `g3` nodes. All modeling artifacts are connected by *(Control) Flows*; `g4` and `g5`. While evolving from the first revision (`Rev_1`) to the second (Rev_2), the following changes are made on the model: the *Fork, Join nodes* `g6` and `g9`, two *Opaque Actions* named `Fill Order` `g7` and `Send Invoice` `g8` are created, the target end of the control flow `g5` is reconnected to the fork node, the name of the action node `g2` is changed from `Receive` to `Receive Order`, and several control flows `g10`, `g11`, `g12`, `g13`, `g14` are created connecting these nodes. The model again evolves into the third revision (`Rev_3`) after making

the following changes: the target end of the control flow `g5` is reconnected to the opaque action `g7`, the target end of the control flow `g12` is reconnected to the final node, and the *Fork* `g6`, *Join* `g9` nodes, the *Opaque Action* `g8` and the *Control Flows* `g10, g11, g13, g14` are deleted.

In the third, last revision, the model is then being further developed by two designers concurrently. `Designer_1` changes the names of the opaque actions `g2` and `g7` from `Receive Order` and `Fill Order` to `Receive Orders` and `Fill Orders`, respectively. These changes are represented and sent to the other instance in form of the DL-based modeling delta. On the other instance, `Designer_2` creates a new *Opaque Action* with the name `Close Order`. The same designer creates one *Control Flow* `g16` and reconnects the target end of the control flow `g12` from the *Final Node* `g3` to the newly created opaque action `g15`. These changes are also represented and sent to the instance, `Designer_1` is working on, as the DL-based modeling delta.

Universally Unique Identifiers (UUID). In order to identify modeling artifacts and to represent referenced data in modeling deltas, each modeling artifact is assigned to a certain UUID (g_x). Assigning modeling artifacts to UUIDs allow for identifying and keeping the track of the modeling artifacts of evolving software models over time. As long as the names of modeling artifacts are not always unique, unique identifiers make evolution of artifacts more comprehensible and sufficient. With unique identifiers the *inter-delta* (i.e., predecessor and successor) and *delta-model* relationships can easily be detected. These identifiers are persistent throughout the evolutionary life-cycle of modeling artifacts. The persistent identifiers allow for tracing *inter-delta relationships* (predecessor and successor) of any particular modeling artifact by detecting the predecessor and successor artifacts of base modeling artifacts, and extract history information about the evolution process of these modeling artifacts. Furthermore, the inter-delta relationships provide foundations to extract the necessary data for the model evolution analysis application of DL that is explained in Chapter 9 (satisfying the requirement *RQ10: Traceable*). The *delta-model* references are used to refer to modeling artifacts from modeling deltas. The delta-model references are mainly used in applying modeling deltas to software models by the DL applier service (explained in Section 6.4). This feature of the DL-based delta operations fulfills the requirement *RQ5: Model Reference*. Only prerequisite in this case that either the meta-models or the data structures of technical spaces should provide an attribute to identify modeling artifacts.

5.2.2 Modeling Deltas

As defined in Definition 5.2, modeling deltas consist of the collection of the DL operations. The collection itself contains at least one DL operation. Modeling deltas modify the given base revision of a software model by creating new artifacts, changing or deleting existing ones. They are therefore applied to models under evolution and development in order to transfer them from one revision to

another. Application of modeling deltas to models is performed by the *DL applier* service. Furthermore, modeling deltas serve as information resources for the *DL tracer* service in mining history information and analyzing the evolution history of software models.

The DL-based modeling deltas can be distinguished in three forms according to their descriptions and transformation properties. According to the collaborative modeling scenarios (i.e., concurrent and sequential model versioning), modeling deltas are represented in the directed forms, i.e., *backward* or *forward* deltas [Mens, 2002] as depicted in Figure 5.7.

Modeling Deltas in Sequential Collaborative Modeling. In sequential collaborative modeling (cf. Subversion [Berlin and Rooney, 2006], Git [Swicegood, 2008]), the differences between subsequent revisions are usually identified and represented in reverse order, i.e., differences are represented in *backward deltas* as depicted Figure 5.7. Application of backward deltas to the given base model revisions results in the older revisions of the same model, i.e., the change descriptions have impacts on the base model in the backward order. The DL delta calculator service (Section 6.3) always receives two subsequent revisions of a model as inputs for calculating a modeling delta between these two revisions. While calculating the backward deltas, the delta calculator service receives the base (working) revision as the first input and the previous revision as the second input. In this case, the delta calculator produces a backward delta which transforms a model from the newer revision to older. The backward deltas are usually used in case of the sequential collaborative modeling scenario. Because, the working copies of models are the most frequently accessed revisions. The older revisions can easily be reverted by applying the backward deltas to that working copies. This makes sequential collaborative modeling more practical and performant. The same way of delta representation is employed by classic source code version control systems [Berlin and Rooney, 2006, Swicegood, 2008].

Modeling deltas in sequential collaborative modeling are computed by comparing two subsequent revisions of models. Therefore, in case of sequential collaborative modeling, modeling deltas consist of the larger set of model differences, whereas they represent the small set of model changes in concurrent collaborative modeling. Modeling deltas consisting of the larger set of differences (in sequential collaborative modeling) are referred to as *macro-versions*. Consequently, this thesis enables the *sequential collaborative modeling* application by *macro-versioning* as depicted in Figure 5.7.

In the macro-versioning scenario, the example depicted in Figure 5.7 describes three subsequent revisions (`Rev_1, Rev_2` and `Rev_3`) of the example model and the two *backward deltas* (Δ(Rev_2, Rev_1) and Δ(Rev_3, Rev_2)) between these revisions. As depicted in Listing 5.8, these backward deltas are directed in reverse order, i.e., each change leads from the later revisions to the older revisions of the same model. The macro-versioning scenario usually demands the reversal operations in order to obtain earlier revisions from the latter. The syntax of the DL-based operations is explained in Section 5.2.3 in detail.

```
1 g6 = createForkNode("");
2 g8 = createOpaqueAction("Send Invoice");
3 g9 = createJoinNode("");
4 g10 = createControlFlow("", g6, g7);
5 g11 = createControlFlow("", g6, g8);
6 g13 = createControlFlow("", g8, g9);
7 g14 = createControlFlow("", g9, g3);
8 g5.changeTarget(g6);
9 g12.changeTarget(g9);
```

FIGURE 5.8: Backward Delta: Δ(Rev_3, Rev_2)

Likewise, Figure 5.9 illustrates the backward modeling delta Δ(Rev_2, Rev_1) including the differences between the second and first revisions.

```
1  g2.changeName("Receive");
2  g5.changeTarget(g3);
3  g10.delete();
4  g11.delete();
5  g12.delete();
6  g13.delete();
7  g14.delete();
8  g6.delete();
9  g7.delete();
10 g8.delete();
11 g9.delete();
```

FIGURE 5.9: Backward Delta: Δ(Rev_2, Rev_1)

The modeling deltas in these figures are the executable descriptions of model differences. Each of these difference deltas allows to revert the base model to the earlier revisions from the latter. The modeling delta in Figure 5.8 reverts the model to the second revision from the third, whereas the modeling delta in Figure 5.9 reverts the same model to the first revision from the second. The concatenation of the modeling deltas from Figure 5.8 and 5.9 leads to a backward delta which transforms the model directly to the first revision from the third. The concatenation of modeling deltas is provided by the *DL optimizer* service which is explained in Section 6.8.

In the DL-based modeling deltas, the unchanged modeling artifacts are implicitly excluded simply not describing DL operations for them. The modeling artifacts that have to be changed are addressed by the persistent identifiers in the delta operations. For example, the third revision contains an *Opaque Action* `g2` named `Receive Order`. It is not referred in the modeling delta in Figure 5.8 because it is unchanged artifact. In the modeling delta in Figure 5.9, the name of that opaque action is changed to the previous name `"Receive"` (cf. Figure 5.9, line 1).

In Figure 5.8 and Figure 5.9, the modeling deltas consist of the list of only changes including all necessary information about the differences fulfilling the requirement *RQ11: Relevance* and the unchanged modeling artifacts are not referred satisfying the requirement *RQ8: Delta-based*.

Modeling Deltas in Concurrent Collaborative Modeling. In case of the concurrent collaborative modeling scenario, modeling deltas are described in a *forward form* where application of modeling deltas to a given model results in the newer revisions of the same model. In this case, the DL delta calculator

service further provides a *change listener/recorder* feature for listening for user actions/changes and record them in the forward deltas. However, the state-based model matching feature of the DL calculator service can calculate forward deltas if two subsequent model revisions are given as they are, i.e., the older revision as the first input and the newer version as the second input. The output of the delta calculator is then the forward delta in this case.

Modeling deltas in concurrent collaborative modeling are usually produced by recording each user action on model editors. In case of concurrent collaborative modeling, modeling deltas consist of the smaller set of model changes, whereas they represent the larger set of model differences in sequential collaborative modeling as discussed above. Modeling deltas consisting of small changes in concurrent collaborative modeling are referred to as *micro-versions*, whereas they are referred to as *macro-versions* in sequential collaborative modeling. This thesis develops the *concurrent collaborative modeling* application on top of the *micro-versioning* scenario as depicted in Figure 5.7.

The forward deltas are utilized in case of the micro-versioning as depicted in Figure 5.7. There, the changes made on one instance (e.g., by `Designer_1`) of a model are detected and sent to other parallel model copies (e.g., of `Designer_2`) in order to update these copies into the new states by the change descriptions in the forward deltas. For example, Listing 5.10 depicts the forward delta Δ(`Rev_3, Designer_1`) consisting of the changes made by `Designer_1`.

```
1 g2.changeName("Receive Orders");
2 g6.changeName("Fill Orders");
```

FIGURE 5.10: Forward Delta: Δ(Rev_3, Designer_1)

In this delta, `Designer_1` changes the name of both *Opaque Action* nodes from `"Receive Order"` and `"Fill Order"` to `"Receive Orders"` and `"Fill Orders"`, respectively. This example represents the model revisions where their changes are not yet synchronized with the working copy (in this case `Rev_3`) and other parallel instance of the model.

In the same vein, Listing 5.11 depicts the forward delta Δ(Rev_3, Designer_2) for the changes made by `Designer_2`.

```
1 g15 = createOpaqueAction("Close Order");
2 g16 = createControlFlow("", g15, g3);
3 g12.changeTarget(g15);
4 g19 = createDNode(g15);
5 g20 = createNode(g19);
6 g21 = createLocation(20, 30, g20);
7 g22 = createSize(10, 15, g20);
8 g23 = createRatio(3.4, g20);
```

FIGURE 5.11: Forward Delta: Δ(Rev_3, Designer_2)

In the forward delta in Listing 5.11, `Designer_2` makes the following changes: a new opaque action named `"Close Order"` is created, a control flow `g16` is created, and the target end of the control flow `g12` is reconnected to the newly created node `g15`.

For the sake of simplicity, the modeling deltas in this section do not consist of DL operations for changes in the layout part. Unlike the previous modeling deltas, this forward delta depicts DL operations for layout information starting from line 4. These operations describe change operations for only newly created opaque action *g15*. There, a new *Node* `g20` is created for the *Opaque Action* `g15`, meantime *DNode* `19` connecting these content and layout elements. On the followup two lines, *Location* (with the attribute values of x, y), *Size* (with the attribute values of width, height), *Ratio* (with the attribute value) are created for *Node g20* of the newly created *Opaque Action* `g15`. These last three operations refer to *Node* using their UUIDs as attributes in the operation body.

In the micro-versioning scenario, modeling deltas are represented by the DL operations in the forward forms. These forward deltas have the forward effect in the models, i.e., the models are updated with the change descriptions defined in the forward deltas. This is a scenario which usually occurs in case of micro-versioning. Because, the recent changes made by other parallel collaborators have to be propagated on models in order to keep them up-to-date.

Active Delta. The most classical source code version control systems [Berlin and Rooney, 2006] store the working copy of software projects and several (backward) deltas representing the differences between software revisions in their software repositories. Because, the working copies (base revisions) of software systems under collaborative development are the most frequently accessed revisions.

This thesis introduces a new term *active delta* to store the working copies of software models as modeling deltas. Active deltas are the DL-based descriptions of the base revision (working copy) of a complete model. Active deltas consist of only `creation` operations. Execution of an active delta creates a complete model out of an empty model ($\emptyset$). Eventually, the working copies of models are not necessarily required to be stored in the repository, instead the repository consists of only modeling deltas such as one active delta for a recent model revision and several difference (backward) deltas representing differences between subsequent model revisions.

The third revision (working copy) of the model depicted in Figure 5.7 is represented by the *active delta* ($\Delta(\emptyset$, Rev_3)) in Figure 5.12 which consists only of creation operations. When this active delta is executed on an empty model, the third revision of the model depicted in Figure 5.7 is generated.

```
1 g1 = createInitialNode("");
2 g2 = createOpaqueAction("Receive Order");
3 g7 = createOpaqueAction("Fill Order");
4 g3 = createActivityFinalNode("");
5 g4 = createControlFlow("", g1, g2);
6 g5 = createControlFlow("", g2, g7);
7 g12 = createControlFlow("", g7, g3);
```

FIGURE 5.12: Active Delta: $\Delta(\emptyset$, Rev_3)

This thesis distinguishes between these three types of modeling deltas based on their suitability and efficiency. As discussed above, represention of modeling deltas

in the backward order is more efficient and practical in the sequential collaborative modeling scenario. As long as modeling deltas are directly the executable descriptions of model differences, they can be reused and applied to the base models to revert them into older revisions from latter. Reverting is also used in the most source code-driven revision control approaches (cf. Subversion [Berlin and Rooney, 2006], Git [Swicegood, 2008], RCS [Tichy, 1985]). The backward deltas are often more effective in the sequential collaborative modeling scenario because they speed up retrieval of the older revisions of software system and probably the most frequently accessed revision.

In concurrent collaborative modeling, the use of the forward deltas is a more suitable choice. The forward deltas are used to update models with recent changes described in them. They transfer models from older states to newer states by propagating the DL-based change operations on the base model.

5.2.3 DL Operations

To represent model differences, the specific DL for UML activity diagrams is derived from its meta-model by applying three atomic operations `create`, `delete` and `change` to the each concept of its meta-model. The model differences in this example are then represented by the operations of the generated specific DL for UML activity diagrams. The complete scenario of deriving specific DLs and basic change operations are explained in Section 5.1 in detail.

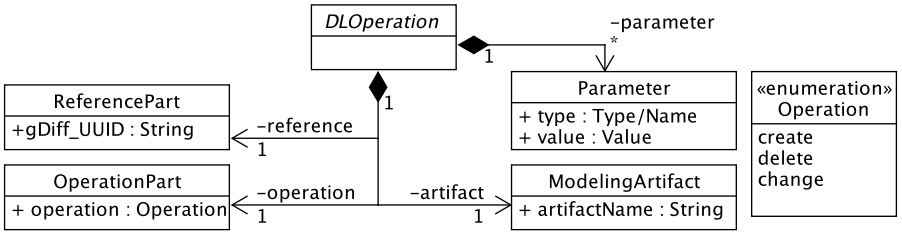

FIGURE 5.13: Conceptualization of DL Operations

Figure 5.13 depicts abstract conceptualization for DL operations. Each delta operation contains a *Operation part* (cf. *g1=createInitialNode("");*) which describes the *type of change* by means of *operations* (one of create, change, delete, explained in Section 5.1) and an *Modeling Artifact (Model Element)* (with attributes if required) (cf. *g1=createInitialNode("");*) which refers to the modeling concept. To refer to modeling artifacts from the DL operations in modeling deltas, persistent UUIDs are used as the *references*. For instance, the operation on the sixth line in Figure 5.12 creates a control flow connecting `g2` and `g7` and assigns it to `g5`.

Operation Part. This part of the DL operations depicts what type of modification is made on a modeling artifact, i.e., it defines the modification type. When the new revision of a model is produced, it is because of artifact creation, artifact deletion and/or attribute value change, i.e., modeling artifacts

can be created, deleted or changed during evolution process. Due to recognition and identification of the change type, DL classifies the model differences by the aforementioned three basic operations. DL makes the following three basic assumptions for its change types:

- If a modeling artifact did not exist in the previous revision of a model, but exists in the current revision, it is denoted as a *created* artifact. In backward deltas, creation operations are reversed as deletion operations.
- If a modeling artifact existed in the previous revision of a model, but does not exist in the current revision, it is denoted as a *deleted* artifact. In backward deltas, deletion operations are reversed as creation operations.
- If a modeling artifact exists in the both, previous and current, revisions of a model, the attribute(s) of that artifact is considered as *changed*, whereas it might be addition, removal or replacement of the attribute values.

These three operations are accepted as the sufficient set of operations for representing all model differences (more details in Section 5.1).

Modeling Artifact (Model Element) Part. Modeling artifacts in the DL operations are the concepts of modeling languages and they are taken from the meta-models of modeling languages. A modeling artifact embedded in this part is the one that has to be modified with the initial DL operation. This part of the DL statements helps to detect what kind of modeling artifact is being changed. It also ensures if the differentiated models conform to their actual concepts provided by meta-models.

Reference Part. Due to delta-model and inter-delta referencing, UUIDs are embodied in the reference part of each DL statement. It is the aid for creating delta-model (references from deltas to models) and inter-delta (references between deltas) relationships for further reuse and exploitation (e.g., delta application, change tracing, etc.) of modeling deltas.

Parameter(s). Each DL operation may consist of parameters. Meta-attributes and associations are defined by the parameter list of DL operations. For instance, opaque actions have names, i.e., the meta-class of opaque actions has an attribute called *name* and of type String. Thus, the active delta depicted in Figure 5.12 consists of creation operations for opaque actions with parameters (cf. line 2; *g2=createOpaqueAction("Receive Order");* or line 5; *g4=createControlFlow("", g1, g2);*).

Section 5.1 is dedicated to present the general idea of generating a specific DL from the meta-models of domain-specific modeling languages. This section has shown the very simplified example of the DL syntax for representing modeling deltas in the macro- and micro-versioning scenarios.

5.3 Summary

This chapter has briefly described how the process of DL generation is performed by the example of deriving the specific DL from the UML activity diagram meta-model. However, the approach is generic with respect to the meta-models of modeling languages. The resulting specific DLs are formed as Model API whereas the DL calculator and applier services use it by passing the required parameters for producing the DL statements and manipulating models. This chapter has further demonstrated the simplified example of the DL-based delta representation. The specification and syntax of DL is clarified by means of the simple example expressing the general idea in this thesis.

To sum up, DL satisfies several requirements that are listed in Section 4.3, so far:

RQ1: Meta-Model Generic. DL is conceptually a family of domain-specific languages. Specific DLs can be generated from the meta-models of modeling languages. The approach provides the DL generator for generating specific DLs. The DL generator is a generic service with respect to the meta-models of modeling languages. If the new versions (improved, extended, optimized) of the standard profiles of modeling languages are released, their respective meta-models can be adjusted accordingly, and new specific DLs for them can be regenerated without any further implementation effort.

RQ4: Operation-Based. Model differences in modeling deltas are represented in terms of the operation-based DL embodying model changes. In the framework of DL, model differences are distinguished by the modification operations such as `create`, `delete` and `change`, i.e., each modeling artifact can be created, deleted or its attributes can be changed during the evolution process. The syntax of DL is based on textual representations.

RQ5: Model Reference, RQ9: Persistent. Modeling artifacts are assigned to persistent UUIDs. It allows for identifying modeling artifacts, in the same vein, their changes. These identifications, in turn, serve as the references to modeling artifacts from modeling deltas by delta-model references (satisfying the requirement *RQ5: Model Reference*), as well as the references between modeling deltas by inter-delta references (satisfying the requirement *RQ9: Persistent*).

RQ6: Expressive. As demonstrated in the example in Section 5.2, DL has an expressive syntax and completely follows compound modeling concepts. The syntax of the DL operations is expressive, i.e., any non-expert user can easily read and understand what is intention behind each operation.

RQ8: Delta-based. Only changed modeling artifacts are referred to in modeling deltas. In modeling deltas, unchanged modeling artifacts are not considered by just not specifying DL operations.

RQ11: Relevance. The DL operations consist of precise information about each model change including the type of change, a reference to the changed modeling artifact and the conceptual name of the changed modeling artifact itself. The changed modeling artifacts are referred to according to their persistent identifiers stored in modeling deltas. Each DL operation encloses complete and only relevant information about every single change.

Difference representation is not the only focus of the DL approach, it also aims at providing several supplementary services for producing, further usage and exploitation of the DL-based modeling deltas in further analysis and manipulations. Chapter 6 explains several DL services which can produce, manipulate and reuse the DL-based modeling deltas.

Chapter 6

Difference Language Services

As discussed in Chapter 4, there are several model difference representation approaches mostly focusing on a subset of appropriate services for calculating, reusing, manipulating and further analyzing their difference information or modeling deltas. The main reason for storing model differences/changes in modeling deltas is to further reuse them in collaborative modeling. Therefore, representing model differences in modeling deltas is not only focus of the DL approach. Besides, this thesis introduces a reasonable set of the *DL services* enabling reusability, extendability and applicability of the DL-based modeling deltas. This chapter addresses these supplementary DL services explaining their functioning principles and main tasks behind each DL service.

As described in the list of research objectives in Section 1.1, this thesis aims at providing *a catalog of supplementary services* for reusing and exploiting the DL-based modeling deltas. The DL services (cf. Definition 4.1) depicted in Figure 6.1 are able to produce, manipulate and reuse the DL-based modeling deltas. The DL services discussed in this chapter are eventually utilized by the specific orchestrations of the DL-based collaborative modeling applications in Part IV.

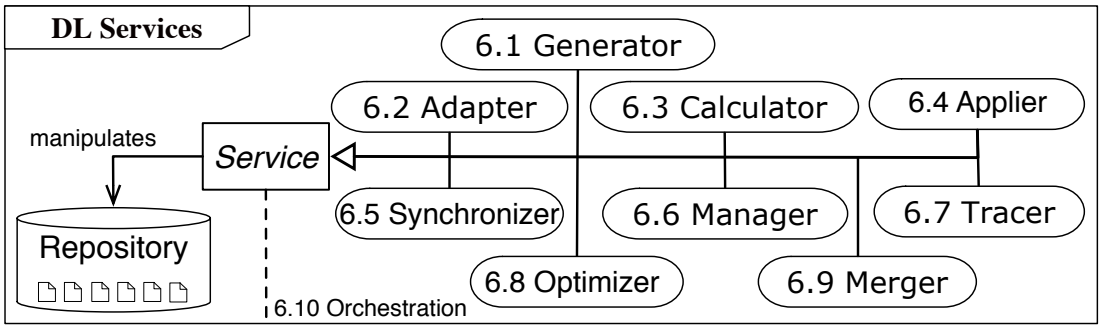

FIGURE 6.1: DL Services

Each DL service has a specific functionality in the specific service orchestrations in the DL applications. The DL services are shortly explained below according to their functionality. Afterwards, each DL service is described in a separate section of this chapter:

Generator. In order to apply the DL-based collaborative modeling infrastructure to particular modeling languages, specific DLs for these model languages have to be generated. This feature is provided by the DL generator service which is explained in Section 6.1.

Adapter. The adapter is introduced due to the integration purposes. It is used to parse software models between technical spaces. For instance, the most modeling tools can export models represented in the XMI serializations. These kind of models can be processed into internal graph-like representations (TGraph 2.4) and vice verse in case of the JGraLab technical space. This service discussed in Section 6.2 is rarely used only if there is a need to parse models between technical spaces or modeling tools and technical spaces.

Calculator. The delta calculator compares the subsequent revisions of the same model and produces modeling deltas in terms of DL. The calculator compares two different states (before and after changes made or parallel revisions) of a model using matching algorithms to detect changes (differences). The DL delta calculator service provides two methods of delta calculation: *state-based comparison* and *change listener*. In sequential collaborative modeling scenario, the *state-based* comparison is employed, whereas the model *change listener* is utilized in concurrent collaborative modeling. The DL delta calculator service is extensively discussed in Section 6.3).

Applier. The DL delta applier is used to apply modeling deltas to models to transfer them from one state to another. It is done by executing DL-based change operations that are described in modeling deltas. For instance, in case of the lost or damage of information on the working copy of a model in sequential collaborative modeling, collaborators might feel a need to roll back their models for obtaining the older revisions of it. In such cases, the delta applier helps to revert the older revisions of models. The applier is also employed in the concurrent collaborative modeling scenario in order to propagate model changes on the parallel copies of shared models (discussed in Section 6.4).

Synchronizer. In concurrent collaborative modeling, collaborators usually develop their software models in parallel in real-time. This requires synchronization of modeling deltas between these collaborators working concurrently. Thus, the DL approach offers a delta synchronization service which is clarified in Section 6.5.

Manager. Model management activities like storing models and their revisions in repositories, loading models and their revisions from repositories are inevitably crucial features which any collaborative development approach has to provide. In the same vein, the DL approach provides a model manager service to control models and their revisions in both, sequential and concurrent collaborative modeling applications. The DL model manager service discussed in Section 6.6 takes advantage of other DL services like *calculator* and *applier*.

Tracer. Tracing the changes of software models is a powerful support for collaborators in comprehending, recognizing and analyzing the model change histories and to make decisions about the further evolutionary life-cycle of model-driven projects. The DL change tracer detects history information about the changed artifacts of models. Information detected by the change tracer can then be browsed or visualized in various forms as collaborators want to see (discussed in Section 6.7).

Optimizer. There might be redundant or useless information in modeling deltas. The delta optimizer is employed to reduce these redundancies and inconsistencies in the DL-based modeling deltas. It allows for obtaining the optimized modeling deltas, eventually. Reordering the DL operations in modeling deltas is also done by the delta optimizer (discussed in Section 6.8).

Merger. In case of the sequential collaborative modeling scenario, the new branches (development lines) of software models are forked by *checking out-/copying*. Collaborators make changes on their working copies in their distributed working environments. After making their changes, they tend to merge their local changes into the main line (repository) of development. Merging requires to compare the master and local copies of the software models in order to detect if there are any change conflicts between compared revisions. The DL merger service takes advantage of the existing *graph merge* feature of the *JGraLab* technical space to deal with merge issue which is discussed in Section 6.9.

These services are sketched and implemented in the framework of this thesis. In the follow up sections, this chapter discusses these services in detail.

6.1 Generator

The process of specific DL generation is partially described in Section 5.1 by generating the specific DL for the UML activity diagram meta-model. This section further discusses more details how the DL generator service can be applied and how it is realized.

As mentioned before, the DL generator (Figure 6.2) is generic regarding the meta-models of modeling languages. It can be applied to a wide range of modeling languages. As proof of the concept, the DL generator service is realized using two technical spaces namely *JGraLab* and *EMF*.

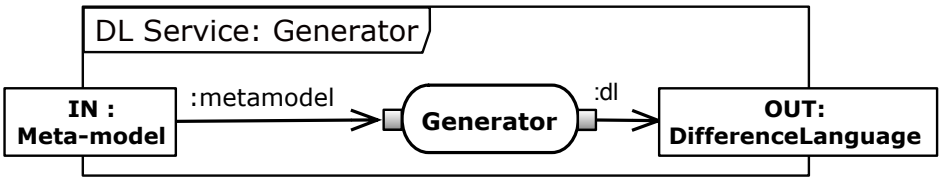

FIGURE 6.2: DL Service: Generator

As discussed in Section 2.4, the *JGraLab* (Java Graph Laboratory) [Dahm and Widmann, 1998] technical environment provides generic features for defining meta-models. In JGraLab, the TGraph schema [Ebert and Franzke, 1995] is used to define meta-models conforming to grUML (graph UML) which the profiled version of UML2 class diagrams. The DL generator service is realized for meta-models defined by TGraph schema. Current implementation of the DL generator service imports meta-models into JGraLab environment from XMI-based serializations from the Rational Software Architect (RSA) [Leroux et al., 2006], by the JGraLab API. The only exception is that every schema needs exactly one class with the stereotype *«graphclass»*. The name of a model becomes the qualified name of the resulting schema after importing it into JGraLab. The meta-models are exported into the **.xmi* formats from RSA and further processed using the **.schema* formats in the JGraLab environment.

The DL generator service is also realized using the Eclipse Modeling Framework (EMF) [Steinberg et al., 2008]. The EMF technical space is based on ECore and Eclipse platform for meta-modeling and model-driven software development. As depicted in Section 5.1, the EMF-hosted meta-models for DL generation combine the Graphical Modeling Framework (GMF) [GMF, 2018] and Sirius [Viyović et al., 2014] notations together with the UML modeling concepts. This combination of different notations allows for generating specific DLs with less implementation effort. If these three notations (i.e., GMF, Sirius, UML) are handled independently, it would require to work with three different sources of modeling notations and APIs, whereas newly generated DLs based on the combined meta-model allows for working only with single underlying meta-model (notation) and API. In order to generate specific DLs, these combined meta-models are designed using the ECore meta-modeling approach (with the **.ecore* formats) in the EMF technical space.

The code snippets of DL API are generated using Apache Velocity Templates [Naccarato, 2004]. The generic grammar of DL is defined in a template file. While generating specific DLs, the DL generator imports the predefined template file together with the meta-models of modeling languages to generate the API of DL in Java. The resulting overall DL consists of *Model API* including Java interfaces and implementations (in *model.impl*) for each modeling concept. It further contains a model utility API including the Adapter Factory to load resources and operate on the loaded resources.

If specific DLs for other modeling languages should be generated, the DL generator follows the same principle to generate specific DLs for the given meta-models. The DL generator can receive any meta-model designed as the UML class diagram (in the **.xmi* formats) and generate specific DLs conforming to the input meta-models. The DL generator does not rely on any specific meta-model satisfying the requirement *RQ1: Meta-model Generic*. After all, differences between the subsequent and parallel revisions of any instance models can be represented in terms of the DL notations.

6.2 Adapter

Software models are usually created and designed using domain-specific model designing tools like Visual Paradigm (VP) [Visual Paradigm, 2013], Rational Software Architect (RSA) [Leroux et al., 2006], EMF-based tools [Steinberg et al., 2008], Papyrus [Lanusse et al., 2009] or UML Designer [Obeo Network, 2017]. Most of these tools do not provide the proper open-source concurrent and sequential collaborative modeling scenarios or model evolution history analysis support.

Integrating the collaborative modeling scenarios or history analysis applications with the existing modeling tools is a challenge. However, these tools provide export and import of software models by, e.g., XML Metadata Interchange (XMI) [Cover, 2001] serializations. Therefore, in order to exchange models between the external modeling tools and the DL-based collaborative modeling support, this thesis provides a *DL adapter* service. Figure 6.3 depicts the overall architecture of the DL adapter service only for the JGraLab technical space.

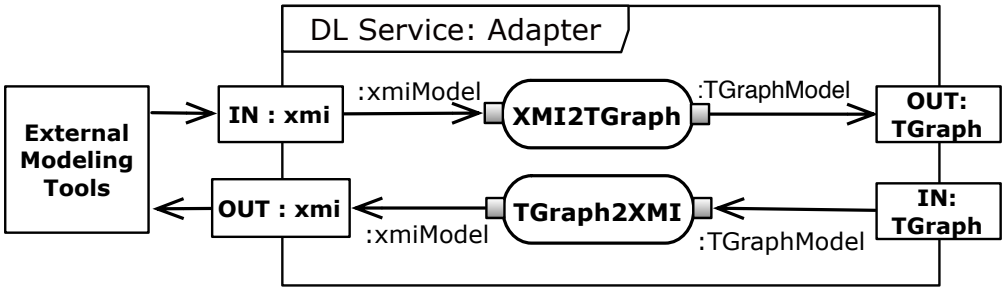

FIGURE 6.3: DL Service: Adapter

In order to make software models generally processable in the DL-based collaborative modeling infrastructure, software models designed using modeling tools can be imported into the DL-based collaborative modeling applications. The DL adapter provides two ways of parsing: (1) parsing models from the XMI exchange formats to the TGraph structures, (2) parsing models from the TGraph structures (explained in Section 2.4) to the XMI exchange formats. The adapter also takes advantage of the meta-models of modeling languages to recognize all modeling concepts specified in the XMI exchange formats. Therefore, it does not rely on modeling tools, modeling languages, and exchange formats and their revisions with respect to their meta-models. By providing the adapter, the DL approach satisfies the requirement *RQ2: Modeling Tool Generic*.

So far, the DL adapter service is realized using the JGraLab technical space. There, software models in the XMI formats are processed using Sax Parser [Grose et al., 2002]. The processed model elements are created using *DL applier* service inside the JGraLab technical space. As mentioned above, the meta-models of models that supposed to be imported always have to be defined in the relevant technical space before these models are imported. The meta-models of models to be imported have to be introduced to the JGraLab technical space as *TGraph Schema*.

6.3 Calculator

As long as model changes are the first-class artifacts in collaborative MDSE, identification of model changes itself is the initial stage of and essential grounds for the all DL services and applications. According to the definition in Defition 4.3, modeling deltas are basically calculated and produced by the *state-based matching algorithms* or *listening for user actions* on modeling editors [Mens, 2002], [Herrmannsdoerfer and Koegel, 2010]. The outcome of the both calculation features is referred to as the *modeling delta*. The DL delta calculator provides these two features of calculating its modeling deltas. It calculates modeling deltas using its *state-based matching* feature in the sequential collaborative modeling scenario, and the combination of the *state-based matching* and *change listener* features in the concurrent collaborative modeling scenario.

State-Based Model Matching.
The state-based matching feature of the delta calculator is used to compute the differences (i.e., modeling deltas) by comparing the two subsequent revisions of the same models. It usually receives two revisions (i.e., states before and after changes made) of the same model and produces the model differences between these revisions. The overall delta calculation process operates in two steps, namely *model matching* and *difference computation* [Brun and Pierantonio, 2008]. First of all, the differentiated model revisions are matched in order to compute the correspondences between the modeling artifacts of the two differentiated revisions of a model. Then, the actual model differences are detected by comparing all corresponding modeling artifacts.

Matching of two differentiated revisions of the same model is a challenge of finding the similarity of modeling artifacts with the direct predecessors of these modeling artifacts. If the identity of two modeling artifacts is explicit according to comparison of the certain properties of these artifacts, two modeling artifacts are considered to be similar, i.e., they are the two subsequent revisions of the same modeling artifact.

As discussed in Section 4.2, several model matching algorithms exist for detecting model differences and producing modeling deltas in the context of collaborative MDSE. The most existing model matching approaches can be divided into two main categories depending on how they identify and compute differences: *persistent identifier (ID-based)* or *similarity metrics-based*. In Section 4.2, the existing delta calculation approaches are broadly discussed including model matching techniques.

As long as there are several sufficient and solid state-based delta calculation approaches, this thesis utilizes the existing generic delta calculator framework `SiDiff` [Schmidt and Gloetzner, 2008] (explained in Section 4.2 in detail) for implementing its *state-based delta calculator service*. The `SiDiff` algorithm is a model difference calculation approach and configurable for many domains because of its graph-based representation of software models. `SiDiff` supports several matching algorithms for the graph-structured software models.

The DL delta calculator takes advantage of both *persistent identifiers – ID-based* and *similarity metrics – similarity-based* matching techniques of `SiDiff` and UMLDiff [Xing and Stroulia, 2005b]. In the framework of the DL delta calculator service, Küpker [Küpker, 2013] investigated existing delta calculation approaches such as UMLDiff [Xing and Stroulia, 2005b] and SiDiff [Schmidt and Gloetzner, 2008, Treude et al., 2007] and combined them into a *gDiff* (generic differentiating) tool. In order to be highly configurable and flexible, the DL delta calculator implements the following model matching techniques:

– *ID-based Matching.* The ID-based matching algorithm delivers the most accurate results in comparison to other matching algorithms. Especially, in case of the graph-like structures of models, the ID-based matching algorithm performs very fast and efficiently. If software models are designed within the DL application and/or persistent identifiers are available for modeling artifacts, the DL delta calculator can be set to the ID-based matching configuration. During experiments, the ID-based model matching algorithm has shown very high performance in the concurrent collaborative modeling scenario (more details in Chapter 10).

– *Similarity-based Matching.* In some cases, software models are designed using external model designing tools such as RSA [Leroux et al., 2006] or EMF-based UML Designer [Obeo Network, 2017], Papyrus [Lanusse et al., 2009] and can be exported into the XMI [Cover, 2001] exchange formats. When these models are imported into the DL-based collaborative modeling environment using the adapter service, persistent identifiers might not be available or not known for the DL delta calculator service. In such situations, it is still capable of calculating the differences of models imported from the external modeling tools. The DL delta calculator can be configured to operate using its *similarity metrics* for matching the non-identified artifacts of the imported models. The DL calculator provides two types of similarity metrics such as the *name* and *structural* similarities. The similarity-based model matching feature of the DL delta calculator service allows for handling software models designed using various external modeling tools.

If software models are designed using external modeling tools, their modeling artifacts may not have been assigned to UUIDs. The DL delta calculator service is implemented in a configurable manner, i.e., it can handle models with or without persistent identifiers. Persistent identifiers are assigned to modeling artifacts during the delta calculation phase. The DL delta calculator utilizes its similarity metrics to match modeling artifacts and detect the correspondences between the proper modeling artifacts. The newly assigned persistent identifiers are always used in the DL operations in order to preserve consistency of modeling deltas.

As depicted in Figure 6.4, the DL delta calculator expects two different revisions of the same model as inputs. It then matches the artifacts of two model revisions using its similarity metrics or ID-based model matching according to its configuration. The candidate artifacts with the highest similarity (if the similarity metrics

are used) are selected as the unique match for the host artifact. Afterwards, the created, deleted and changed artifacts (the type of artifact change) are detected out of the candidate artifacts.

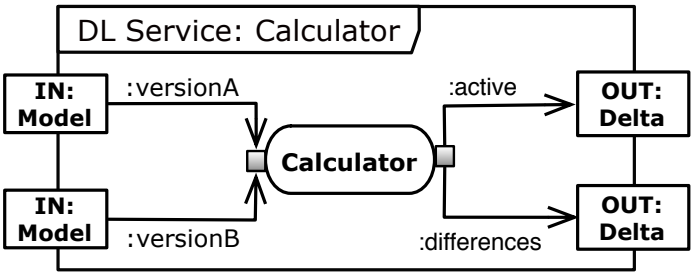

FIGURE 6.4: DL Service: State-based Calculator

As the result of the calculation process, the DL delta calculator produces two modeling deltas in terms of DL by using a specific DL interface (in Section 5.1): an *active delta* – the DL-based descriptions of the working copy of a model and a *difference delta* – representing DL-based descriptions of the differences between the given two revisions (as explained in Section 5.2).

Currently, the state-based delta calculation feature of the DL delta calculator service is realized using `JGraLab` technical space. The compared model revisions are represented using TGraphs internally. The resulting model differences are represented in modeling deltas in terms of DL, using the APIs of newly generated specific DLs. The state-based DL calculator is utilized in the DL-based concurrent collaborative modeling application `Kotelett` in Section 7.2. It is also used in realization of the DL model manager service in Section 6.6.

Change Listener.
The change listener (or recorder) is entitled as *change-based delta calculator* in some literature [Herrmannsdoerfer and Koegel, 2010], [Schneider et al., 2004] which listens for user actions and records (or logs) the detected changes. The detected changes are then represented in modeling deltas. The DL delta calculator provides the change listener feature for the concurrent collaborative modeling scenario. As long as concurrent collaborative modeling occurs in real-time, changes have to be instantly detected on model editors and recorded in modeling deltas, as well as synchronized in real-time providing sufficiently high performance. This signifies the use of the change listener feature in concurrent collaborative modeling instead of the state-based matching to detect modeling deltas.

As depicted in Figure 6.5, the change listener feature of the DL delta calculator service is set to models that are being changed. While the change listener is set, all changes are recorded (i.e., logged, registered) into modeling deltas. The change listener represents the model changes in terms of DL, as well. The change listener is used in the concurrent (real-time) collaborative modeling in order to ease detection of model changes in real-time.

The change listener and state-based comparison features of the DL delta calculator service are utilized in the concurrent collaborative modeling applications of DL in Chapter 7. The collaborative modeling application in Section 7.3 takes advantage

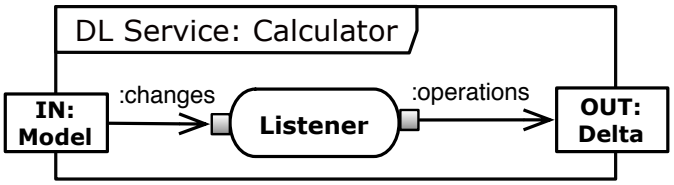

FIGURE 6.5: DL Service: Calculator – Change Listener

of the DL change listener for EMF-hosted models which is realized using the *Resource Set Listener*. It listens for the *Transactional Editing Domain*. The use of transactional editing domain provides to perform the redo/undo operations without extra implementation effort.

The DL change listener realized in the EMF technical space listens for *Notifications* which consists of information about the changed modeling artifacts and the change types. The change types in these notifications are mapped to the DL change types. The change types *ADD*, *ADD_MANY* are mapped to the *create* operation, *REMOVE*, *REMOVE_MANY* are mapped to *delete*, *SET, UNSET* are mapped to *change*, and *MOVE* is handled by the combination of the *create* and *delete* operations.

The delta calculator (incl. state-based delta calculator and change listener features) identifies only changed artifacts and allows for representing them in modeling deltas instead of storing whole model revisions. This eases the storage of subsequent model revisions in sequential collaborative modeling and real-time change synchronization in concurrent collaborative modeling. Thus, the delta calculator is one of the central components of the DL-based collaborative modeling.

6.4 Applier

As depicted in Definition 4.4, difference application is the process of transforming software models from one state (revision) to another by creating new artifacts, removing existing ones or changing the properties of existing artifacts.

Application of modeling deltas to models is the central task for the most collaborative MDSE approaches. Similarly, applicability of modeling deltas to software models is an essential requirement for this thesis. As listed in Section 4.3, applicability of modeling deltas is enabled by the executable descriptions of model differences (*RQ7: Executable*). Thus, this section introduces the *DL applier* service in order to apply the DL-based modeling deltas to software models under collaboration.

There are two main scenarios of applying modeling deltas to models in the DL-based collaborative modeling. These application scenarios are listed and described, below:

– *Reversion.* Modeling deltas are used to transform software models from one revision to another. In case of the *sequential collaborative modeling* scenario,

there might be lost or damage of information in models or the users might want to see their model how it was in earlier revisions. Thereby, models have to be reverted to older revisions in order to avoid lost or damage of information or to see the other revisions of the models. The DL applier transforms models from one revision to another by applying (the chain of associated) backward modeling deltas in sequential collaborative modeling. The delta applier is further used by the DL model manager service (explained in Section 6.6) to load the working copies of models by applying active deltas to empty models.

– *Propagation.* In case of the *concurrent collaborative modeling* scenario, user changes on one copy of a model are detected by the DL change listener and sent to other parallel copies as forward modeling deltas. After forward modeling deltas are delivered to others, they have to be applied to other parallel copies of that model in order to propagate the newest changes on all other copies. It allows for keeping all parallel model copies up-to-date by synchronizing changes between all of these parallel collaborators. The DL applier service is used to propagate changes described in forward modeling deltas on the parallel model copies in concurrent collaborative modeling.

As depicted in Figure 6.6, the DL delta applier expects a *model* and a *modeling delta* as inputs. The input delta is applied to the input model and the result is another revision of the same model.

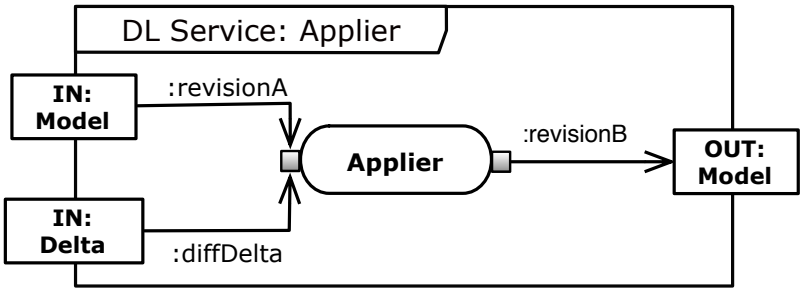

FIGURE 6.6: DL Service: Applier

According to the example in Section 5.2, execution of the active delta results in the working copies of models ($Version_3 = \emptyset.apply(\Delta_{active})$). To apply backward deltas to the working copies of models, initially the working copies of models themselves are reverted from active deltas. Then, application of each backward delta to the working copy of a model leads to the previous revisions from the current ($Version_2 = Version_3.apply(\Delta_{(3,2)})$ and $Version_1 = Version_2.apply(\Delta_{(2,1)})$.

If the DL delta applier is requested to revert the earlier revision of a model, for instance, the fifth revision from the fifteenth, the DL delta applier firstly invokes the DL delta optimizer which runs through the sequence of modeling deltas between the revisions fifteenth and the fifth. It attempts to optimize the patching process by concatenating the delta operations in these deltas. Because, several operations in these steps can be skipped using the optimizer to obtain faster and efficient reversion in the end. The optimizer helps to skip the subsequent list of

operations which does not really affect on the resulting revision (e.g., created, and later on, deleted artifacts).

The DL delta applier service is realized using the `JGraLab` and `EMF` technical spaces, so far. As DL consists of technical space-specific API to manipulate model resources and artifacts, the DL delta applier uses DL to manipulate models. The DL API uses *in-place model transformations* of `JGraLab` API (as described in Section 2.3) to manipulate TGraph-based models.

The delta applier service is also realized to be useful in the EMF technical space for concurrent collaborative modeling application as discussed in Section 7.3. It converts the DL operations in modeling deltas to executable commands in the *Recording Command*. Then, these operations are executed in the *Transactional Command Stack* of *Transactional Editing Domain* of EMF.

The DL-based modeling deltas form directly the executable descriptions of model changes satisfying the requirement *RQ7: Executable*. The executable descriptions of model changes in modeling deltas are another advantage of the DL notation.

6.5 Synchronizer

In concurrent collaborative modeling, developers open the different parallel copies of centralized software models and design their models in parallel in real-time. While changing their copies, these changes are constantly recorded by the *change listener* feature of the DL delta calculator service and produced as forward modeling deltas in terms of DL. Consequently, the produced forward deltas are synchronized with the other parallel copies of other collaborators so that changes made by all collaborators are propagated and reflected on all parallel copies of models under collaborative development. In order to support synchronization of modeling deltas in concurrent collaborative modeling, this thesis provides a *DL synchronizer* service (Definition 4.5).

As clarified in the example in Section 5.2, modeling deltas are described in the forward forms in concurrent collaborative modeling, where application of the modeling deltas to models results in the newer revisions of the same models. These deltas are synchronized with other parallel model copies in order to transfer them into the new states by the change descriptions in forward deltas.

The DL synchronizer service is built based on the *client-server* architectures to provide communication among collaborators. The DL synchronizer service is hosted on the server side, whereas each client is connected to that server once they join centralized models as collaborators. As depicted in Figure 6.7, the DL synchronizer receives forward deltas from the connected collaborators and sends them back to other collaborators. The synchronizer does not send modeling deltas to the original sender of that delta.

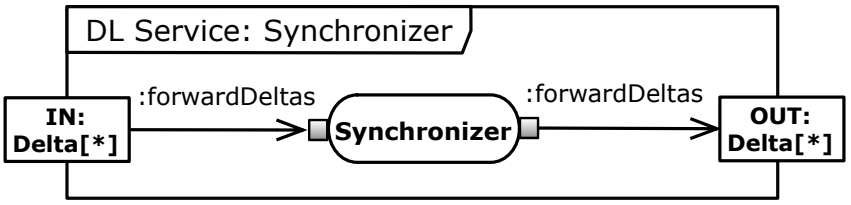

FIGURE 6.7: DL Service: Synchronizer

The server side of the DL synchronizer service supports synchronization of forward modeling deltas which is referred to as *micro-versioning* in the example in Section 5.2. Because, these forward deltas (i.e., micro-deltas) in concurrent collaborative modeling are rather small in comparison to the backward modeling deltas in sequential collaborative modeling. On the client side, the changes made by collaborators are constantly detected by the *change listener* while they are made using the model editors and constantly sent to other parallel clients through the *synchronizer service*. Once these deltas arrive at other collaborators, they are applied to models by the *applier service*. As long as the synchronization of modeling deltas among clients carried out by the synchronizer on the server, the communication between collaborators is provided based on *star-topology*.

The synchronizer service is realized using the `KryoNet API` (discussed in Section 2.4) and employed in developing the DL-based concurrent collaborative modeling applications in Chapter 7.

6.6 Manager

Any collaborative development system must be capable of storing and managing software projects and their revisions (i.e., change histories) under collaborative development and evolution. For instance, collaborators should be able do model management activities like storing and loading their software artifacts and their revisions (as defined in Definition 4.6). Likewise, the DL-based collaborative modeling infrastructure in this thesis provides the *DL model manager* service to manage models and their revisions under collaborative development.

As shown in Figure 6.8, the repository stores several backward modeling deltas and one active delta for each software model under collaboration. The DL model manager service directly operates on that repository. For instance, new models can be created in the repository, existing ones can be opened by collaborators to join collaboration, existing models can be deleted, revisions can be stored and loaded, etc.

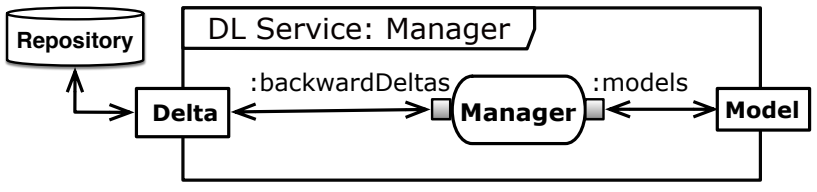

FIGURE 6.8: DL Service: Manager

Any revision of a specific model can be stored by collaborators' requests at any moment during concurrent collaboration and any model or/and any revision of a specific model can be loaded from the repository using the DL manager service. For instance, while developing models in concurrent collaboration, collaborations might store any state of their model in the repository and load it when needed.

The DL manager service takes advantage of the DL applier service to load models and to revert their revisions. When any collaborator wants to join a model as a collaborator, the manager service loads that model by applying a relevant active delta to an empty model using the delta applier service. If a collaborator is already working on a model and collaborator wants to load an earlier revision of model, the manager service reverts the requested revision by applying several backward deltas to the base model as many times as needed.

Currently, the DL model manager service is completely realized using the `JGraLab` technical space and partially realized using the `EMF` technical space. This service is employed in developing the DL-based sequential collaborative modeling applications in Chapter 8, as well as manager service on the server side of concurrent collaborative modeling applications in Chapter 7.

6.7 Tracer

As long as software models evolve over time undergoing various changes, developers and stakeholders intend to be aware of how their models evolve, see how was their model in a particular revision and analyze the histories of changes. For instance, they want to know about the list of modeling artifacts that are changing more often or when a specific modeling artifact was created or deleted. These analysis questions raise the problem of extracting history information about the evolutionary life-cycle of software models. Thus, this thesis provides a *DL change tracer* service for mining the DL-based modeling delta repositories by tracing modeling deltas. This section explains basic ideas and concepts behind the DL change tracer service.

Mining software model repositories (cf. Definition 3.5) plays an essential role in comprehending and analyzing the histories of evolving models. In fact, seeing the visualized or colored view of evolving modeling artifacts and their changes is more effective than reading the plain text. Collaborators intend to analyze the whole model histories or trace the specific aspects and artifacts of their models. It allows for presenting the structural and behavioral modifications to collaborators. Analysis of the model histories is a key support in comprehending and understanding the evolutionary life-cycle of models. According to knowledge obtained from history analysis, collaborators can make decisions about the further life of their models.

As described in Section 3.3, software systems are usually stored in *software repositories* [Arnold, 1996]. Likewise, this thesis also stores its DL-based modeling deltas in modeling delta repositories. As the DL approach can utilize the same

central repository for its both, sequential and concurrent, collaborative modeling applications, the DL tracer service takes advantage of the same underlying modeling delta repository as its main *information resource*. In software engineering, the change tracing action is also know as *repository querying* [Kagdi et al., 2007].

As given in Definition 4.7 and Definition 3.5, change tracing has been coined to describe a broad class of investigations into the examination of software repositories. Here, software repositories refer to modeling delta repositories representing modeling artifacts that are modified, produced and archived during collaborative modeling.

In the mining phase of modeling delta repositories, the DL change tracer service inspects the chain of associated modeling deltas. As the result, it then produces the associated data set according to the request provided by users. The change tracer service generates its results in ways that collaborators request. For instance, if collaborators request history information about a particular modeling artifact, the result consists of a list of changes obtained from the list of modeling deltas. If collaborators want to see only particular revision of their model, the only requested model revision is retrieved from the repository. In general, the DL change tracer capable of reverting all revisions of software models together with associated change-sets between these revisions. It can then be investigated in analyzing the evolutionary life-cycle of software models with the particular purpose asking questions such as *why? who? and when?*.

According to history analysis steps in Section 3.3 [Robbes, 2007], modeling delta repository mining undergoes two main steps: (1) *data extraction/retrieval* which retrieves necessary and pertinent information from modeling delta repositories, and (2) *data browsing, visualization* which properly visualizes difference information obtained from the data retrieval phase in a readable way. In the first phase, the DL change tracer receives the *chain of modeling deltas* from the repositories and *analysis queries* as input, as depicted in Figure 6.9. Then, it seeks for modeling artifacts and change information according to the *analysis queries* given by users. It verifies these modeling deltas based on the persistent identifiers of modeling artifacts. All predecessor or successor modeling deltas in the list are inspected in the same way and the result of change tracing query is reported as the set of associated data.

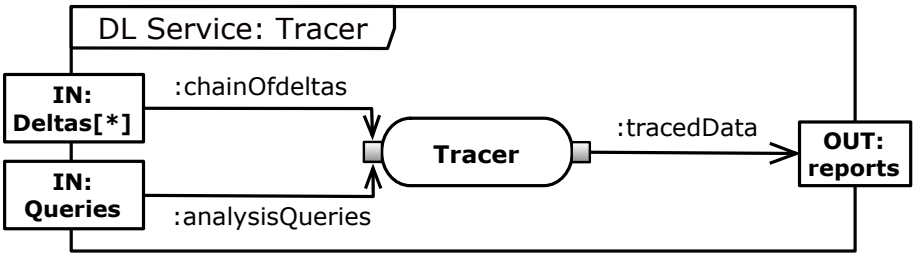

FIGURE 6.9: DL Service: Tracer

For instance, Figure 6.10 illustrates all history information about the control flow `g5` in the example in Section 5.2. The traced modeling artifact underwent three different changes in three revisions. These change information is traced by the DL

tracer service slicing three modeling deltas, the *active delta* in Figure 5.12 and two backward deltas in Figure 5.8 and Figure 5.9.

```
1 g5 = createControlFlow("", g2, g7);
2 g5.changeTarget(g3);
3 g5.changeTarget(ForkNode g6);
```

FIGURE 6.10: Change Report of *Control Flow* `g5`.

The DL change tracer service is capable of tracing the three basic categories of information from modeling delta repositories [Kagdi et al., 2007]:

- *Revisions.* Model revisions are mainly the states of software models after each change-set is made. Information about different revisions can be queried from modeling delta repositories. Information about revisions might be diverse according to the purpose of mining, analysis and even the type of software artifact. For instance, how many revisions does the modeling artifact `g5` have? in which revisions was the artifact `g5` created first? in which revision is it deleted (if deleted)? etc.
- *Differences.* As long as model differences are the first-class citizens in development and evolution of software models, the differences between artifact revisions are the main subject in analyzing the evolutionary life-cycle of modeling artifacts. For example, which change type is made to the target-value of `g5`? etc.
- *Meta-Data.* In addition to model revisions and differences between revisions, meta-data such as commit comments, user-ids, timestamps, and other similar information is also quite interesting information in history analysis. These meta-information describe, respectively, why, by whom and when the context of software artifacts is changed [Kagdi et al., 2007]. For example, who changed the artifact `g5` last time? when exactly a particular change is made on the artifact `g5`? etc.

After detecting all history-related data, the results of the DL delta tracer can be visualized in suitable ways. This thesis takes advantage of the combinations of the several visualization techniques discussed in Section 3.3. It is the combination of the *graph-based visualization*, *coloring differences* and *tabular view*. The detailed discussion of visualization techniques and user queries in the DL history analysis application are given in Chapter 9.

To sum up, the DL delta tracer service serves as the essential grounds for the model evolution history analysis application (Section 9) in this thesis. This service utilizes the DL-based modeling delta repositories as information resources for extracting necessary change history information. It enables users to reuse history information in analyzing the model evolution histories and find out answers to their various questions about model evolution. The DL delta tracer service satisfies the requirement *RQ10: Traceable*.

6.8 Optimizer

Application of the DL-based modeling deltas to models has to result in correct models. To obtain correct and non-redundant models as the result of delta application, modeling deltas should embody complete, yet correct and only relevant DL-based change operations as much as possible. Being concise and relevant is the key objective of modeling deltas. Correctness of modeling deltas provides consistency and correctness of software models and analysis of their evolutionary life-cycle. In order to optimize modeling deltas, this thesis introduces a *DL delta optimizer* service in this section.

In some cases, there might be a lot of useless operations or redundancies in modeling deltas. For instance, if a particular modeling artifact is created and deleted later in the same delta, two operations, creation and deletion, can be computed or recorded in a modeling delta where both have no effect on a model in the end. Another example might be changing one modeling artifact several times in one modeling delta. In this case, it is optimal to save only the last change instead of several change operations for that modeling artifact. In order to avoid aforementioned redundancies and increase efficiency by reducing useless operations and information in modeling deltas, optimization of modeling deltas is required. Optimization helps to receive more optimal modeling deltas, eventually. The DL delta optimizer (Figure 6.11) basically receives modeling deltas as input and returns optimized modeling deltas as the result.

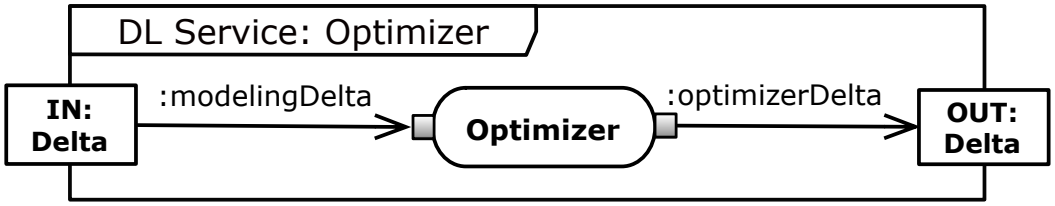

FIGURE 6.11: DL Service: Optimizer

Moreover, the order of delta operations in modeling deltas matters. Because, application of the DL operations to the base model has to follow certain prescriptions: creation operations have to be applied first, change operations second and deletion operations last. These prescriptions allow for obtaining concise and consistent software models after applying modeling deltas to models by the delta applier discussed in Section 6.4. The creation operations are lifted to the top and deletions are dropped to the end and changes are placed in the middle in all modeling deltas (for instance, cf. Section 5.2.2). Considering the graph-like structures of software models, the DL operations in modeling deltas are classified in that certain order as follows:

1. *Creations.* The creation operations must be applied to the base model, firstly. Newly created model elements might be referred to by other delta operations. For instance, if any end of a control flow has to be changed to a newly created node in case of the activity diagrams, a new node has to be

created first, then an existing control flow can be reconnected to that new node. There is a further assumption for only creation operations themselves. The creation operations for nodes (i.e., meta-classes) must be placed before the creation operations for edges (i.e., meta-associations). Because, (1) if an association should be created with a creation operation, its source and target artifacts must already exist in the model, (2) if an attribute should be created in a modeling artifact, that modeling artifact itself must exist.

2. *Changes.* Change operations are applied to the base model after creation operations and before deletion operations. For instance, if there is a modeling artifact that should be deleted by a follow-up delete operation, all associations connected to that artifact have to be reconnected to other modeling artifacts. So that the associations connected to the artifact that is being deleted remain in a model if necessary later on. Furthermore, (1) if the source or target end of an association should be reconnected, its new source or target end artifact must exist, (2) the same condition holds for the values of attributes, i.e., if the value of an attribute has to be changed, its container artifact must exist.

3. *Deletions.* The deletion operations have to be applied to models after all other operations. Deleting any modeling artifact earlier might result in structural error and redundancy in a model, later on. If a modeling artifact should be deleted, that deletion should not affect to other modeling artifacts and correctness of the overall model. Therefore, applying deletion operations in the end ensures the correctness of the application results. In modeling deltas, the deletion operations for edges (i.e., meta-associations) have to be placed before the deletion operations for nodes. Because, if a node deletion operation is applied to a modeling artifact, all links (associations) connected to that artifact are usually deleted. Moreover, if a modeling artifact should be deleted, it must exist in the model.

The delta optimizer is usually utilized by other DL services (e.g., DL calculator) to optimize the DL-based modeling deltas. The delta optimizer can also concatenate subsequent modeling deltas. The *delta applier* uses delta optimization to ease delta application process so that it can skip some delta operations between subsequent modeling deltas. For example, there are twenty revisions of the same model and it is requested to revert the tenth revision of that model. If a modeling artifact is created and deleted between revision ten and twenty (for instance, created in version fifteen and deleted in version eighteen), that both operations are, in fact, not necessary to consider in the reverting process. Because, they do not affect to the resulting (tenth) revision of the model.

Applicability of modeling deltas requires to follow the aforementioned concrete application conditions which are considered by the DL delta applier. These application conditions are considered and fulfilled by the *DL delta optimizer* service. This service improves the relevance of representation information in modeling deltas (partially satisfying the requirement *RQ11: Relevance*).

As long as DL-based modeling deltas are stored in delta repositories using textual representation, the DL optimizer service operates on the textual modeling deltas. It is realized in a generic way which independent from any MDSE technical spaces and can be used in any technical space.

6.9 Merger

In the sequential model versioning scenario, the new branches (development lines) of software models are forked by *checking out/copying*. Collaborators make changes on their local working copies. After making their changes, they *commit* their local changes into the main line (repository) of development. Before performing the commit operation, collaborators have to *update* their local working copies with the master copy in order to obtain the latest changes made by other collaborators. The update operation, in turns, requires to compare the master and local copies of software models in order to detect if there are conflicts between changes made by different collaborators. This scenario is referred to as the *merge* problem and an actively discussed topic by the current research committe. There are several approaches for merging model differences [Cicchetti, 2008], [Langer, 2011], [Altmanninger et al., 2007], [Koegel et al., 2010], [Brosch et al., 2010]. This thesis takes advantage of the existing *graph merge* feature of the `JGraLab` technical space. Thus, this thesis does not address the problem of model merge (incl. conflict resolution) and the extended discussion on this challenge is out of the scope in this thesis.

In model merging, there might be change conflicts. These conflicts should be somehow resolved either *automatically* if possible or with human-interaction [Altmanninger et al., 2009]. Change conflicts can not be fully resolved automatically but requires human interaction which is referred to as *semi-automated* merging. The most merge techniques perform possible merges, in unsolvable cases, they switch to interaction mode with collaborators for providing an interactive resolver feature. The unsolvable conflicts can then be demonstrated to collaborators in the form of recommendations by browsing conflicts [Koegel et al., 2010], [Brosch et al., 2010]. In a simple case, the DL merge service compares two model revisions to be merged, detect the possible conflicts, resolve them if possible, and finally obtain a complete, consistent model.

As depicted in Figure 6.12, the DL merger service receives two different revisions of the same model as inputs and produces the merger copy of these model revisions.

The DL merger service does not provide interactive conflict resolution feature. It merges the conflicting changes of the given model revisions into one if these conflicts are automatically resolvable. If the conflicting changes are not automatically resolvable, the both revisions of these differences are combined in the final revision, so that collaborators need to inspect the resulting model in order to resolve duplicated conflicts.

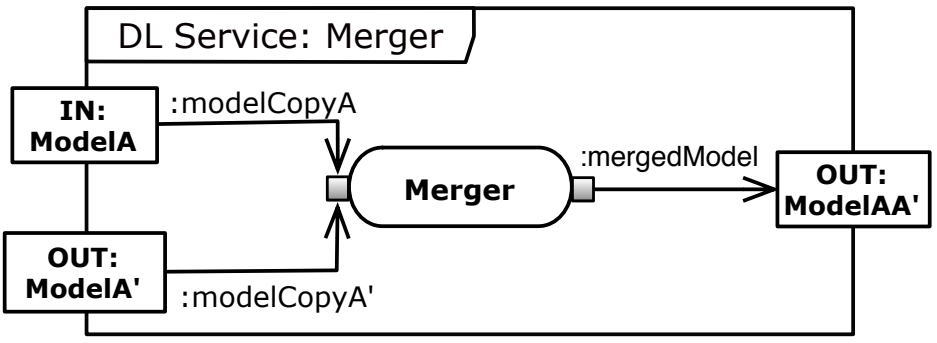

FIGURE 6.12: DL Service: Merger

Like the most merge techniques, the DL merger service classifies the following combination of operations as *conflicts* between two different model revisions:

Change/Change. A modeling artifact is supposed to be changed by two different collaborators at once. This case might occur only if it leads to different results in merging both changes; if the merge results are the same, this case is not treated as conflict.

Change/Delete. A modeling artifact is supposed to be changed by one designer and deleted by another or vice verse.

These scenarios are the most addressed conflict scenarios which require human interaction in case of merging two different revisions of the same model. If the results of any pair of changes are the same, then it is donated as non-conflicting changes.

The DL merge service is utilized in implementation of the sequential collaborative modeling application in Section 8.2. The concurrent collaborative modeling applications in Chapter 7 have not faced any conflicts during experiments with the team of about ten collaborators geographically located in long distance from the server. Thus, concurrent collaborative modeling applications did not need the merging and conflict resolution feature. This is probably attributed by the very small DL-based forward modeling deltas which enabled high performance in real-time collaboration. These small modeling deltas have provided fast synchronization of model changes before conflicts actually occur.

6.10 Service Orchestration

This chapter has introduced a catalog of the supplementary DL services so far. These services are capable of calculating, manipulating and reusing the DL-based modeling deltas. This section explains how these services can be utilized in constructing the DL-based collaborative modeling scenarios such as concurrent and sequential collaborative modeling, as well as model history analysis. The DL

services are orchestrated to perform specific tasks or fulfill certain operational scenarios. This section further demonstrates some orchestration scenarios of the DL services.

The DL-based collaborative modeling support enables collaborators to perform diverse operations. For instance, they design their models using modeling tools, where changes made on each copy of models are synchronized with the other parallel copies of that model. This overall scenario requires to *listen* for user actions to *detect* changes, *synchronize* with other working copies, and *apply* the changes to the other model copies. As this example scenario shows that several DL services are involved in the overall scenario. In order to accomplish this particular scenario and other similar scenarios (discussed below), all operative actions that might happen in the DL collaborative modeling require the certain **orchestrations** of the DL services.

> ***Definition*** 6.1. **Service Orchestration.**
> According to [Jelschen, 2014], arranging individual services in such a way to jointly achieve a more complex goal, is called *orchestration*.

The individual DL services are orchestrated according to the control-flow among the involved DL services. As long as each DL service does a particular task, they are involved in orchestrations whenever they are needed. An orchestration represents a chain of services, to that an extent, orchestrations and services form a composite pattern, with the orchestrations being the composites, and the DL services being leafs.

Chapter 3 has discussed several use cases of collaborative development including source code-driven collaborative development, textual document editing and collaborative modeling. According to these use cases, the most collaborative development approaches operate based on concrete scenarios. This section derives and briefly describes some main orchestration scenarios according to the discussions of use cases in Chapter 3.

- *Creating/Adding.* In the framework of the DL collaborative modeling, software models can be initially either created using existing modeling editors in the DL tooling environment or imported from external model designing tools (cf. Chapter 7 for more details). To import models from external modeling tools, these models have firstly be brought to the DL collaborative modeling environment by *add* operation. This scenario requires the *DL adapter* to parse that model from exchange formats into the DL internal model representations and *DL calculator* to produce an active delta for this initial revision of model. In this case, the service orchestration for this particular scenario invokes the *DL adapter* and *DL calculator* services.
- *Saving/Committing.* While developing software models in concurrent collaborative modeling, collaborators usually store the complete revisions of their

model at the end of a working day. In sequential collaborative modeling, collaborators usually check out a central repository into their local work space and manipulate software models on their local work space. After making their changes, they commit the local changes into the main repository, i.e., these changes have to be merged into the central repository.

The *save/commit* operation itself requires the orchestration of several DL services. If the model is designed using external modeling tools, the *DL adapter* firstly parses the model into the internal TGraph structure (cf. Section 2.4) from the XMI exchange formats. If the model is designed within the DL collaborative modeling environment using the concurrent collaborative modeling application, the DL adapter is not involved to *save* a model revision. Thereby, the *DL calculator* compares the model revision in main repository and the saved model revision, and produces one difference (backward) delta and one active delta in terms of DL. This scenario is repeated in each save/commit. The delta calculator uses the *DL optimizer* service to produce the optimized modeling deltas and store them in repository. The *commit* operation is usually used in case of the sequential collaborative modeling scenario, whereas the *save* is usually used in the concurrent collaborative modeling scenario. Eventually, the service orchestration in this scenario amalgamates the *DL adapter* (if needed), *DL calculator* and *DL optimizer* services.

– *Loading/Reverting.* Collaborators might need to load/revert other revisions of a model if they face loss or damage of data on their working copies of models. Thereby, the *DL applier* service, firstly, reverts the base revision of a model by applying an active delta to an empty model, secondly, applies several backward modeling deltas to the working copy of a model in order to obtain the requested revision of that model. The service orchestration invokes only *DL applier* service in this scenario.

– *Synchronizing.* If collaborators work on the shared and centralized model in parallel at the same time (i.e., in concurrent collaborative modeling), there are several parallel copies of the base model. Each collaborators makes changes on his/her copy of the model. As long as collaboration occurs in real-time, the changes on each copy of the model have to be detected and synchronized with other parallel copies of the model.

  In this scenario, model changes on each copy are detected by the *change listener* feature of the DL calculator after every single change is made, sent to other collaborators through the *DL synchronizer* and applied to their models by the *DL applier* service. After all, the synchronization scenario requires the orchestration of the *DL calculator: listener*, *synchronizer* and *applier* services. The detailed description of this scenario is given in Chapter 7.

– *Analyzing.* The model history analysis application in this thesis is built on the top of the *DL tracer* service. Like other scenarios, the analysis scenario requires the specific orchestration of one or more DL services. In the first place, history information has to be mined from the repositories for creating analysis views.

History information is extracted by the *DL tracer* service by analyzing a chain of modeling deltas according to the queries requested by collaborators. Then, the extracted change history reports are passed to a *visualizer* in order to visualize or browse history information in convenient ways that is understandable for collaborators. The DL history analysis application is discussed in Chapter 9 in detail.

These scenarios are very general forms of the DL orchestration scenarios, but there might also be other specific orchestrations of the DL services according to the needs of collaborators. In general, each DL service is designed and developed in a generic way (service-oriented manner), and has the particular roles and tasks so that they can be integrated and invoked in any specific orchestration. Because, collaborators might request building other scenarios rather than listed above.

This thesis distinguishes between different granularity levels of the DL services and service orchestrations in order to make them sustainable, adaptable, extendable and reusable. The DL services can be replaced by another analogous services or extended with other complementary features. Thus, it is not the strict catalog of the DL services and they still can be extended, improved with additional features or replaced by other services (satisfying requirement *RQ3: Extensible*). As already explained in this chapter, each DL service has a particular task receiving certain input data, performing its task and producing some results as output. The DL service orchestrations are the main foundations for building the DL applications that are explained in Part IV in detail.

6.11 Summary

Providing a list of supplementary services is the notable contribution of the DL approach. These DL services can produce, manage and use the DL-based modeling deltas. They extend applicability of DL in various application areas and modeling domains. This chapter has presented the set of core DL services that contribute to extension of the DL application areas by the specific orchestrations (discussed in Part IV). Besides, these DL services can further be extended, improved and replaced by other implementations.

Additionally, the DL services explained in this chapter satisfy some of the requirements listed in Section 4.3:

– *RQ2: Modeling Tool Independent.* There are several modeling tools and they have own internal model representation techniques. In order to handle models designed in different modeling tools, the *DL adapter* service provides transformation of models from the `XMI`-based exchange formats into the internal model representation TGraph and vise verse. The DL adapter enables integration of the DL-based collaborative modeling environment with existing CASE tools. If the DL delta calculator and applier services can

be adopted for further modeling tools (like it is for UML Designer in Section 7.3), the rest of collaborative modeling infrastructure can be reused for these modeling tools without additional implementation efforts.

– *RQ7: Executable.* The DL-based modeling deltas already form the executable descriptions of model differences. To provide applicability of the DL-based modeling deltas, this thesis has introduced the *DL applier* service. The DL applier implements these deltas by *in-place model transformations* in JGraLab and *Recording Command* in EMF for applying modeling deltas to software models.

– *RQ10: Traceable.* As described in Chapter 5, the change operations in modeling deltas are assigned to *UUIDs* which allow for effortless identification, elicitation and visualization of the histories of evolving software models from the DL-based modeling delta repositories. In order to enhance feasibility of aforementioned features and provide sufficient information in model history analysis, this thesis delivers the *DL tracer* service which makes the DL-based modeling deltas straightforward and accessible for further analysis.

The most existing research ideas provide solutions addressing these services independently or concentrating on the specific aspects of the general research problem. Alongside the generic difference representation technique DL, this thesis provides a catalog of services that are not strictly embedded behind DL. Instead, the DL services are realized in a *service-oriented, component-based* manner and can certainly operate independently from each other making DL a common underlying difference representation format (change exchange format) for all DL services. In this sense, DL and its services are generic, flexible and extensible. This separation of concepts, yet enabling integration of these concepts, is one of the relevant contributions of this thesis.

Any further services, making their only prerequisite to recognize the DL syntax, can be developed as the extension of the DL services. Thereby, DL serves as the efficient and convenient exchange format for modeling deltas among these services. Eventually, all DL services are utilized in developing the DL applications in Part IV.

Part IV

Applications

One of the main goals of any practical and engineering research work is its real-world applications alongside its novelty. Each research work has to be applied to the real-world applications by developing research prototypes in order to present its applicability and as proof of the concept. In the same vein, applicability criteria is one of the core objectives of this thesis. Thus, due to applicability of the approach and as proof of the concept, several application areas (use cases) of the approach are realized in this part of the thesis.

As already explained so far, the DL applications are developed by the specific orchestrations of the DL services. These applications are concurrent collaborative modeling applications `Kotelett, CoMo` (enabled by micro-versioning) discussed in Chapter 7, sequential collaborative modeling application *Generic Model Version Control System*–`GMoVerS` (enabled by macro-versioning) explained in Chapter 8, and model evolution history analysis application `MoHA` discussed in Chapter 9. The architectures of these application areas are developed by the specific orchestrations of the DL services explained in Chapter 6 and built based on DL as the underlying change representation concept. The DL-based modeling delta repositories serve as the *single point of truth* for all aforementioned DL applications.

Each chapter of this part is dedicated to the detailed description of the aforementioned DL applications, whereas each chapter follows the concrete structure; defining *overall scenarios*, *meta-models*, *concrete architectures*, *realization/tooling*, and *contributions of DL*.

Chapter 7

Concurrent Collaborative Modeling

As motivated in Chapter 3.1, writing huge textual documents, developing large-scale models or software systems require support for concurrent collaboration of multiple writers, designers or developers in real-time in parallel. In the same vein, support for collaborative development in MDSE is required. This chapter introduces concurrent collaborative modeling applications built based on micro-versioning.

As long as software models are usually very huge with several thousand artifacts, developing large models requires collaboration of several modelers in real-time in parallel. Due to the huge amount of artifacts and complexity of software models, modeling artifacts are designed and manipulated collaboratively by a group of collaborators. In concurrent collaborative modeling, collaborators are able to efficiently and quickly create and manipulate modeling artifacts in real-time. Additionally, it helps to reduce the occurrences of the change conflicts, enables collaborative discussions, and decision making process among collaborators simultaneously.

As explained in Section 1, the collaborative modeling support developed based on *micro-versioning*. Because, modeling deltas in this application consist of rather small set of DL operations. These change operations in deltas are detected while changing models on modeling editors. They are constantly detected by the change listener feature of DL calculator and produced in forms of the *small* modeling deltas consisting of only small set of changes.

The main focus of concurrent collaborative modeling is to provide the team-work of several designers and developers on the shared modeling artifacts in real-time. As long as modeling occurs in real-time in case of concurrent collaborative modeling, the high performance of modeling delta synchronization between the multiple copies of the shared models is one of the main challenges considered by DL. This issue is eased by *small modeling deltas* consisting of the small changes of models. In general, DL aims at providing the required support described in Section 3.1.3 for model difference representation.

Although concurrent and sequential collaborative modeling are two different scenarios of collaborative modeling, this thesis addresses to provide the both scenarios based on the same underlying representation technique DL. Interoperability of these scenarios is accomplished by the same common DL-based modeling delta repositories as a *single point of truth*. Thereby, an adequate and efficient difference representation technique is essential for synchronization of small changes among the various parallel copies of models under development and evolution. Furthermore, several services like the listener and delta applier are required for detecting modeling deltas and propagating these deltas on other parallel instances of these models. For that purpose, this research work has introduced a solid difference representation technique in Chapter 5 and services in Chapter 6 which rely on the DL syntax. Thus, this chapter applies DL to *concurrent collaborative modeling* applications.

As inspected in Chapter 4, very few collaborative modeling approaches exist supporting concurrent collaborative modeling. The most research in field of collaborative modeling are dedicated to the sequential collaborative modeling, or they focus on one of two collaborative modeling scenarios instead of considering both at once. *Web-based* online approaches (e.g., GenMyModel [GenMyModel, 2015] and creately [Cinergix Pty., 2015]) are usually hosted on the cloud-based web servers and exchange changes over WebSocket using web browsers.

In Section 7.1, this chapter briefly explains the general concepts and reference architecture for concurrent collaborative modeling. Section 7.2 and Section 7.3 introduces the DL-based concurrent collaborative modeling applications `Kotelett` and `CoMo`, respectively. The concrete architectures, utilized meta-models and features of these applications are described in their respective sections. The main contributions of DL in concurrent collaborative modeling are described in Section 7.4. This chapter ends in Section 7.5 by drawing some conclusions.

7.1 Reference Architecture

Section 3.1 has explained the state of the art in collaborative document editing, software development and modeling based on concurrent collaboration. The same section has defined the list of various common underlying principles, technologies and architectures of the existing concurrent collaborative development approaches. As long as concurrent collaborative modeling shares the similar underlying principles, technologies and architectures, this section takes them into account in developing the DL-based concurrent collaborative modeling applications in this chapter. Additionally, this section introduces a common *reference architect* for concurrent collaborative modeling applications. The reference architecture is then used as a blueprint in developing the DL-based concurrent collaborative modeling applications in Section 7.2 and Section 7.3.

Architectures and Synchronization.
As discussed in Section 3.1, the existing concurrent collaboration approaches are

built on top of the different collaboration architectures and synchronization algorithms providing various features. These architectures are revisited in order to decide on and derive the best candidate to develop the DL-based concurrent collaborative modeling application in this chapter.

Collaboration Architectures. The most existing concurrent collaborative systems are built using two different types of architectures. These are *centralized* and *decentralized*. The DL-based concurrent collaborative modeling application takes advantage of the *centralized* architecture. In this architecture [Nichols et al., 1995], the shared models are located in centralized repositories and shared with collaborators. Only single master copy and several client copies of the shared model is available in this architecture. This architecture is chosen for providing the high performance of synchronization of model changes in collaborative modeling in real-time.

Synchronization Algorithms. According to the calculation techniques of modeling deltas in concurrent collaborative modeling, the synchronization algorithms are classified into the *state-based* and *operation-based* techniques. The DL-based concurrent collaborative modeling utilizes the combination of both. The *state-based (differential synchronization)* approach considers the different states of the shared models. The state-based comparison is utilized in the `Kotelett` tool (Section 7.2) As long as performance is a key factor in concurrent collaborative modeling, the synchronization using the state-based change computation technique might not be very efficient in case of the very large models. Because, the differences between the states have to be computed every time after each edit action by users, resulting in increased time complexity [Ahmed-Nacer et al., 2011]. Considering the aforementioned challenge, this thesis uses the *operation-based synchronization* in one of its concurrent collaborative modeling applications. The operational synchronization technique is identified as the suitable approach and widely used by concurrent collaborative systems [Clarence et al., 1991], [Ressel et al., 1996], [Sun and Ellis, 1998]. Modifications on the shared software models are presented by means of the DL operations. To sum up, the DL-based concurrent collaborative modeling applications implement the state-based comparison of the shared model (`Kotelett` in Section 7.2), as well as the *change listener* (operational synchronization) feature (`Como` in Section 7.3) for detecting model changes in real-time. But, in both cases, model differences (changes) are represented by DL operations.

The most existing collaborative editing and development approaches further provide some additional tool specific features such as *model editor*, *user authentication*, *messaging*, *file sharing*, etc. These features are discussed in respective sections (cf., Section 7.2 and Section 7.3) of each DL-based concurrent collaborative modeling applications.

Reference Architecture for Concurrent Collaborative Modeling.
In order to realize the DL-based concurrent collaborative modeling applications,

Figure 7.1 depicts a common *reference architecture* for concurrent collaborative modeling. This reference architecture is sketched based on the main operational scenarios of the concurrent collaborative development approaches discussed in Section 3.1.

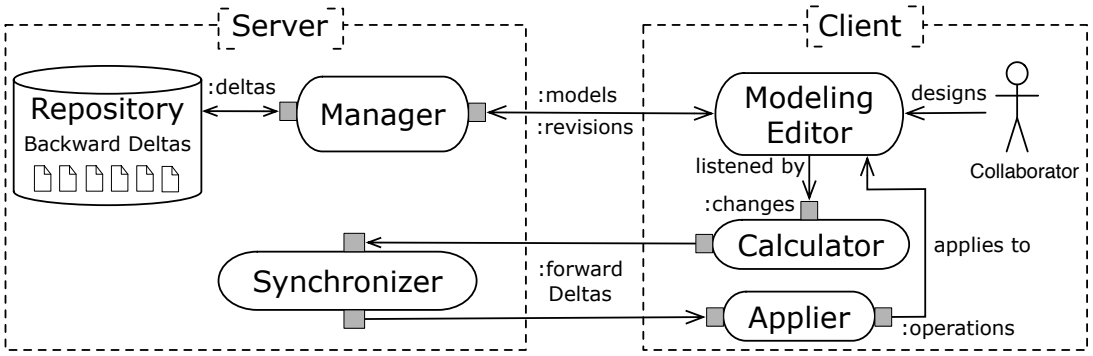

FIGURE 7.1: Reference Architecture for Concurrent Collaborative Modeling

The reference architecture in Figure 7.1 describes the main operational scenarios that collaborators may perform during concurrent collaborative modeling. The architecture is separated into *server* and *client* parts. The server side of the architecture serves as the *single point of truth* in collaboration. The overall architecture is a *blueprint* (or guide) to orchestrate the DL services in order to develop certain concurrent collaborative modeling applications. However, the DL service orchestrations are not required to rely strictly on this architecture, i.e., the architecture can be adopted or customized accordingly. Hereafter, the main DL services making up the reference architecture are briefly discussed.

Server Side. The server side of the architecture depicts two main DL services namely model *manager* and *synchronizer*, as well as the *repository* to store the DL-based modeling deltas.

- *Synchronizer.* In the concurrent collaborative modeling applications, models are always shared among several collaborators and they simultaneously change their models in real-time. These collaborators are usually located in different places and synchronization of changes must be done among these collaborators through network without delays or other inconsistencies. Eventually, the rapid synchronization of every single change made by collaborators in real-time is quite significant for the on-line collaboration of several collaborators. The high performance of change synchronization is successfully achieved by synchronizing DL-based forward modeling deltas. These modeling deltas represent changes made by collaborators on their modeling editors as depicted on the client side of the architecture. The *DL synchronizer* service hosted on the server side takes care of synchronization of modeling deltas among collaborators. It receives the DL-based modeling deltas detected by the *DL calculator* service on the client side and sends them back to the *DL applier* service of other collaborators. As long as the synchronization of modeling deltas among clients carried out by the

synchronizer on the server, the communication between collaborators is provided based on *star-topology*.

– *Manager.* While developing shared software models in concurrent collaborative modeling, collaborators intends to store models and their revisions in repositories, so that software models and their revisions are available in repositories to load, revert and restore. For instance, collaborators may want to store a particular state of their model if they feel that state of their model is complete and correct. The working copies and revisions of shared models are then stored for further development and maintenance. After all, collaborators should be able to handle various models and load any older revisions of their models which they have saved earlier. The *DL manager* service on the server side is dedicated to perform aforementioned model management activities such as creating new models in the repository, storing or loading models and their revisions, or deleting existing models, etc. The DL manager service stores models and their revisions in the *repositories* on the server. These models are stored as *active deltas* and their revisions are as *backward deltas* in terms of DL (as discussed in Section 5.2). More applications of the DL manager service is explain in Chapter 8 in detail.

Client Side. On the client side of the reference architecture, collaborators design their copies of shared models using *model editors*. The *DL calculator* and *DL applier* services have to be installed on these modeling editors in order to be able to collaboratively work with the DL-based concurrent modeling infrastructure.

- *Calculator.* In case of the concurrent collaborative modeling in centralized environments, collaborators make changes on their instances. Every single change made by collaborators needs to be identified and detected in order to propagate them on other parallel copies. Eventually, all copies remain up to date. Especially, identifying and producing these changes is quite problematic and reluctant in case of models with several thousand modeling artifacts. Thus, either *change listener* or *state-based matching* feature of the *DL calculator* service can be employed for detecting changes made by collaborators on modeling editors. The DL calculator service on the client side represents detected changes in the DL-based *forward deltas*.
- *Applier.* After detecting changes, they are delivered to other clients of shared models by synchronizer as forward deltas. These changes are propagated on other parallel copies of models so that they are kept up to date in real-time. The *DL applier* service on the client side is used to apply the DL-based forward deltas to models.

The concrete architectures of concrete DL-based concurrent modeling applications in Section 7.2 and Section 7.3 are developed based on the reference architecture depicted in Figure 7.1.

7.2 Kotelett

The DL-based collaborative modeling tool entitled `Kotelett` is developed by the students' project group in the Software Engineering Group at the Carl von Ossietzky University of Oldenburg [Project Group, 2014]. This concurrent collaborative modeling tool takes advantage of the DL-based delta representation for synchronizing changes among various collaborators in real-time. The DL application `Kotelett` is developed for collaboratively modeling on UML class diagrams. Section 7.2.1 depicts the meta-model of UML class diagrams that is used for generating a specific DL for class diagrams. Section 7.2.2 portrays the concrete architecture of the concurrent collaborative modeling application `Kotelett` based on the reference architecture depicted in Figure 7.1. Section 7.2.3 explains the `Kotelett` tool in detail.

7.2.1 Meta-Model

UML class diagram [Raumbaugh et al., 2004, pp. 47ff] is one of the most popular UML diagrams. It is usually used in modeling object-oriented software systems as well as modeling real-world systems in a object-oriented manner. They directly serve as documentations for the object-oriented software systems and real-world systems. Executable software systems can be automatically generated from class diagram models. Moreover, they are extensively used in designing the meta-models of modeling languages.

As DL is conceptually a family of domain-specific languages and generic approach, a specific DL is generated from the meta-model of UML class diagrams using the *DL generator* service (explained in Section 6.1). In order to apply the DL-based collaborative modeling infrastructure, the DL generation is required as initial step. Figure 7.2 depicts the meta-model of UML class diagram which is utilized to realize the DL-based concurrent collaborative modeling application `Kotelett`.

The meta-model in Figure 7.2 is separated into two parts by a dashed line. Below the line, it depicts the *content part* which is used to represent the subset of the modeling concepts of UML class diagram. In graphical modeling, every modeling object has design information such as color, size, and position, also called layout information. Above the dashed line, the figure consists of the *layout part* that is used to depict the notation of layout information for the content part. The complete meta-model is used for creating collaborative modeling application `Kotelett` throughout this section.

In the meta-model portrayed in Figure 7.2, every modeling artifact can be of type the *KNode* linked to the *BoundingBox* or *KRelationship* linked to the *GraphicalEdge*. Both, edges and boxes belong to the *Diagram*, whereas the *Diagram* itself is of type *ModelElement*. Each edge has the *BendPoint* and *LabelPosition*. According to the *ModelElement* class of the meta-model, each model element has

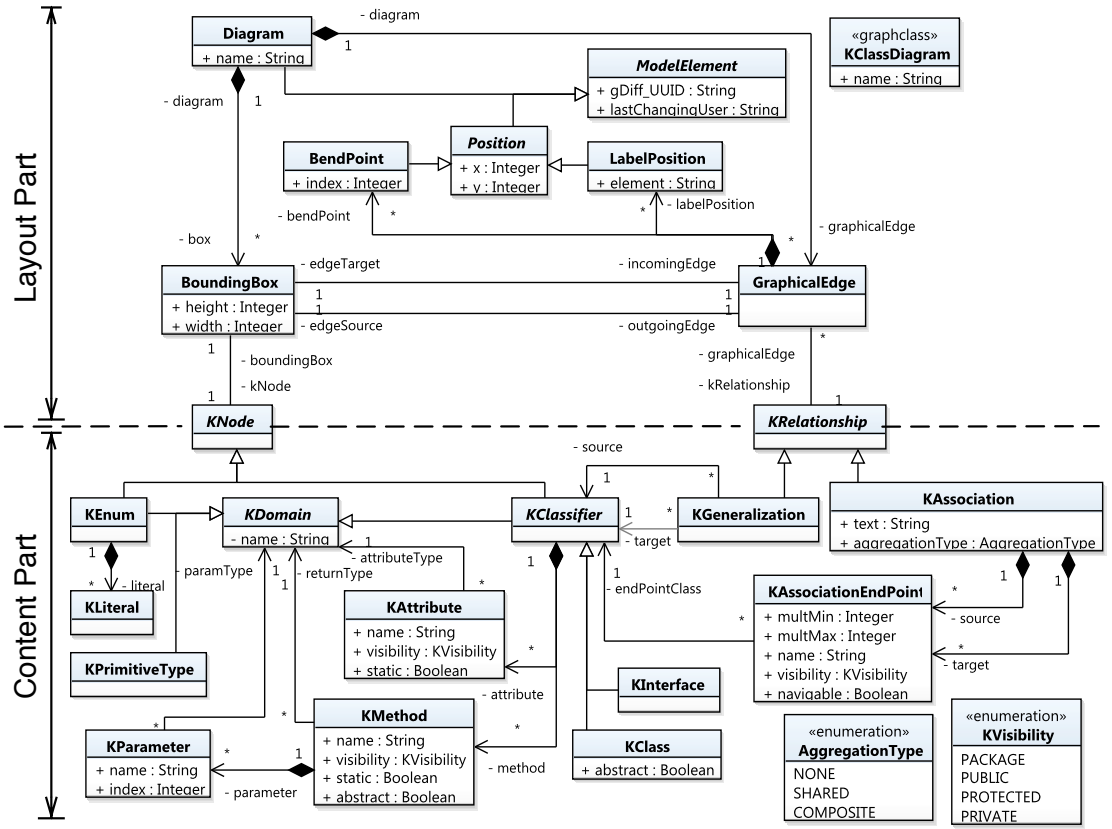

FIGURE 7.2: UML Class Diagram Meta-model for Kotelett

an attribute named *gDiff_ UUID* which means all modeling artifacts are assigned to UUIDs.

Usually, layout information is not depicted in the standard meta-models (i.e., profiles) of modeling languages. In `Kotelett`, the meta-modeling approach is also used for handling the data structures of layout notation. This allows for using the same technique for representing and synchronizing changes in content data (i.e., abstract syntax) and layout data (i.e., concrete syntax). This art of designing meta-models is another advantage of the approach. Because, in case of concurrent collaborative modeling, modeling editors require layout information to display modeling artifacts on their modeling editors. The layout part of the meta-models allows for representing layout information (in modeling deltas) for graphical editors by the DL operations. Additionally, layout information represented by the DL operations are exchanged among several parallel tool instances. For instance, if the position or the size of a modeling artifact is changed on one tool instance, the values of position and size are changed on the other parallel tool instances, as well. This can be seen in the screen-shot of the `Kotelett` tool in Figure 7.4.

The layout part of the meta-models in the DL approach enables extend-ability of the approach for further modeling languages. In order to extend the DL applications for further modeling languages, the content part (below the dashed line) of the meta-model depicted in Figure 7.2 should be replaced by the respective meta-models of other modeling languages. Eventually, the same layout information of

the meta-model can be reused for extending the collaborative modeling framework for further modeling languages.

As long as the DL services behind the `Kotelett` tool are realized using the *JGraLab* technical space, the meta-model depicted in Figure 7.2 is designed using the domain-specific modeling tool *Rational Software Architect (RSA)* [Leroux et al., 2006] and imported into the TGraph schema using the *JGraLab* technical space. The *DL generator* service generates a specific DL from this meta-model for concurrent collaborative modeling of UML class diagrams. As explained in Section 6.1, a newly generated specific DL for UML class diagrams consists of *Model API* (including interfaces and their implementations) and *Model Utility* in Java. The resulting *Model API* is used to process modeling artifacts, whereas *Model Utility* is employed to do various operations on the instance models and their artifacts. The conceptual idea of generating specific DLs is explained in Section 5.1 in detail.

7.2.2 Concrete Architecture

Figure 7.3 depicts the overall concrete architecture of the `Kotelett` tool. This concrete architecture is the concrete implementation of the reference architecture depicted in Figure 7.1. Thus, the DL services in the concrete architectures are defined as *components*, whereas they are defined as *services* in the reference architecture.

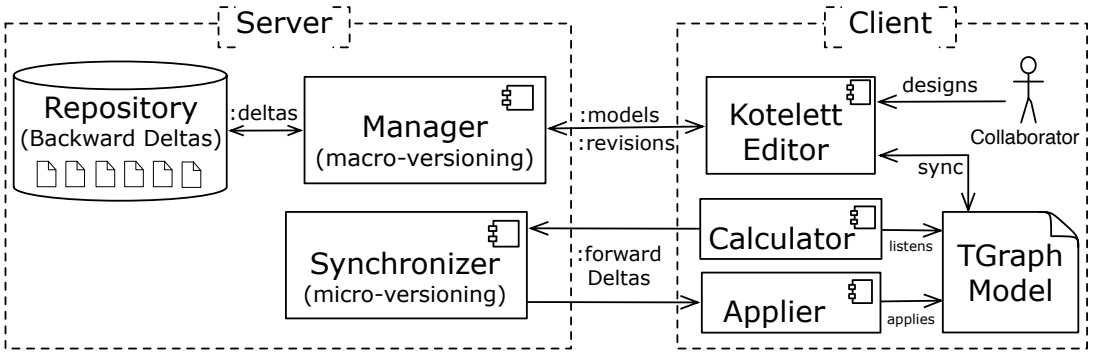

FIGURE 7.3: Concrete Architecture of Concurrent Collaborative Modeling `Kotelett`

Like the reference architecture, the concrete architecture is separated into the *server* and *client* sides.

Server. As described in Section 7.1, the server side of the architecture depicts the DL services *manager*, *synchronizer*, and *repository* to store the DL-based backward modeling deltas. The *manager* component in the concrete architecture is the concrete implementation of the *DL manager* service, whereas *synchronizer* component is the concrete implementation of the *DL synchronizer* service. The manager service is realized using the `JGraLab` technical

space (cf. Section 2.4) for this particular application `Kotelett`. The manager service takes advantage of the *DL applier* service to load models and their revisions. The synchronizer service is realized in a way that is completely independent from any technical spaces.

Client. The client side of the concrete architecture depicts the DL *calculator* and *applier* services, as well as *Kotelett modeling editor* and the *TGraph* representation of models under collaboration. The *Kotelett editor* is developed based on the `EMF` technical space using EMF-based *GMF – Graphical Modeling Framework* [GMF, 2018] for graphical editing. The modeling editor part allows collaborators to design their models visually. Below, this section explains the graphical user interface of the application in detail.

Software models within this particular tool are represented by *TGraphs* (explained in Chapter 2.4) internally. Each client embeds the *TGraph* representations of models. There is a bidirectional synchronization between TGraph-based and graphical representations of models. Once the user makes changes on model copies on their model editors, these changes are propagated on the TGraph representation of that model. If any change is applied to the TGraph representations of models by the *applier* service, these changes are interpreted on the model editor, as well. The *DL calculator* service is attached to each client side of the `Kotelett` tool in order to compute changes on the TGraph-based models. In *Kotelett*, the DL calculator calculates modeling deltas comparing the changed and unchanged revisions of models based on the *ID-based model matching*. The DL calculator computes *forward modeling deltas* in this case. These forward modeling deltas are then synchronized among other collaborators via the synchronizer service hosted on the server side. Once the modeling deltas are delivered to other clients, they are applied to TGraph-based models, whereas changes on the TGraph-based models are also interpreted on graphical model editors. The same scenario occurs for all other clients as well. All synchronizations are done by the **synchronizer** component on the server side. Each client of the `Kotelett` tool also embeds meta-model depicted in Figure 7.2 in order to ensure correctness of model copies on these clients.

This concrete architecture is used in developing the `Kotelett` tool in Section 7.2.3.

7.2.3 Kotelett Tool

This section explains the main features of the collaborative modeling tool `Kotelett` in detail. Figure 7.4 depicts the screen-shot of the overall user interface of the `Kotelett` tool. The figure displays two independent tool instances working on the same model in parallel. Each user interface (IU) consists of several windows such as *History Menu (A)*, *Model Tree (B)*, *UML class diagram elements (C)Model Editor Area (D)*, *User List (E)*, and *Log (F)*. When the tool is launched, it shows the list of models which are currently available in the repository and asks the

user which model to join as a collaborator. However, the users can open multiple models simultaneously during concurrent collaborative modeling.

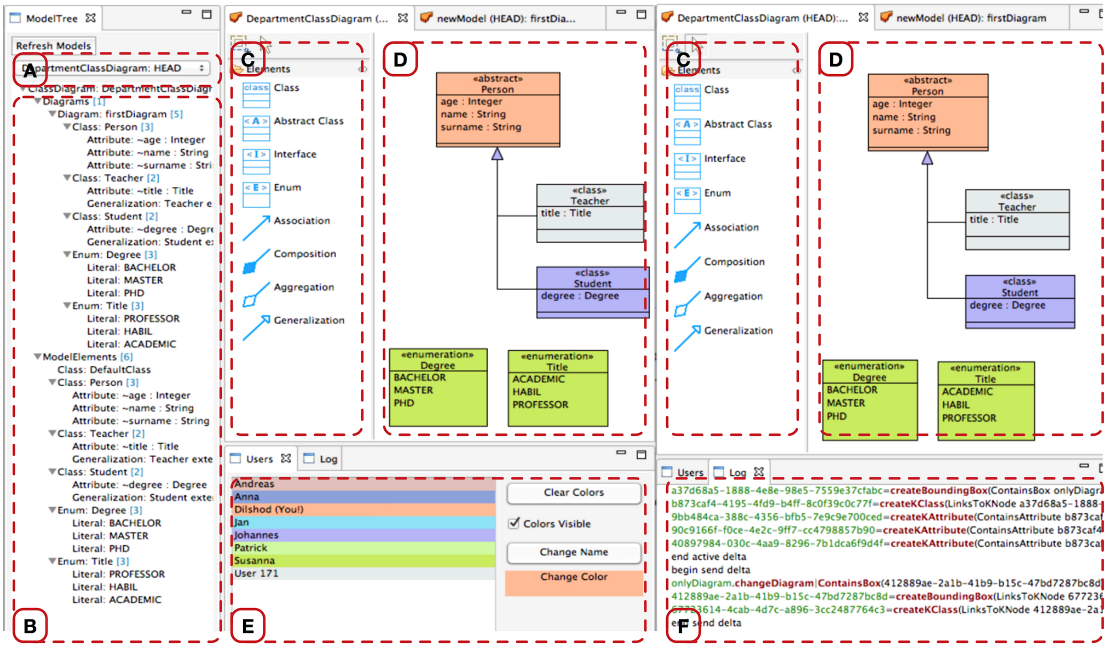

FIGURE 7.4: Screenshot of `Kotelett` Tool

– *Model History (A).* This window shows all available revisions (in repository) of currently active model. It lists all automatically and manually saved revisions of the current model. By selecting the necessary revision from the menu, it can be opened in the editor area. The *DL manager* service is utilized to revert other revisions behind this feature.

– *Model Tree (B).* On the most left side, the UI shows the list of diagrams a collaborator is currently working on. Each diagram belongs to a specific model. It also shows the list of model artifacts that are created in their relevant diagrams.

– *Modeling Concepts (C).* This area of the tool depicts the UML class diagram notations where the collaborators can select and draw that element on the model editor. These notations of the UML class diagrams are created based on the meta-model depicted in Figure 7.2. The correctness of the instance models on this editor is checked according to that meta-model, automatically.

– *Model Editor (D).* The model editor area is the main part which allows users for designing the UML class diagrams in the graphical editor. Several modeling editor tabs can be opened at the same time. As shown in the model editor, each collaborator is given a specific color so that the model elements created is highlighted with that given color.

– *User List (E in left instance).* This window lists all collaborators that are currently working on the diagram in the activated tab. These collaborators are

highlighted with different colors in order to show clear distinction between them and recognize which change is made by which collaborator on the editor. As the graphical editor displays, the modeling artifacts are highlighted with the same color of the creator of that model element. If a model element is created by one collaborator and changed by another, color of the last collaborator changed is applied to that model element. Additionally, each collaborator can change their names and select necessary color (E) that should appear on the `Kotelett UI`. In order to allow a particular user for distinguishing himself from other collaborators, he can see additional text `"You!"` right next to his name.

– *Log (F in right instance).* The log window constantly displays the modeling deltas (Figure 7.5) that are exchanged among collaborators after each change. Creating one modeling artifact on the graphical modeling editor may result in one or many change DL operations that are contained in one modeling delta and synchronized with others.

– *Configuration (E).* Once each collaborator is joined concurrent collaborative modeling, that collaborator is given a name and specific color. The collaborators can configure the their part of the user interface such as changing color, name and clearing colors.

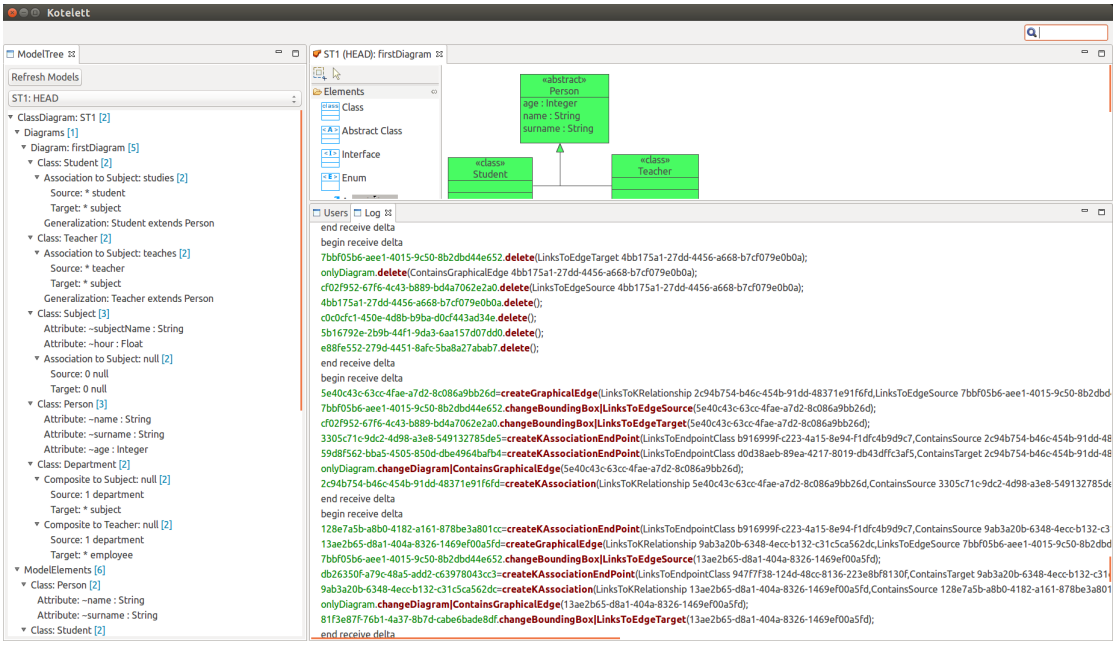

FIGURE 7.5: DL-based Change Representation in Kotelett Tool

As mentioned above, model changes are represented by DL operations and exchanged by modeling deltas consisting of these DL-based change operations. Figure 7.5 displays the zoomed screen-shot of the *Log (F)* window of the `Kotelett` tool. Thereby, the DL operations are always listed representing what changes are made and exchanged among collaborators.

As shown in the log window, each modeling delta is isolated with the *begin send delta* and *end send delta* messages which means sending a delta is started and

finished, respectively. Other tool instances receive these deltas as change requests to apply on their models.

7.3 Collaborative Modeling – CoMo

There are several open-source domain-specific modeling tools such as EMF-based `UML Designer` [Obeo Network, 2017], `Papyrus` [Lanusse et al., 2009], etc. As the DL-based collaborative modeling infrastructure intends to be meta-model and tool generic, it is applied to Sirius-based [Viyović et al., 2014] domain-specific modeling tool `UML Designer`. This section demonstrates the application results of DL to the UML Designer tool.

As discussed in Section 2.4, Sirius [Viyović et al., 2014] is an Eclipse Foundation project that provides opportunity to develop domain-specific graphical modeling tools. Sirius itself is developed based on EMF (Eclipse Modeling Framework) [Steinberg et al., 2008] and GMF (Graphical Modeling Framework) [GMF, 2018]. The graphical modeling tools developed using Sirius consist of a collection of different editors (charts, tables, trees). These editors are described by models. In EMF-based Sirius, all model-related data is stored as EMF models in the form of XML Metadata Interchange (XMI), whereas the `JGraLab` technical space stores models as *TGraphs* in case of the `Kotelett` application.

`UML Designer` [Obeo Network, 2017] is a Sirius-based open source modeling tool. The UML Designer provides possibility to design several UML diagrams. The collaborative modeling application entitled `CoMo` – *Collaborative Modeling* is developed as an extension for UML Designer. `CoMo` takes advantage of the DL-based modeling deltas for synchronizing model changes among the collaborators of the shared models. The `CoMo` application is applied to UML activity diagrams. Section 7.3.1 presents the meta-model combining UML activity diagram content and layout parts which is used in developing *CoMo*. Section 7.3.2 depicts the `CoMo` concrete architecture conforming to the reference architecture in Figure 7.1. Section 7.3.3 demonstrates the `CoMo` tool itself.

7.3.1 Meta-Model

UML activity diagrams [Raumbaugh et al., 2004, pp. 95ff] are frequently used in modeling business processes, activities and work-flows. Computational and organizational process are usually modeled using activity diagrams representing the overall control and object flows.

A specific DL for UML activity diagram is generated by the *DL generator* service (explained in Section 6.1) importing the meta-model depicted in Figure 7.6. Then, the model changes in modeling deltas are represented in terms of DL on the instance activity diagrams.

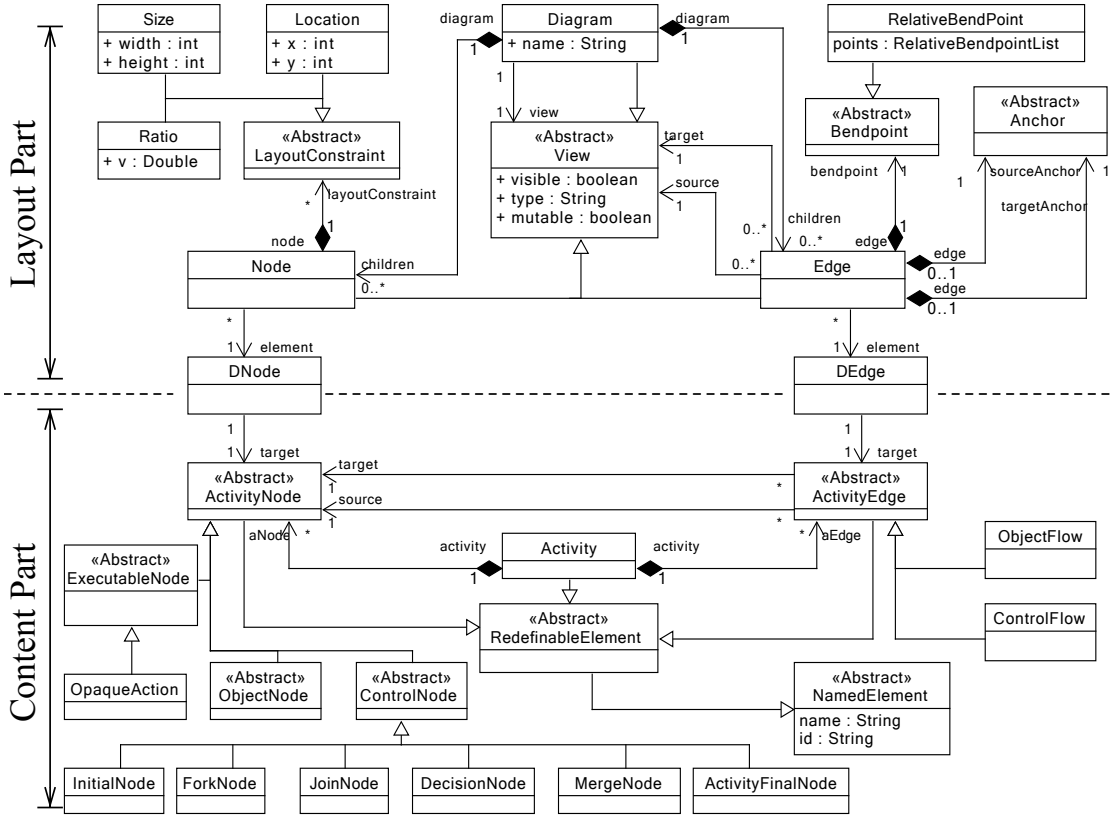

FIGURE 7.6: UML Activity Diagram Meta-model for `CoMo`

Like the UML class diagram meta-model presented in Figure 7.2, the meta-model depicted in Figure 7.6 is separated into two parts by a dashed line. Below the line, it depicts the *content part* (i.e., abstract syntax) which is adopted from the standard UML activity diagram meta-model for EMF (Eclipse Modeling Framework) [Steinberg et al., 2008]. Above the dashed line, Figure 7.6 portrays the *layout part* (i.e., concrete syntax). The layout part of the meta-model depicts the substructure of Graphical Modeling Framework (GMF) notation [GMF, 2018] which supports notations for developing visual modeling editors based on EMF. The *ActivityNode* and *ActivityEdge* of the content part are connected to the *Node* and *Edge* of the layout part through the *DNode* and *DEdge* artifacts of the Sirius *odesign* notation.

As long as `CoMo` in Section 7.3.3 is realized based on the EMF technical space, the meta-model depicted in Figure 7.3.1 is designed using the EMF-based Ecore meta-modeling feature.

This way of designing meta-models allows for using the same collaborative modeling infrastructure for different modeling contents within the same technical space. The complete meta-model is used for creating overall collaborative modeling application `CoMo` explained in Section 7.3.3.

7.3.2 Concrete Architecture

After generating the specific DL for the given meta-model, namely the UML activity diagram meta-model (incl. Content and Layout parts) in this case, the DL-based collaborative modeling infrastructure is used to handle collaboration activities for UML activity diagrams in UML Designer. It is built by the specific amalgamation of the several DL services as depicted in Figure 7.7. It is the concrete architecture based on the reference architecture in Figure 7.1.

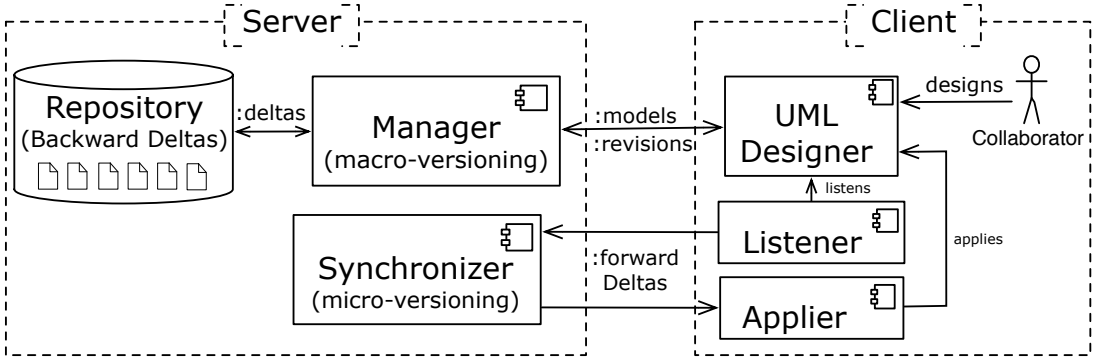

FIGURE 7.7: Concrete Architecture of Concurrent Collaborative Modeling `CoMo`

Like the reference architecture, the concrete architecture is separated into the *server* and *client* sides.

Server. As described in Section 7.1, the server side of the architecture depicts the DL services *manager* (enabled by *macro-versioning*), *synchronizer* (enabled by *micro-versioning*), and *repository* to store the DL-based backward modeling deltas. The *manager* component in the concrete architecture is the concrete implementation of the *DL manager* service, whereas the *synchronizer* component is the concrete implementation of the *DL synchronizer* service. The most implementation parts of the manager service is taken over and reused as they are. Its implementation is slightly changed to be useful in this particular application `CoMo`, whereas the synchronizer service is directly utilized in `CoMo` without any adaptations, as it is independent from the technical spaces. The manager service takes advantage of the *DL applier* service to load models and their revisions.

Client. The client side of the concrete architecture depicts the *listener* feature of the DL calculator service and *applier* services, as well as *UML Designer* as modeling editor. In the `CoMo` application, the *change listener* feature is employed instead of the *state-based matching* of the DL calculator. The changes made by collaborators are constantly detected by the change listener while they are made using the model editor, in this case, UML Designer. These changes are then represented in forward modeling deltas and constantly sent to other parallel clients through the *synchronizer* service on the server. Once these deltas arrive at other clients, they are applied to models by the *applier* service.

In case of the EMF-hosted UML Designer, the listener listens for *Notifications* which consists of information about the changed modeling artifacts (i.e., resources) and the change types. As discussed in Section 6.3, the change types in these notifications are mapped to the DL change types.

The concrete architecture depicted in Figure 7.7 is utilized as the concrete blueprint in development of the `CoMo` application in Section 7.3.3.

7.3.3 CoMo Tool

This section explains the collaborative modeling application `CoMo` (`C`ollaborative `Mo`deling) of DL. It is developed by the specific orchestrations of the DL services, on top of the DL-based delta representation, based on the concrete architecture in Figure 7.3.2 and the meta-model depicted in Figure 7.3.1.

The `CoMo` support is developed as an extension for Sirius-based domain-specific modeling tool UML Designer. Figure 7.8 depicts a screen-shot of `CoMo`. It displays two different tool instances working on the same model concurrently. These tool instances describe the micro-versioning scenario depicted in Figure 5.7 after the changes are synchronized. Each `CoMo` tool instance consists of several windows as explained below.

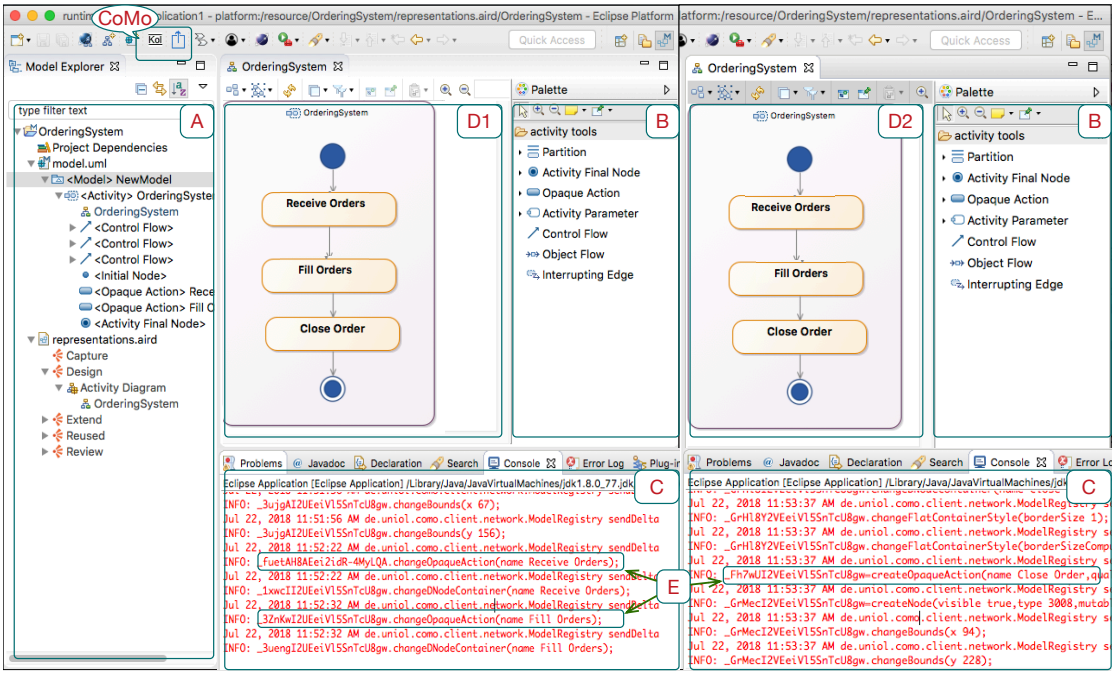

FIGURE 7.8: Screenshot of `CoMo` Tool

Figure 7.8 actually depicts the user interface of UML Designer. As long as the DL-based collaborative modeling approach is applied to UML Designer which is an EMF- and Sirius-based domain-specific modeling tool, `CoMo` is completely realized using the EMF technical space.

– *CoMo Support (CoMo in left instance).* After installing the `CoMo` support, two buttons appear in the tool as shown on the left instance under the indicator `CoMo`. When the first button (*Kol*) is clicked, the list of models currently available in the repository is displayed asking the user which model to join as a collaborator. From the displayed dialog window, collaborators can either select an existing model from the list or create a new model in the repository. If they select to join an existing model as a collaborator, that model is opened in the editor (D) and they can continue further developing that model. The users can open multiple models at once during collaboration.

  Models under collaboration can be saved by clicking the save button whenever the model is complete and correct. When the tool is asked to save the model by clicking the second button under the indicator `CoMo`, it calculates the differences (backward deltas) between the last and active revisions. Furthermore, new active deltas are also generated when the new revisions of models are stored. As the result of each click, one backward delta (representing differences between the last and active revisions) and one active delta (representing the working copy) are stored in the repository. This feature of the `CoMo` tool is currently under development.

– *Model Tree (A in left instance).* The model tree (A) shows the list of models and diagrams (incl. the elements of these diagrams) the collaborators are currently working on.

– *Modeling Concepts (B in both instances).* The both instances display the modeling concepts of the UML activity diagrams (B). These concepts conform to the meta-model depicted in Figure 7.6.

– *Model Editor (D1 and D2 in both instances).* The model editor (D1 and D2) areas on the both instances show the example activity diagrams that `Designer_1` and `Designer_2` are developing as depicted in Figure 5.7. In this case, these both instances are displayed after their changes are synchronized. The modeling editor provides the *Redo/Undo* features of *Transactional Command Stack* to revert changes on the editor.

– *Logger Window (C in both instances).* The logger window (C) constantly displays the modeling deltas that are exchanged among collaborators once they are synchronized. Creating one modeling artifact on the graphical modeling editor may result in one or many change operations that are contained in one modeling delta that is synchronized between collaborators.

– *DL-based Changes (E).* The letter E indicates three changes made by two designers. These changes are: the name changes from `"Receive Order"` and `"Fill Order"` to `"Receive Orders"` and `"Fill Orders"`, respectively (highlighted on the left logger), as well as the creation of new *Opaque Action* named `"Close Order"` (highlighted on the right logger).

Modeling deltas on the logger windows (C) are represented by the specific DL generated from the combined meta-model in Figure 7.6 including the standard

UML profiles (content part, i.e., abstract syntax) and GMF notation (layout part, i.e., concrete syntax). The DL change listener and applier services are extended using the EMF technical space features such as the command stack and resource set listener extensions for the editing domains. All other underlying technologies such as the DL synchronizer and model manager remain unchanged.

7.4 DL Contributions

During experiments, the both DL applications `Kotelett` [Kuryazov et al., 2018] and `CoMo` [Appeldorn et al., 2018] have shown sufficiently high performance by synchronization of small DL-based modeling deltas. So far, they have not faced any change conflicts in concurrent collaborative modeling. This probably is attributed to the rapid synchronization of small modeling deltas with compact syntax. Thus, concurrent collaborative modeling enabled by the micro-versioning currently does not focus on the issue of conflict resolution. In the current implementations of concurrent collaborative modeling applications, the most recent changes are propagated on all parallel instances of models.

Establishing the concurrent collaborative modeling applications is a challenging task because of its real-time performance, change synchronization and complexity of software models. However, the DL-based delta representation approach brings a number of advantages to the concurrent collaborative modeling and aims at resolving these challenges.

With the DL-based modeling delta representation approach, software models are made commonly available to multiple collaborators in different locations and communication is provided by synchronization of *small DL-based modeling deltas*. Model changes on the shared software models are rapidly detected and synchronized in real-time allowing users to communicate without delays. It enables users to cooperate their changes into the centralized project instantly, regardless the network speed, and the complexity, large-scale of their shared software models.

The DL approach solves several additional technical challenges with developing concurrent collaborative infrastructure for MDSE. It can be applied to the wide range of modeling languages and modeling tools. It also supports its tooling *modeling as a service (MaaS)*.

Specific DLs can easily be generated using the modeling concepts of various domain-specific languages. Eventually, any service, component or plug-in required for collaborative modeling can easily be developed on the top of that modeling language enabling language and tool genericity, extend-ability, and re-usability properties. DL brings several advantages to the concurrent collaborative modeling scenario, as follows:

– *Single Underlying Meta-model and API.* The meta-models depicted in Figure 7.2 and Figure 7.6 have two parts separated by the dashed line. This way of

designing meta-models allows for using the same collaborative modeling environment for different modeling contents. The layout notation is usually not depicted in the content of meta-models. The combined meta-models allow for using the same DL-based delta representation technique for representing, storing and synchronizing the layout data of models by DL operations.

DL is generic with respect to the meta-models of modeling languages. The combined design of meta-models provides to generate single underlying API for the given meta-models. For instance, in case of the `CoMo` tool, the existing model APIs of GMF, EMF and Sirius can be used to work with instance models in collaborative modeling. But, the tools developers need to deal with three types of APIs, whereas the DL generator generates only one API for working with UML activity diagrams.

– *Changes Only.* Only changed modeling artifacts are represented in DL-based modeling deltas. The unchanged modeling artifacts are not included in modeling deltas simply not defining DL operations for them. In turns, it provides small modeling deltas which result in quick synchronization of changes among collaborators.

– *No Conflicts.* The concurrent collaborative modeling applications are experimented by the users located in long distance in Canada, Mozambique and Uzbekistan, whereas the server was located in Oldenburg, Germany. During these experiments, the tool has not shown any inconveniences with conflicts, performance, agility, and amount of collaborators. The tools have not faced any change conflicts because of rapid synchronization of model changes by small DL-based modeling deltas. Change synchronization was fast enough to exchange all changes before conflicts may occur, so far.

– *Undo/Redo Operations.* In case of `Kotelett`, the *redo/undo* operations with changes are easily performed in real-time because of the small DL operations. In `CoMo`, the DL change listener is realized using the *Resource Set Listener* which listens for the *Transactional Editing Domain* of Sirius sessions. The use of transactional editing domain provides to perform the redo/undo operations without extra implementation effort.

– *Single Point of Truth.* DL serves as a common difference representation and exchange format for the various modeling tools, components and services of collaborative modeling. For instance, the both concurrent collaborative modeling applications `Kotelett` and `CoMo` can use the same DL-based modeling delta repositories. The same underlying repository can be used as the single point of truth without any further adaptations or implementation efforts.

– *Separation of Representations.* In `Kotelett`, clear separation between the graphical editor and internal graph representation of models did not result in any inconveniences in collaborative work. The collaborative modeling has been quite efficient and fast. All changes are made on the graph-like representations of models and these changes are then synchronized with graphical

editors. Operating on the graph-like structure of models is quite fast, for example, on calculating and applying changes. This separation of internal model representations can be the additional advantage of the approach in developing further collaborative modeling applications by just developing graphical editors and reusing the same graph-like model representations together with collaborative modeling support.

– *No Model Locking.* As surveyed in [Altmanninger et al., 2009], several collaborative development approaches utilize the *lock-modify-unlock* concept during collaborative development and evolution. The *lock-modify-unlock* model is usually used for avoiding possible change conflicts that might occur during collaboration. Unlike these approaches, the collaborative modeling application of DL enables collaborators to work on shared software models without locking their models. In the DL-based concurrent collaborative modeling, conflicts are less likely to occur because of high synchronization performance provided by small modeling deltas, as discussed above.

7.5 Summary

This chapter has presented the DL-based concurrent collaborative modeling applications enabled by the concurrent model versioning scenario (i.e., *micro-versioning*). In these applications, the DL approach is applied to the concurrent collaborative modeling applications to `Kotelett` for modeling UML class diagrams and to `CoMo` for modeling UML activity diagrams based on UML designer. The collaborative modeling application `Kotelett` itself was developed by a project group of students in Software Engineering Group at the Carl von Ossietzky University of Oldenburg. The approach is applied to UML designer in the framework of bachelor thesis [Appeldorn, 2018]. The collaborative modeling tools are used in Software Engineering lectures for teaching purposes by a group of students including more than ten collaborators in parallel. The `Kotelett` tool is successfully presented by the project group on a study exhibition day for school children.

The meta-models depicted in Figure 7.2 and Figure 7.6 support representation, storage and synchronization of both layout information (i.e., concrete syntax) and modeling language notation (i.e., abstract syntax) by the same underlying DL. The both applications can be extended for further modeling languages by extending or replacing the content part of the meta-models. The layout part remains unchanged. If the graphical modeling editors can be redeveloped for further modeling languages, all other underlying technologies and services such as the DL calculator, applier, manager, synchronizer and DL-based delta representation remain the same and do not require much implementation effort.

Chapter 8

Sequential Collaborative Modeling

As discussed in Section 3.2, software models evolve over time undergoing various changes in order to meet user changes and improvements. Evolution of software models results in the several revisions of the same modeling artifact differing from each other. During development and evolution of models, artifact changes have to be properly detected, identified, stored, maintained and reused for further analysis and manipulations.

The main objectives of *sequential collaborative modeling* are concerned with maintaining the historical archive and managing the evolution of modeling artifacts [Altmanninger et al., 2009]. A huge number of modeling artifacts undergo various changes which have to be controlled in large and complex software models. In developing and maintaining the large-scale and complex models, the distributed team of developers aim at storing the history of evolving modeling artifacts and managing complex software models with multiple subsequent revisions. The histories of evolving software models are usually managed by sequential collaborative modeling that is also known as *model version control* in some literature [Altmanninger et al., 2007], [Taentzer et al., 2012], [Swicegood, 2008].

There are several sequential collaborative modeling tools like Subversion [Collins-Sussman et al., 2004], Git [Swicegood, 2008], Monotone [Hoare et al., 2005] and many more for developing and maintaining source code-driven software projects. As long as these tools deal with textual artifact representation, it is commonly agreed (as discussed in Section 3.2) that their difference representation techniques can not provide sufficient information in different representation of graph-like software models serialized as eXtendable Interchange Format (XMI) [Cicchetti, 2008], [Steinberg et al., 2008].

Chapter 4 has investigated several approaches dedicated the problem of sequential collaborative modeling, for instance, *EMF Store framework* [Helming and Koegel, 2013], *SMOVER* (Semantically enhanced Model Version Control System) [Altmanninger et al., 2007], *AMOR* (Adaptable Model Versioning System) [Langer, 2011].

These tools are dedicated to sequential collaborative modeling of EMF-based software models [Steinberg et al., 2008]. They provide several standard sequential collaborative modeling activities.

In sequential collaborative modeling, it is properly effective and convenient to store the history of artifact revisions in form of *difference documents* which are also referred to as *modeling deltas* in this thesis. Especially, in MDSE, the efficient representation of modeling deltas allows for easy storage and reuse of difference information in further development and evolution. DL introduced in this thesis is a reasonable means to represent the change histories of evolving software models in modeling deltas. It is effective and reasonable foundations for sequential collaborative modeling in storing the histories of software models in modeling deltas. This chapter explains the sequential collaborative modeling application of DL, enabled by *macro-versioning*.

This chapter firstly explains the general repository architectures and scenarios regarding sequential collaborative modeling in Section 8.1. The same section depicts a reference architecture for developing sequential collaborative modeling architectures based on the DL representation and explains several DL services that are involved in developing the reference architecture for sequential collaborative modeling. Section 8.2 and Section 8.3 introduce the sequential collaborative modeling applications of DL. This chapter identifies several contributions of DL to sequential collaborative modeling in Section 8.4. Section 8.5 draws some conclusions for this chapter.

8.1 Reference Architecture

In Section 3.2, the existing sequential collaborative modeling approaches for both source code-driven and model-driven software development are discussed in detail. The existing approaches take advantage of diverse principles, technologies, repository and storage architectures, and models. This section revisits these underlying architectures, principles and technologies utilized by the existing sequential collaborative modeling systems in order to identify and derive principles, storage and repository architectures and models that can be reused for the sequential collaborative modeling application in this chapter. This section also gives the detailed description of the reference architecture for DL-based sequential collaborative modeling.

As already defined in Definition 3.3 in Section 3.2, sequential collaboration is used for identifying differences between subsequent revisions, storing them in delta documents, and reusing them when needed. These delta documents are then utilized in retrieving the revisions of modeling artifacts, analyzing the evolution histories of modeling artifacts under development and evolution [Glasser, 1978]. The basic idea behind sequential collaborative modeling (i.e., revision control) is to preserve the histories of software models under development and maintenance.

Repository Architectures and Storage Models.
As discussed in Section 3.2, the existing sequential collaboration systems use different software repository architectures, history models and storage models regardless of the type of software artifacts. Below, these repository architectures and models are discussed that are used in the DL sequential collaborative modeling application:

Software Repository Architectures. There are two approaches for building up the architectures of software repositories [Altmanninger et al., 2009]: *Centralized* and *Distributed*. The DL sequential collaborative modeling application takes advantage of the centralized development architecture. Models under development and evolution, and their histories are stored in a single central repository. Collaborators can have the working copies cached locally.

History Models. The most classical sequential collaborative systems use two types of history models for storing software repositories. These are *snapshot-oriented* and *change-oriented*. As long as the snapshot-oriented approach stores the whole revision (*snapshot*) of software projects as the first-class object, it is very costly, ineffective, memory and time consuming approach in case of software models. Thus, the DL sequential collaborative modeling application employs the *change-oriented* history model for storing the model histories in repositories.

Storage Models. As discussed in Section 3.2, the existing sequential collaboration systems use diverse repository storage models depending on their implementation details. As this thesis uses the *delta-based* (i.e., *change-oriented*) storage model, the DL sequential collaborative modeling application takes advantage of *delta algorithms*. This is dedicated to compare the subsequent revisions of software models or listening for user changes and to create *modeling deltas* as discussed in Section 6.3. The DL optimizer (Section 6.8) service partly realizes the *delta combination* approach [Hudson, 2002], [Proceedings, 2006]. While applying modeling deltas to models, the chain of modeling deltas are emerged (concatenated) into a singe delta, then applied by the DL applier service. This approach dramatically increases performance of the delta application process. Similar technique is entitled *skip-delta* [Hudson, 2002] or known as delta concatenation.

Reference Architecture.
The most basic architectural and operational concepts as well as terminologies of source code-driven sequential collaboration systems are applicable to the sequential collaborative modeling application, as well. The architectures of the code- and model-driven sequential collaboration applications can be considered as similar. Moreover, their main operating scenarios are analogous, e.g., *add*, *checkout*, *commit*, *update*, *revert* and *delete* can be used to manage models under collaborative development.

Figure 8.1 depicts a reference architecture for the DL-based sequential collaborative modeling applications. It presents four main operative scenarios of sequential

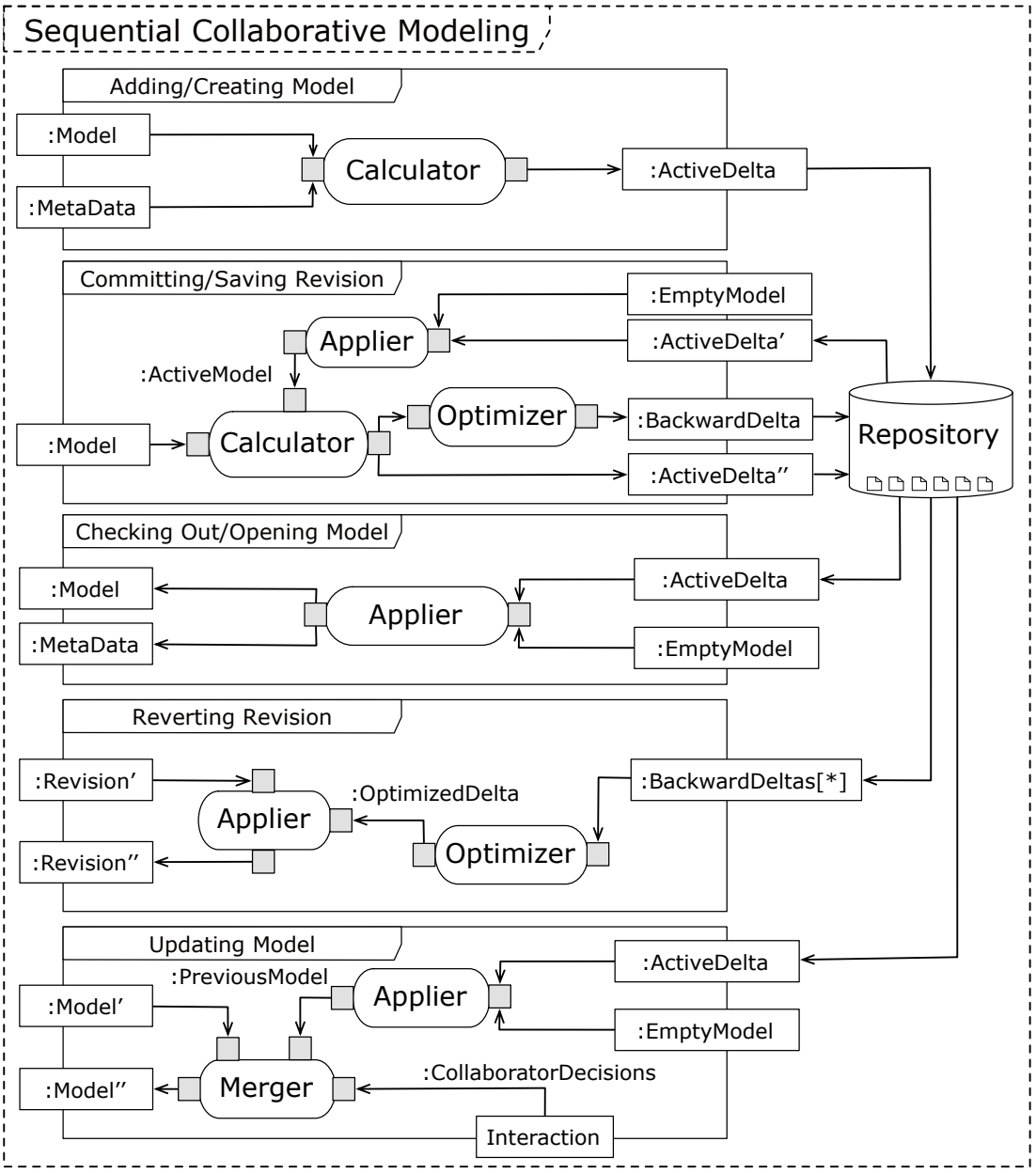

FIGURE 8.1: Reference Architecture for Sequential Collaborative Modeling

collaboration, namely, *adding/creating model*, *committing/saving revision*, *checking out/opening model* and *reverting revision*. In order to perform these operative scenarios, the relevant DL services are orchestrated in each scenario. Below, these orchestrations in the reference architecture are explained in detail.

Adding/Creating Models. Before starting sequential collaboration process with software models, models have to be *added* to the collaborative modeling applications or new models have to be *created*. After adding to or creating models in collaborative development system, their artifacts are usually identified as the initial revision. Then, the histories of these modeling artifacts are stored and maintained during further development and maintenance. Eventually, while software models under collaboration are being developed

and maintained by developers, their change histories are stored in repositories keeping them persistent. For adding existing models or creating new ones, the *meta-data* about these models should be provided. The meta-data may consist of the names of new models, additional comments, etc. Eventually, active deltas are generated by the *DL calculator* service and stored in the repositories, in initial step.

Committing/Saving Revisions. After making changes on the working copies or branches of models, collaborators commit/save their local changes into the main development lines of their models. In order to perform this scenario, models have to be given as input. Then, the *DL applier* service creates the previous revision of that model by applying relevant *active delta* to an *empty model*. After having the given (current) and reverted (previous) revisions, the *DL calculator* service compares these model revisions to compute one *backward delta* representing differences between these revisions and one *active delta* to store the given model revision in the repository. When storing *backward deltas* in the repository, they are optimized by the *DL optimizer* service according to optimization requirements discussed in Section 6.8.

Checking Out/Opening Models. If collaborators intend to open or check out models from the repository, this scenario invokes the *DL applier* service. Thereby, the DL applier fetches an active delta for the requested model and creates the model out of that active delta. Collaborators are able to copy/check out (co) any model from the repository into their local working spaces in order to further develop that model. These working copies of the models are usually entitled to be the parallel copies or branches of the central model.

Reverting Revisions. In case of the lost or damage of information on software models, collaborators intend to revert their models to older, correct revisions. To perform the reversion scenario, the *DL applier* service receives a relevant model that should be reverted. A list of *backward deltas* are fetched from the repository related to the given model. These deltas are then concatenated by the *DL optimizer* service in order to skip some delta operations if possible. After all, the DL applier service applies these chain of backward deltas to the given model and reverts the requested revision of that model. The amount of backward deltas in the set depends on which revision has to be reverted. For instance, if the tenth revision should be reverted out of twenty, then the delta list consists of ten backward deltas.

Updating Models. In sequential collaborative modeling, collaborators may have different development branches or copies in distributed environments. These development branches or copies are developed by different collaborators resulting in different copies the same software model. Before *committing* the local changes into the main repository, the local working copy must be updated (up) in order to obtain recent changes from the repository made by others. If another collaborator has already committed other changes to the main repository before this initial commit, all other collaborators have to

update their local copies with the changes in the main repository so that the changes in the main repository are fetched and available in all parallel copies or branches. The merge operation, in turn, requires to resolve conflicts if there are any during merging. The *DL merger* service can be utilized to merge different revisions of models. There, the DL merger service obtains the last revision of models by applying the relevant *active* to an *empty model*. Simultaneously, it receives the given model revision as the second input. Eventually, as the result of merging, it produces the merged revision of that model.

While merging different revisions (with different changes) of the same model, change *conflicts* between revisions might arise. In case of conflicts, they are classified into two classes such as *resolvable* and *unsolvable*. The resolvable conflicts are the conflicts that can be automatically resolved by the *DL merger* service (as discussed in Section 6.9) without requiring any user interaction. The unsolvable conflicts require interaction of collaborators in deciding which revision to accept or reject in final revision, or postpone for resolving later. However, the latter is planned as the future work.

Further Scenarios. In addition to the aforementioned orchestration scenarios, there are further management operations provided by the DL-based collaborative modeling infrastructure. These are *getting model list* that are available in the repositories, *getting model revision list* of a particular model, or *deleting a model* from the repository.

The aforementioned orchestration scenarios are the core objective of the DL approach in developing its sequential collaborative modeling applications. These scenarios are developed and fulfilled by the specific orchestrations of the DL services in this chapter.

8.2 Generic Model Versioning System – GMoVerS

As the sequential collaborative modeling is another prominent application of DL, the DL approach is applied to application entitled *Generic Model Versioning System (GMoVerS)* in this section. Like the concurrent collaborative modeling application, the sequential collaborative modeling application is established by the specific orchestrations of the DL services based on the reference architecture depicted in Figure 8.1. `GMoVerS` is developed on top of the main scenarios of sequential collaborative modeling explained in Section 8.1. Section 8.2.1 explains what kind of modeling languages `GMoVerS` can handle. The concrete architecture of this sequential collaborative modeling application is defined in Section 8.2.2. Section 8.2.3 clarifies the concrete realization of the `GMoVerS` tool.

8.2.1 Meta-Model

The DL sequential collaborative modeling application is enabled by *macro-versioning*. This application is usually dedicated to the problem of managing software models and their revisions. Whenever specific DLs are generated, all DL services are capable of operating based on that DLs. Consequently, the sequential collaborative modeling application developed by the specific orchestrations of the DL services operates on top of the DL-based modeling repositories.

The sequential collaborative modeling application `GMoVerS` can be used as either a standalone tool or embedded behind the concurrent collaborative modeling applications (in Section 7) for managing DL-based modeling delta repositories. If it has to be used as a standalone tool for sequential collaborative modeling, specific DLs have to be generated by the *DL generator* service only for sequential collaborative modeling. The *DL manager* service in the reference architecture of the concurrent collaborative modeling applications is also provided by the `GMoVerS` application. If it has to be used as an embedded model manager behind a concurrent collaborative modeling application, one common DL can be generated for the both collaborative modeling applications at once.

As `GMoVerS` is used behind the *DL manager* service, it is applied to the meta-models used in `Kotelett` (Section 7.2) and `CoMo` (Section 7.3). In this case, `GMoVerS` is used as embedded tool. Thus, the DLs for both concurrent collaborative modeling applications `Kotelett`, `CoMo` and sequential collaborative modeling application `GMoVerS` (i.e., *DL manager*) are generated altogether.

As experimental illustration, `GMoVerS` is applied to UML state machine diagrams [Raumbaugh et al., 2004, pp. 81ff]. Figure 8.2 depicts the simplified substructure of the meta-model for UML state machine that is utilized for validation purposes. The simplified meta-model depicted in Figure 8.2 presents a *state* that might be one of simple state with a name, *initial* or *final states*, *transitions* including *guard*, *trigger* and *action*.

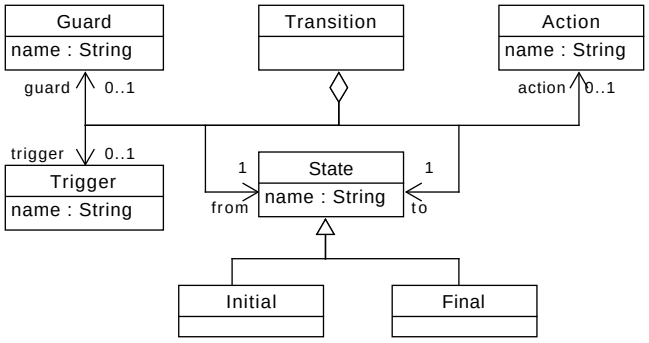

FIGURE 8.2: Simplified Substructure of UML State Machine Meta-model

8.2.2 Concrete Architecture

`GMoVerS` stores the DL-based modeling deltas in its delta repositories. In turns, it is developed by the specific orchestrations of the DL services that can directly operate on the DL-based modeling delta repositories. Figure 8.3 illustrates the concrete architecture of `GMoVerS` based on the reference architecture depicted in Figure 8.1.

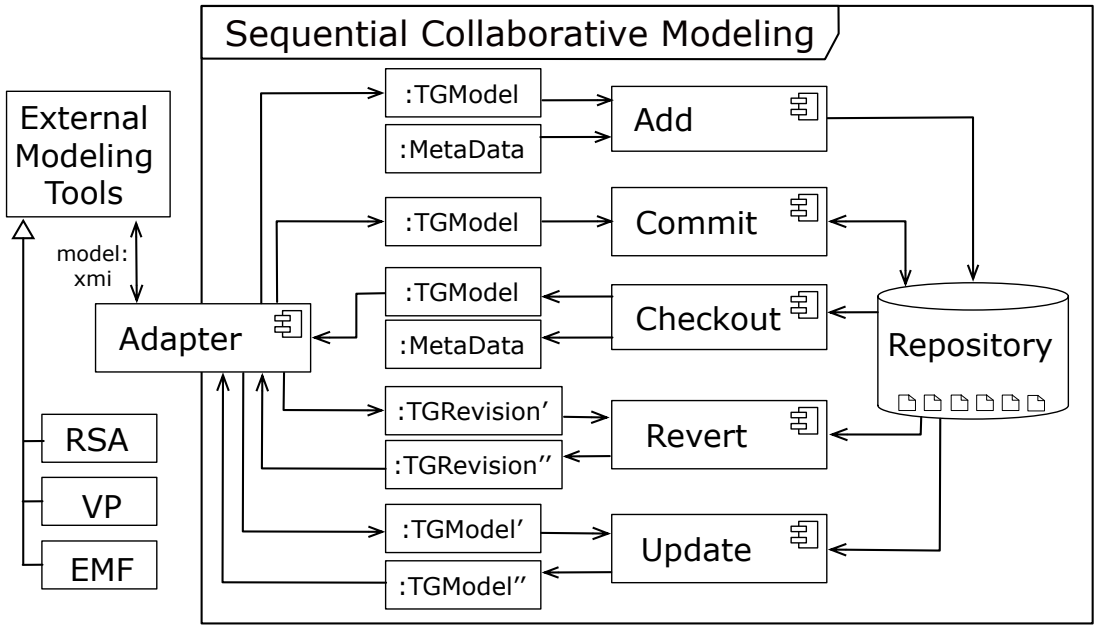

FIGURE 8.3: Concrete Architecture of `GMoVerS`

The concrete architecture in Figure 8.3 considers all orchestration scenarios of sequential collaborative modeling such as *adding*, *committing*, *updating*, *reverting* discussed in Section 8.1.

The DL-based concurrent and sequential collaborative modeling applications can also be used either in a combined way or independently. The concrete architecture in Figure 8.3 considers independent use of `GMoVerS` from the other DL-based concurrent collaborative modeling applications. However, `GMoVerS` can also be utilized as the DL-based model repositories together with existing modeling tools such as *Rational Software Architect (RSA*, *Visual Paradigm (VP)* or EMF-based tools. Thus, the concrete architecture further depicts the *DL adapter* service to enable integration of `GMoVerS` with other modeling tools. The DL adapter service is utilized to convert models in exchange formats (e.g., XMI exported from these tools) into internal TGraph models (*TGModel* or *TGRevision*) and vise verse. Because, `GMoVerS` processes software models using TGraph (cf. Section 2.4), internally.

`GMoVerS` is also used together with other DL concurrent collaborative modeling applications to provide the model management features. All model management features provided by the *DL manager* service are provided by this sequential collaborative modeling application. In case of the *DL manager* service, the same

concrete architecture in Figure 8.2.2 is utilized without the *DL adapter* service. As long as sequential collaborative modeling is used as the model manager for the concurrent collaborative modeling applications, models are created directly using the concurrent collaborative modeling editors.

8.2.3 GMoVerS Tool

Figure 8.4 displays a screenshot of the `GMoVerS` development environment.

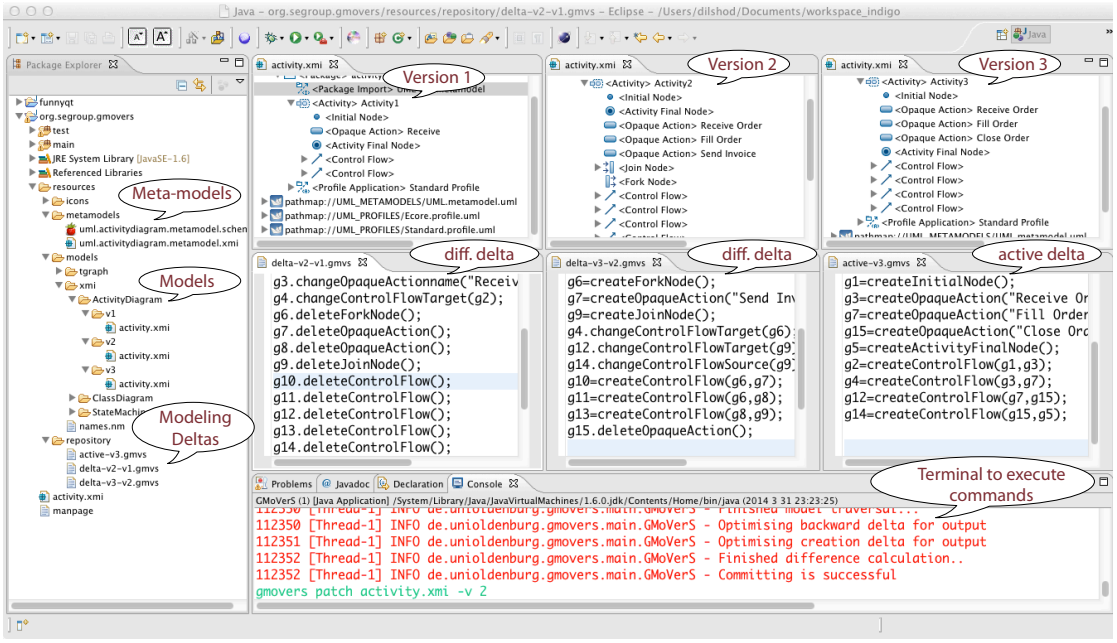

FIGURE 8.4: `GMoVerS` Screenshot

The package explorer on the left shows arrangement of the working space including the *meta-model* in Figure 7.6, the *models* and *modeling deltas* in the example in Section 5.2. Particularly, all DL-based modeling deltas are stored in the `GMoVerS` repository. On the upper row of the right side, Figure 8.4 displays three subsequent revisions of the example model in Figure 5.7 in the exchange formats. The central row of the right shows two *backward modeling deltas* and one *active delta* in Figure 5.12, Figure 5.8 and Figure 5.9, respectively. Finally, the most bottom of the screenshot is a terminal (command line) to type and perform aforementioned operative scenarios such as add, commit, update, revert, delete, etc.

While committing changes from workspace onto the main repository, conflicts might occur between differentiated models. In such cases, conflicts have to be detected when they arise and resolved either automatically (if possible) or semi-automatically. If the user involvement is needed in case of the semi-automatic way, the sequential collaborative modeling system has to provide interactive conflict resolution feature by browsing conflicts. The merge feature is provided by the *DL merger* service. However, a feature for the interactive resolution of conflicts is planned as the future work.

8.3 Versioning Sustainability Reports

Sustainability reports are documents which describe data about different performances of companies. They intend to report and analyze sustainability information to provide sustainable future. During sustainability development, reports have to be stored and versioned to analyze the histories of sustainability reports, to represent changes and improvements of sustainability data. Sustainability reporting allows companies to document their performance on specific sustainability issues by measuring, tracking, and monitoring companies information such as economy, environment or social impacts [Kuryazov et al., 2013].

Sustainable Online Reporting Model (STORM) [Solsbach et al., 2011] which is developed at the University of Oldenburg aims at dialogue-based public or private sustainability reporting which intend to engage stakeholders in sustainability development. While maintaining sustainability reports, STORM stores all required information in its relational database and obtains various reports by appropriate requests. The storing procedure used in STORM may lead to "flood of information" within the system. Another issue which appears in case of database approach is to revise reports in a way which is easy to analyze the revision histories.

The DL-based difference representation approach was applied to versioning sustainability reports at companies [Kuryazov et al., 2013]. Versioning sustainability reports does not directly belong to the MDSE domain. However, the core problem is still similar and can be resolved by the DL-based model versioning technique. Because, sustainability reports at companies are also subject to constant changes and evolution. The changes in sustainability reports are represented by DL and version control is supported by `GMoVerS`.

Section 8.3.1 illustrates the meta-model (i.e., schema) of sustainability reports to generate a specific DL.

8.3.1 Meta-Model

According to the data structure of sustainability reports, version control of reports and associated data in STORM is not as simple as version control of textual documents. By looking at sustainability reports and associated data, the data structure of them can be viewed as a sustainability meta-model. The problem of version control of sustainability reports is a similar issue to version control of sustainability models. In case of STORM, reports are stored in a database based on schema which is, at some point, inconvenient to provoke delta between the revisions and analyze the version histories. Taking these issues into consideration, the DL approach was applied to STORM sustainability models.

To derive a specific DL for representing differences between the report revisions, the approach has utilized the schema of sustainability reports depicted in Figure 8.5. As `STORM` takes advantage of relational databases to store its sustainability reports, this schema is designed based on the database schema of `STORM`.

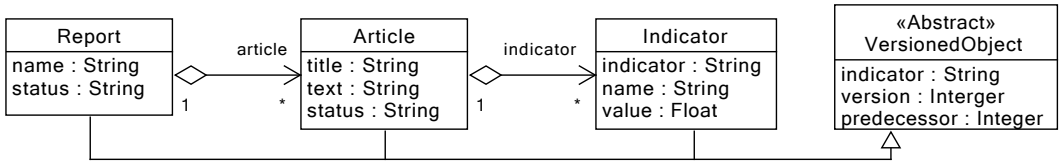

FIGURE 8.5: Simplified Schema of Sustainability Reports

According to the schema (i.e., meta-model for DL) in Figure 8.5, a *report* (with *name* and *status*) may consist of several *articles*. In the same vein, each *article* (with a *title*, *text* and *status*) consists of multiple *indicators* (with *indicator*, *name* and *value*). In order to provide the DL-based versioning of sustainability reports conforming to this schema, each *report* is considered as a *Versioned Object*, whereas each versioned object has a time-stamp *date*, revision number *version* and indicator *predecessor* to trace the predecessor object of each versioned object. In this way, differences between each subsequent pairs of revisions are identified as one modeling delta.

The simplified schema in Figure 8.5 presents only an excerpt data schema of the sustainability report within `STORM`. The complete `STORM` schema conforms the GRI C3 standard.

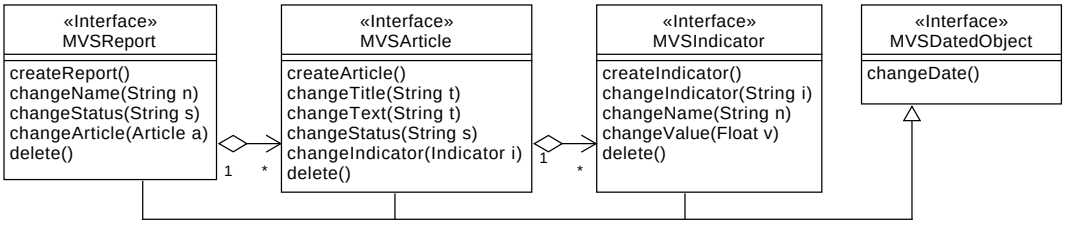

FIGURE 8.6: Abstraction of DL for Sustainability Reporting

The schema depicted in Figure 8.6) shows the abstract DL notation to represent modeling deltas in `STORM`. As the DL approach is meta-model generic, it is applied to meta-model-like schema. The *DL generator* service imports the data schema of sustainability reports (cf. Figure 8.5) and generates a sequence of operations which represents a minimal set of possible activities to store report versions (cf. Figure 8.6). The operations represent only differences in modeling deltas. Modeling deltas, in turn, allow for identifying, analyzing and reusing the revisions of sustainability reports in convenient ways.

8.3.2 Concrete Architecture

The sequential collaborative modeling scenario of sustainability reports is developed based on the reference architecture of DL depicted in Figure 8.1. The most sequential collaborative modeling scenarios are covered in versioning sustainability reports. The similar DL-based modeling delta repositories are utilized for storing

the backward deltas in case of the DL-based sequential sustainability report versioning. The same concrete architecture of `GMoVerS` depicted in Figure 8.3 is used to develop the sequential version control application for `STORM`.

8.4 DL Contributions

As discussed in Chapter 3.2, sequential collaborative modeling is quite crucial for storing and managing the histories of evolving software models. The storage of evolving software models and their revision control is a challenging research question. Therefore, the DL approach in this thesis is utilized in developing a solid and competitive sequential collaborative modeling application.

The `GMoVerS` application of DL can be applied to a wide range of modeling languages by providing the meta-models of modeling languages (*RQ1: Meta-model Generic*). As long as the DL services are not strongly consolidated to the `GMoVerS` tooling environment, they can be optimized, extended with additional features or completely replaced by other implementation as well (*RQ3: Extensible*). Further DL services can be realized and integrated into the DL service orchestrations. They are capable of handling the DL-based modeling deltas.

Modeling deltas consist of DL operations to represent only changed modeling artifacts *RQ8: Delta-based*. The unchanged modeling artifacts are not included simply not defining DL operations for them. The DL-based representation of backward deltas in sequential collaborative modeling forms directly the executable descriptions of model differences, i.e., the DL-based modeling deltas can directly be applied to models in order to transform them into earlier revisions *RQ7: Executable*.

The DL-based sequential collaborative modeling support can be used as either an independent model version control system or integrated with existing DL-based concurrent collaborative modeling applications. It can further be integrated with external modeling tools by the *DL adapter* service (*RQ2: Modeling Tool Generic*). It is also utilized in the DL concurrent collaborative modeling support `Kotelett` and `CoMo` as the *DL manager* in the server side. The generic DL-based representation is used as a common change representation notation and exchange format for all revision control-specific orchestrations described in Section 8.1. The both, concurrent and sequential collaborative modeling applications can take advantage of the same underlying DL-based modeling delta repositories as *Single Point of Truth*.

The DL and its sequential collaborative modeling application is further applied to represent the changes of sustainability reports which is not directly in the MDSE domain.

8.5 Summary

This chapter has presented the DL-based sequential collaborative modeling application `GMoVerS`. The operation-based, textual DL approach facilitates a difference representation notation for representing differences between the subsequent revisions of software models in case of sequential collaborative modeling.

DL introduced in Chapter 5 is applied to `GMoVerS` in Section 8.2 using the specific orchestrations of the DL services explained in Chapter 6. The same sequential collaborative modeling support is applied to version sustainability reports in `STORM`.

In the DL-based sequential collaborative modeling applications, differences between the subsequent revisions of software models are represented by the DL-based modeling deltas. The use of DL-based modeling deltas in `GMoVerS` and `STORM` provides high performance in all orchestration scenarios of sequential collaborative modeling and the efficient storage of model differences. Because, the DL-based modeling deltas embody minimal and relevant information about model differences and form directly the executable descriptions of model differences. High efficiency and performance of the sequential collaborative modeling is significant contribution of the DL approach especially because of the large-scale, complex and constantly evolving software models.

Chapter 9

Model History Analysis

As discussed in Chapter 7 and Chapter 8, software models with a large number of revisions and artifacts are developed, managed and maintained using concurrent and sequential collaborative modeling applications. In case of both applications, collaborators feel a need for analyzing the model histories, comprehending and understanding what changes are made by other collaborators or know how their models are evolving. Also, observing the model history and its evolution process assists the users in making important decisions in maintenance of their model-based software projects. Considering these concerns and significance, this chapter explains model history analysis application of the DL approach.

The model history forms overall evolutionary life-cycle of software models. The issue of model history analysis consists of two sub-challenges that have to be resolved. *Firstly*, information about the model histories have to be mined from the model repositories where modeling artifacts are produced and archived during the evolution process. *Secondly*, model history data that is mined has to be properly displayed to collaborators using the browsing and visualization techniques as discussed in Section 3.3. These two features contribute to efficient analysis of the evolutionary life-cycle of software models with a particular focus asking questions such as *who? why?*, *when?* and *what?*.

This chapter discusses the reference architecture for the DL-based model history analysis applications in Section 9.1 and explains orchestration scenarios by revisiting the discussions in Section 3.3 about the history analysis of software systems. Section 9.2 introduces the model history analysis application of the DL approach. Section 9.3 discusses some contributions of the DL-based model difference representation to the model history analysis application. Finally, Section 9.4 sums up this chapter by drawing conclusions about the DL model history analysis application.

9.1 Reference Architecture

This section revisits the basic categories of history information, the purpose of history analysis, history analysis steps in Section 3.3. They can partially be utilized in the model history analysis application of this thesis. This section further presents a reference architecture and the service orchestration scenarios for DL-based model history analysis.

Infrastructure.
Like other DL applications, several construction technologies are required in building an infrastructure for model history analysis. For this purpose, this section takes advantage of already existing architectural foundations and patterns of software history analysis discussed in Section 3.3.

Information Resources (Software Repositories). As long as the DL-based concurrent (e.g., `Kotelett`, `CoMo`) and sequential (e.g., `GMoVerS`) collaborative modeling applications utilize the common DL-based modeling *delta repositories* (cf. Figure 9.1) for storing and archiving software model histories, the DL model history analysis application takes advantage of the same common repository as its information resource.

Information Categories. As classified by [Kagdi et al., 2007], there are three basic categories of information in software repositories that can be mined: *model revisions* – the sequential and concurrent states of software models, *differences between revisions* – the differences (modeling deltas) between the revisions of software models, and *meta-data about model differences* – the commit messages, user-ids, time-stamps, and other similar information. The meta-data of model changes are stored in modeling delta repositories in this thesis, as well. In developing the DL-based collaborative modeling infrastructures, all aforementioned data is stored in the central modeling delta repositories.

In general, the problem of model history analysis can be leveled into two common objectives: (1) representation of model differences in modeling deltas and (2) extracting necessary information from these repositories [Robbes, 2007]. Tracing the history of any modeling artifact is heavily depend on the representation techniques of models, modeling deltas, and identification of differences and references (inter-model and delta-model).

Reference Architecture.
Figure 9.1 depicts the reference architecture for DL-based model history analysis.

According to [Robbes, 2007], the history analysis of software models consists of several sub-activities as follows:

Change Representation. Change operations are usually distinguished in two forms: *atomic operations: artifact creations, artifact deletions, artifact's*

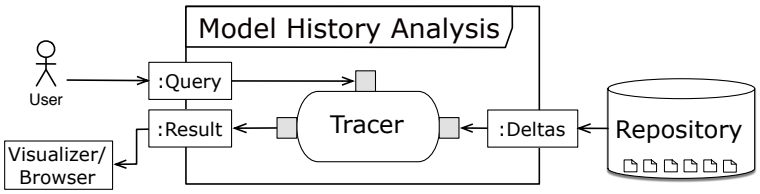

FIGURE 9.1: Reference Architecture for DL-based Model History Analysis

property changes and *composite operations: moves* [Robbes, 2007]. As introduced in Section 5, DL represents all possible model changes by the atomic operations *create, delete, change* and stored in DL-based modeling deltas repositories. Model changes are represented in modeling deltas that stored in the *repositories*.

Data Extraction. This thesis has introduced the DL tracer service in Section 6.7 which helps to extract model history information from the DL-based modeling delta repositories. The reference architecture depicted in Figure 9.1 utilizes the *DL tracer* service to trace the chain of *modeling deltas* and extract necessary data based on the *queries* provided by *users*.

Browsing, Visualization. After extracting necessary history information from the repository based on the given queries, that information has to be browsed or visualized in convenient ways, so that users can analyze the visualized data. The visualization/browsing usually depends on the users in which ways they want to interpret these history data. Thus, this part of the reference architecture is flexible and clearly separated from the rest of the reference architecture. Section 9.2 demonstrates the visualization and browsing techniques that are used in the realizations of the DL-based model history analysis application.

Querying. The model history application is usually supposed to operate based on the queries that users might ask. Below, several questions are listed that the DL model history analyze application is capable of answering. Designers and stakeholders might ask various questions based on their needs, as well as for extracting necessary knowledge about the change history of modeling artifacts. In order to analyze the histories of whole model or to trace particular artifacts of evolving models, collaborators need to determine answers to several questions as follows:

1. How often does a modeling artifact change?
2. When was a modeling artifact created?
3. When was a modeling artifact deleted?
4. Which modeling artifacts are constantly changing?
5. How does the history of a modeling artifact look like?
6. How were the states of a whole model in earlier versions?
7. What are the differences between any two versions of a model?
8. Who made a particular change?
9. Which modeling artifacts are associated to a modeling artifact? etc.

There are several questions that can be answered by analyzing *meta-data* stored in the repositories. For instance, the questions formulated with *why* are usually answered by analyzing *commit messages* in the repositories. The questions formulated with *when* are usually answered by considering the *time-stamps* stored as the part of meta-data.

The questions formulated with *who* can answered by creating an additional attribute in the root meta-class of meta-models. For instance, in case of the `Kotelett` tool, the root meta-class *ModelElement* of the meta-model depicted in Figure 7.2 has an attribute *lastChangingUser* which is used to store the names of users that changed modeling artifacts. In this way, the most questions formulated with *who* can be answered.

These questionnaires are also partly addressed by Wenzel et al. [Wenzel, 2008, Wenzel and Kelter, 2008]. These questions are listed in a generalized form and addressed to by the model history analysis of the DL approach in this chapter.

9.2 Model History Analysis

As long as the DL operations refer to modeling artifacts by their persistent identifiers, *connected subsequence* (sometimes called *Traceability Links* [Meier and Winter, 2018]) of history information (correspondences) can easily be detected from the chain of modeling deltas. By using these associated change operations in modeling deltas, the history of any selected artifact in any model revision can be traced back or forward by eliciting necessary information like change type, modeling concept, etc. Back tracking of associated operations leads to the creation point (revision) of the selected modeling artifacts, whereas forward tracking leads to the working copy of modeling artifacts. Eventually, all history information between the creation point and working copy (evolutionary life-cycle) of modeling artifacts is available for analysis. As already described in Chapter 5, the DL operations consist of *reference part* which allows for *identifying* (model-delta) and *tracing* (inter-delta) model changes by the global and persist UUIDs.

As explained in Chapter 6.7, this research work provides the *DL tracer* service which allows for tracing a specific (set of) modeling artifacts and gather required information about the selected modeling artifact(s). The *DL tracer* service makes use of the idea of *program slicing* technique [Weiser, 1981] for detecting the necessary slice of change operation from modeling deltas. The slicing process continues until it finds all necessary operations by matching the references between modeling artifacts and delta operations.

The *history analysis application* – `MoHA` is built on the top of the DL change tracer (Figure 9.2). In order to detect history information based on user queries, the change tracer fetches a set of modeling deltas from the repository and runs throughout these deltas based on persistent identifiers by gathering required information from each modeling delta. The outcome of the change tracer service is the

associated (sub-)sequence of the change operations (history information) about the selected modeling artifact(s).

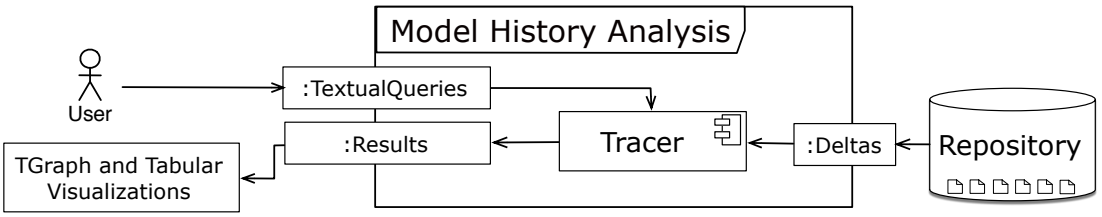

FIGURE 9.2: Concrete Architecture of Model History Analysis

`MoHA` firstly builds a history tree for each model running throughout all modeling deltas to show models (available in repository) as a general tree (A). For example, on the left side of Figure 9.3, all model revisions are outlined in a tree view (B) including all modeling artifacts and their attributes. Then, the *DL tracer* service traces the history of selected modeling artifact based on its persisted identifier. After detecting a requested change operation, the change tracer verifies the identifier of that change operation with the unique identifiers found from the successor and predecessor modeling deltas if necessary. Finally, the change tracer creates a chain of change operations where all operations are associated with each other and each operation embeds all necessary history information of the requested modeling artifact.

The screenshot in Figure 9.3 depicts the current prototypical implementation of the DL-based `MoHA`. It displays the example model from Section 5.2. It further shows a list of available models in the repository under the *Select the Model* pop-up menu (A). If any model is selected from the list, all existing revisions of the selected model are then shown in the model history tree (B), right below that pop-up menu. The model history tree shows all model revisions in a tree view including modeling artifacts belonging to each revision.

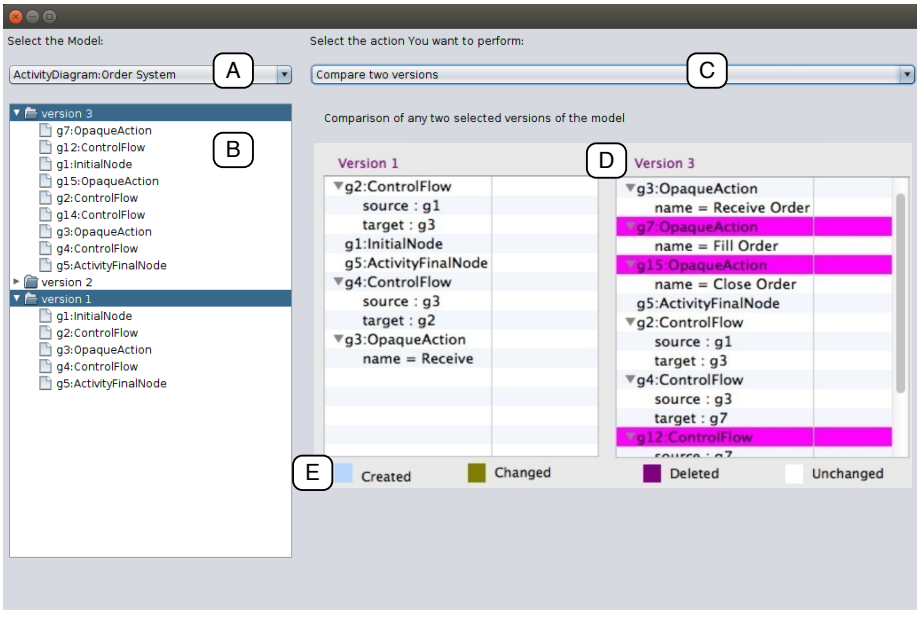

FIGURE 9.3: Screenshot of `MoHA`

The pop-up menu under the section *action you want to perform* (C) consists of all the history analysis questionnaires described in Section 9.1. In order to find out the answer for analysis questions, users can select corresponding model revision or artifact from the model history tree and the question they want find out an answer from the menu. The history report then appears in the tabular view (D) displaying history information according to user's choice. In this example, the tabular representation view (D) displays the comparison of the two revisions of the model. However, it is always adjusted to represent relevant query results, accordingly. The color area (E) shows the type of change with different colors.

`MoHA` utilizes *tabular*, *highlighting* (as depicted in Figure 9.3) and *model-based* forms of visualizations. If users intend to see the whole model in a particular revision, they can simply double-click that revision in the revision tree (B). Eventually, the double-clicked revision is opened in the *TGraph* forms.

9.3 DL Contributions

Each DL operation depicts sufficient history information about each individual model change. The reference part of the DL operations embodies globally unique and persistent identifiers, thus, model changes are easily identified by these references. The correspondences between modeling artifacts in model revisions are effectively traced, as well. Each of these correspondences defines the *occurrences of artifact changes*. All change occurrences and types they depict are traced by the *DL tracer* service. The most desired analysis questions can be answered by querying the model repository.

Based on the DL representation, advanced querying, browsing and visualization features for software model repositories can efficiently be developed. The contributions of DL to model history analysis are manifold:

- It improves tracing performance by small change operations in modeling deltas.
- It allows for tracing model changes without ignoring actual modeling concepts.
- Whole models or their certain aspects can be queried and the query results can be browsed and visualized in many different ways.
- As long as only changed model elements are referred to in modeling deltas, tracing the change histories from modeling deltas is quite effective to detect required history information.
- Analysis questions can be extended by just extending the tracer service.

`MoHA` takes advantage of the tabular view and graph visualization techniques for visualizing its query results. These visualization techniques are completely separated from the *DL tracer* service. Thus, both tracer and visualization can be further developed and extended without relying on each other.

9.4 Summary

As proof of the concept, this thesis has applied the proposed DL to the model history analysis, as well. As all three DL applications operate on the DL-based modeling delta repositories, the DL-based `MoHA` can be used alongside the DL-based `Kotelett`, `CoMo` and/or `GMoVerS` applications. Moreover, the history analysis support can also be utilized as a standalone tool.

In order to develop the DL model history analysis, the DL approach provides the *DL tracer* service. The DL tracer service can query the DL-based modeling delta repositories and its outcome is a list of associated artifact changes that can be classified based on analysis questions introduced in Section 9.1. The results of the *DL tracer* service can then be visualized in convenient ways for further analysis. The current prototypical implementations of the history analysis application (`MoHA`) visualizes these change reports using the *tabular*, *highlighting* and *model-based* views. But, these change reports can notably be visualized in any other forms which users want to see if the further visualization features are implemented.

Part V

Evaluation

One of the main goals of any research work is its validation based on different criteria and requirements. In the same vein, Section 3.1.4 has expressed the expected benefits of DL for concurrent collaborative modeling, Section 3.2.4 has explained the expected benefits of DL for sequential collaborative modeling, and Section 3.3.4 has defined the expected benefits of DL for model history analysis. Eventually, Section 4.3 has generalized these expected benefits and defined several requirements for the DL-based difference representation approach, its services and applications. The common required support for all three application areas was to have a common underlying model difference representation technique and further supplementary services to develop these applications by the specific orchestrations. To that end, Part III has introduced *Difference Language* for representing model differences in modeling deltas and a set of supplementary services to extend the application areas of DL.

This part of the thesis inspects the main contributions of DL, its services and applications based on the requirements defined in Section 4.3 and required support defined in Chapter 3.

Chapter 10

Validation

In Section 10.1, this chapter demonstrates the practical value and usefulness as well as flexibility of the DL concepts by several applications. The DL approach is validated to achieve its intended goals by fulfilling the requirements identified in Section 4.3. More specifically, these requirements are revisited in Section 10.2 for validating that these requirements are satisfied by the DL approach throughout this thesis. All DL concepts, its services and applications provide evidence to the fulfillment of these requirements. Section 10.3 inspects the DL-based difference representation approach and its main properties based on expected benefits (defined in Chapter 3) concurrent and sequential collaborative modeling, and model history analysis. This chapter is concluded in Section 10.4.

10.1 Applicability

In Chapter 5, the operation-based, meta-model generic difference language has been explained in detail illustrating the simplified example. Furthermore, Chapter 6 has introduced a catalog of the DL services that are capable of producing, manipulating and reusing the DL-based modeling deltas. The concrete applications of DL are presented in Part IV by concurrent collaborative modeling applications `Kotelett` for UML class diagrams, `CoMo` for UML activity diagrams based on Sirius-based UML Designer, sequential collaborative modeling application `GMoVerS`, and model history analysis `MoHA` as proof the DL concepts.

Modeling Languages.
In order to validate the ideas behind this thesis, several modeling languages are inspected according to their popularity among academicians, researchers and industry experts. Eventually, several UML diagrams are chosen for validation purposes in this thesis. The approach is validated in UML *class diagram* (in Chapter 7), *activity diagrams* and *sustainability reports* (in Chapter 8). UML state machine diagrams are also considered in experimental validations.

These applications are flexible and can be extended for further modeling languages with less implementation effort. DL is generic with respect to the meta-models of modeling languages. For this purpose, Section 5.1 has introduced the DL generator service. It is capable of importing the meta-models of modeling languages and deriving the specific DLs from the given meta-models. If the meta-models of modeling languages are changed, i.e., updated with new revisions, for example, standard UML 2.2 profiles are being updated with UML 2.4 profiles, these new profiles can be imported and new specific DLs can again be generated. Specific DLs can be derived as many time as needed regardless of changes on meta-models. With such feature, the approach does not require extra effort for generating new specific DLs if the meta-models of modeling languages are modified or extended with additional modeling concepts.

The meta-models can be designed combining content (i.e., abstract syntax) and layout (i.e., concrete syntax) of modeling languages. The *layout part* is used to depict notation for layout information for the content part. In the concurrent collaborative modeling scenario, modeling editors require layout information to display modeling artifacts on their editors. The layout part of the meta-models allows for representing layout information (in modeling deltas) for graphical editors by the DL operations. Additionally, layout information represented by the DL operations are exchanged among several parallel tool instances. The layout part of the meta-models in the DL approach enables extend-ability of the approach for further modeling languages and tools. In order to extend the DL applications for further modeling languages, the modeling *content part* of the meta-models should be replaced by the relevant meta-models of modeling languages. Eventually, the same layout notation can be reused for extending the collaborative modeling framework for further modeling languages, probably with very less effort.

DL Services.
All DL services explained in Section 6 are realized based on the service-oriented software development concepts [Erl, 2005]. After all, these services are orchestrated in order to perform certain chain of operations in the reference architectures of the DL applications. Thus, these DL services can be configured if needed or replaced by other implementations and/or extended with further services. Only prerequisite for these services is to recognize the syntax of DL. After adding newly implemented services or optimizing existing ones in the service catalog of DL, these services can be utilized in service orchestrations in developing and extending the DL applications. If needed, new reference architectures or orchestration scenarios can be defined by using the same DL services. These DL services can be reused in reference architectures and orchestration scenarios as many times as needed.

DL Applications.
The DL applications such as concurrent collaborative modeling `Kotelett`, `CoMo`, sequential collaborative modeling `GMoVerS` and model history analysis `MoHA` are emerged by the specific orchestrations of the DL services. Accordingly, their functionality and features can also be improved by extension of the DL services. For instance, the DL change listener and delta applier service can be extended in the

DL service catalog for other modeling tools, whereas the whole collaborative modeling infrastructure can be taken over for that modeling tools. As explained in Section 7, if the change listener feature of the DL calculator service and DL applier service are extended for new domain-specific modeling tools and new specific DL is generated for that domain-specific modeling language, the whole DL-based collaborative modeling infrastructure can be utilized for collaborative modeling, model repository management, and model history analysis.

The concurrent collaborative modeling tool `Kotelett` represent software models under collaboration using the graph-like structures, as well. The graphical editor of the `Kotelett` tool is separated from its internal graph representation. This allows for extending and improving the graphical editor of the `Kotelett` tool as much as needed without changing underlying DL services. Because the DL services calculator and applier operate on the graph representation of models, whereas there is a bidirectional synchronization between graphical editor and graph representation. This feature of the approach enables tool developers to define further graphical notations, shapes and views as they want and reuse existing collaborative modeling infrastructure.

10.2 Fulfillment of Requirements

In order to accomplish a solid, appropriate, generic and effective difference representation approach for collaborative MDSE, this thesis has identified several requirements for model difference representation in Section 4.3. These requirements have stayed as the central focus and the DL approach has attempted to satisfy these requirements throughout this thesis. This section combines and summarizes all DL features fulfilling these requirements. Below, these requirements are revisited together with the fulfillment by the DL features [Kuryazov and Winter, 2015a].

– **RQ1: Meta-model Generic.** There are several modeling languages following diverse formal specifications and modeling concepts. The abstract syntax of modeling languages, i.e., the modeling concepts are defined by their corresponding meta-models. As defined in Section 5.2, DL can be applied to the wide range of modeling languages if their meta-models are provided. Conceptually, DL is a family of model difference languages. Specific DLs are derived from the meta-models of modeling languages by the *DL generator* introduced in Section 5.1. For instance, the specific DLs for UML class diagrams (in case of `Kotelett` in Section 7.2) [Kuryazov et al., 2018], UML activity diagrams (in case of `CoMo` in Section 7.3) [Appeldorn et al., 2018] and Sustainability Reports (in Section 8.3) [Kuryazov et al., 2013]are generated by the DL generator importing their respective meta-models. After generating specific DLs for them, their changes on the instance models conforming to these given meta-models are represented by the specific DLs.

– **RQ2: Modeling Tool Generic.** There are several model designing tools (e.g., Sirius-based UML Designer, Rational Software Architect - RSA, etc.) as well and they have own internal model representation structures. To be able to handle models designed in different modeling tools, DL is not restricted to a certain modeling tool. This thesis has introduced the *DL adapter* in Section 6.2 for transforming software models from XMI exchange formats to TGraph internal structures and vice verse. For instance, the sequential collaborative modeling application *GMoVerS* is applied to version control of the models designed in RSA. Moreover, the meta-models of modeling languages can be designed for representing layout information together with the content of languages which allows for representing layout changes in DL terms. This feature of the approach extends applicability of DL in various modeling tools [Kuryazov, 2014]. If the DL calculator and applier services can be extended, the approach can be applied to various modeling tools as it is applied to UML Designer.

– **RQ3: Extensible.** DL is flexible, i.e., it can be further improved, extended, adapted and integrated. Particularly, the provided services by the approach are available for further improvements and extensions. Because, these services are realized based on service-oriented and component-based concepts, and orchestrated in the reference architectures in Section 7.1 and Section 8.1.

– **RQ4: Operation-Based.** Section 5.2 has shown simplified example of the DL-based model difference representation. In DL, artifact changes are represented using simple edit operations. The DL operations consider the composite structure of modeling concepts. DL is a special form of domain specific language being completely independent from the underlying technical spaces. DL represents all types of model changes using three atomic change operations such as *create*, *delete* and *change*.

– **RQ5: Model Reference.** In order to refer to modeling artifacts from DL operations in modeling deltas (cf. deltas in Section 5.2), DL operations embody references (i.e., UUIDs) to modeling artifacts. The references so-called *delta-model references* are used in applying modeling deltas to differentiated models and analyzing the change histories of evolving models.

– **RQ6: Expressive.** The syntax of DL operations (Section 5.2) is meaningful being easily understandable by users and is practical to implement using various technical spaces. The DL syntax is self-expressive enabling tools developers to understand it and develop further tools on the top. By looking at the DL syntax, it can easily be comprehended what kind of modeling artifact is being changed including its value and what kind of change is made.

– **RQ7: Executable.** DL operations in modeling deltas (e.g., deltas in Section 5.2) form the executable descriptions of model differences. Modeling deltas are then applicable to its differentiated models in order to transform them from one revision to another. This thesis has introduced *DL applier* service in Section 6.4 to enable applicability of modeling deltas.

– **RQ8: Delta-Based.** Only changed modeling artifacts are referred to in modeling deltas. Unchanged modeling artifacts are implicitly excluded by just not defining DL operations for them in modeling deltas. As described in Section 5.2.2, modeling deltas consist of a set of operations which refer to only changed modeling artifacts. Considering only changed modeling artifacts bring several advantages in terms of time and storage memory in the sequential collaborative modeling scenario, as well as in the synchronization performance of model changes over network in the concurrent collaborative modeling [Cicchetti et al., 2007], [Herrmannsdoerfer and Koegel, 2010].

– **RQ9: Persistent.** DL change operations in modeling deltas are persistent throughout the evolutionary life-cycle of software models. Each operation in modeling deltas can be identified by their UUIDs with respect to its successors and predecessors. It allows for maintaining and persisting each modeling artifact together with the change history. These identifiers serve for tracing the associated set (chain) of change operations from the chain of modeling delta by *inter-delta references*. Using these identifiers, the change histories of modeling artifacts are traced by the *DL tracer* service for extracting necessary knowledge for further analysis.

– **RQ10: Traceable.** The DL operations in modeling deltas are available for further reuse and utilization. DL is straightforward and accessible by the *DL tracer* service which is used for further analysis in model history analysis application `MoHA`. The change histories stored in the DL-based modeling delta repositories are analyzed by mining necessary information with the *DL tracer* service (cf. Section 6.7) [Kuryazov and Winter, 2015b].

– **RQ11: Relevance.** As described in Section 5.2.2, DL-based modeling deltas consist of precise information about each change including the type of change, a reference to modeling artifact, and a modeling artifact which has to be changed. These operations allow for representing all model changes embodying all necessary information about each independent change. The relevance and correctness of DL operations is affirmed by the meta-models of modeling languages. DL-based modeling deltas are complete consisting of relevant information about differences between the subsequent or parallel revisions of models.

These requirements have played an essential role in finding suitable difference representation approach in modeling deltas for collaborative MDSE. By fulfilling these requirements, DL provides more efficient ways of managing, manipulating and reusing difference information improving performance of data processing. The aforementioned properties provided by DL contribute to solid and common syntactic grounds for representing model differences in diverse domains and effortless development of further services on the top. DL has satisfied aforementioned properties throughout this thesis. It can easily be adapted for many domain-specific modeling languages with respect to their meta-models regardless their graphical constructs.

10.3 Fulfillment of Expected Benefits

In Chapter 3, each use case (concurrent and sequential collaboration, and history analysis) has defined a list of the expected advantages of DL. This section recalls these expected advantages if they are provided by DL.

Benefits for Concurrent Collaborative Modeling.
The technical challenges described in Section 3.1.2 and required support defined in Section 3.1.3 are satisfied by the generic DL.

The modeling deltas in concurrent collaborative modeling are represented by DL enabling quick synchronization of change between collaborators in real-time. DL supports several following advantages in concurrent collaborative modeling:

1. The model changes represented by DL completely satisfy *operational synchronization* principles which provide high performance in change synchronization in real-time. This benefit is already accomplished in case of `Kotelett` (Section 7.2) and `CoMo` (Section 7.3) applications.

2. The high performance by DL-based change representations allows for avoiding possible change conflicts in real-time. `Kotelett` is used in Software Engineering lectures for teaching purposes by a group of students including more than ten collaborators in parallel. The tool was also used experimentally by more than ten users located over long distance (Germany, Canada, Mozambique, and Uzbekistan), all connecting to the same server located in Germany. During these experiments, the tool has shown sufficiently high performance by synchronization of small DL-based modeling deltas [Kuryazov et al., 2018].

3. According to general language design, DL serves as a common change representation and exchange format for various components and services of concurrent and sequential collaborative modeling, and history analysis. As discussed in Part IV, concurrent and sequential collaborative modeling scenarios, as well as model history analysis application take advantage of the same common DL-based modeling deltas repositories as the single point of truth.

4. The same underlying DL is utilized to represent changes on concurrent revisions (in *forward deltas*), differences between the subsequent revisions of software models (in *backward deltas*) and working copies (in *active deltas*).

5. Any revision of shared models or their parts can be traced and browsed for further analysis by the *DL tracer* service based on UUIDs.

6. In concurrent collaborative modeling, modeling deltas consisting of small set of DL operations are easily be transferred over the network. Since it consists of only changed parts of software models, the performance has mostly been high.

Benefits for Sequential Collaborative Modeling
In case of sequential collaborative modeling, representations are based on the *change-oriented* technique, i.e., only changed modeling artifacts are considered in backward modeling deltas. Storing only changed modeling artifacts is quite efficient in case of a huge amount of model revisions with several thousand modeling artifacts. DL yields several significant contributions to sequential collaborative modeling:

1. DL facilitates tool developers' productivity with precise, concise and clear descriptions (extensible). It is fully expressive, yet unambiguous and necessary knowledge about each change can easily be gained.
2. It is declarative enough by making implicit any concepts or mechanisms that can be intuitively interpreted from the context. It does not rely on specific technical spaces or modeling languages.
3. DL-based modeling deltas form directly executable descriptions of model differences. It allows for converting software models from one revision to another, applying modeling deltas by the *DL applier* service.
4. DL-based delta operations embody only changed parts of software models saving memory and time.

Benefits for Model History Analysis.
Since software models are the visual form of software system design and consider all aspects of design level concepts, the *DL tracer* service provides advanced querying, browsing and visualization features for software model repositories.

1. The *DL tracer* service improves the performance of data extraction by inspecting the DL-based modeling delta repositories. The extracted data is then visualized in convenient ways for collaborators and stakeholders.
2. Regardless the textual representation of the DL-based modeling deltas, required history data (i.e., model changes) can be traced by following actual modeling concepts.
3. The *DL tracer* operates based on the queries provided by collaborators. It can trace complete models or their certain aspects.

10.4 Summary

This chapter has recalled the DL-based collaborative modeling applications and modeling languages that the DL-based collaborative modeling infrastructure either fully or partly validated. Specific DLs conforming to specific modeling languages are generated by their underlying meta-models, i.e., language concepts. These

specific DLs are then utilized as the common underlying modeling delta representation means, more prominently, for both concurrent and sequential collaborative modeling, as well as model history analysis. Because, there are very few approaches which are meant to cope with both collaborative modeling scenarios by the same underlying representation approach for MDSE. The DL-based modeling delta representation and its supplementary services fulfill the requirements defined in Section 4.3. Moreover, as the result of the studies and investigations in Section 3, several expected advantages of difference representation for concurrent and sequential collaborative modeling, and model history analysis are identified. DL and its services are capable of delivering these expected advantages.

In general, the significance and quality of any research work or thesis is estimated by their applicability. In order to examine the applicability and usability of the approach, DL is applied to various applications which are validated in several domain-specific modeling languages. Although the DL-based collaborative modeling application is not the end products in industrial scale, yet as a research prototype it has performed sufficiently and conveyed considerable results.

Chapter 11

Conclusion

Generally, this thesis addressed the problem of model difference representation by a **Difference Language (DL)** and its applications by providing several services. Part I has given a brief introduction to the overall collaborative development and defined main research objectives of this research work. In Part II, Chapter 3 was dedicated to study the state of the art in source code-driven collaborative development to derive underlying similar concepts, architectures, principles and terminologies for collaborative modeling. Chapter 4 has investigated the existing difference representation approaches and their supplementary services. As the result of the literature study, Section 4.3 has derived and defined several requirements regarding applicability, re-usability and adaptability of model difference representation.

In Part III, this thesis has introduced DL for model difference representations and several reasonable services for extending application areas of DL. Part IV has demonstrated concurrent and sequential collaborative modeling, and model history analysis applications that are developed on top of DL and based on the specific orchestrations of the DL services. By developing the collaborative modeling infrastructure on top of DL, this thesis has demonstrated its applicability.

Section 10.2 of this part has discussed the fulfillment of requirements (from Section 4.3) by DL, its services and applications. Section 10.3 has revisited all expected benefits (from Chapter 3) of DL for concurrent and sequential collaborative modeling, and model history analysis applications.

This, conclusion chapter summarizes the overall thesis by explaining some learned lessons in Section 11.1 and contributions of the thesis in Section 11.2.

11.1 Lessons Learned

In the framework of this thesis, the conceptual idea of DL is elaborated and several applications are developed based on DL. These applications are validated in

various domains, as well. During these research, development and validation activities, several research findings and lessons are determined and learned. More considerably, these findings are not considered by the existing approaches in the research field. This section describes research findings and lessons learned during this research.

- *Combined Meta-model.* In order to provide collaborative modeling on both, the modeling concepts of modeling languages and the layout concepts of modeling tools, the meta-models used by DL can be designed consisting of modeling language content (i.e., abstract syntax) and layout (i.e., concrete syntax) part. With this way of designing meta-models, DL is used to represent, store and synchronize changes on concrete syntax together with abstract syntax. This separation extends applicability of the underlying collaborative modeling infrastructure to a wide range of modeling languages and tools.

- *Conflicts in Concurrent Collaboration.* The concurrent collaborative modeling applications are experimented by several users located in different locations over long distance. During these experiments, the tools have not shown any conflict or performance issues. Change synchronization was fast enough to exchange all modeling deltas before conflicts may occur.

- *Active Delta.* The most concurrent and sequential collaborative systems inspected in Chapter 3 and Chapter 4 usually store the working copies of software systems (i.e., source code or models) in their repositories. The working copies of software models are stored as a whole by notations of domain languages. This thesis has introduced a term *active delta* to represent (store) the working copies of software models using DL operations. Like forward and backward modeling deltas, active deltas are also represented in terms of specific DLs and they consists of only creation operations. The working copies of software models are then derived by applying active deltas to empty models.

- *Additional Attributes.* If the meta-models of modeling languages or modeling tools do not provide an attribute on their root meta-class for handling persistent identifiers, that additional attribute can be added to the root meta-class in order to store the persistent identifiers of modeling artifacts. Moreover, additional attributes can be added for storing additional meta information, e.g., about collaborators (e.g., user names), time-stamps, etc. These additional information might be useful during collaborative modeling or when analyzing model histories.

- *Simultaneous Modification of Same Artifact.* In the concurrent collaborative modeling applications, when two collaborators change a modeling artifact at the same time, the behavior of concurrent collaborative modeling is not defined. It can happen that the deltas are swapped on a client, which overwrites the respective change of the collaborator. For instance, if the attribute value of a class is changed by one collaborator and the same attribute is changed by

another collaborator at the same time, the change the first collaborator just made can be updated by the incoming delta from the second collaborator. A latest change wins in such cases.

11.2 Contributions

As long as there are several modeling languages and model designing tools, the DL-based collaborative modeling infrastructure does not rely on a specific modeling language or modeling tool. The DL-based collaborative modeling supports both concurrent and sequential collaborative modeling scenarios making DL the common underlying change and difference representation technique for both. Since difference representation lies at the core of collaborative modeling, DL delivers several notable contributions to collaborative MDE. Below, the main scientific contribution of this thesis are revisited:

Awareness of Content and Layout. The content of software models is recognized by looking at the meta-models they conform to. The graphical design of models is aligned by their layout data in the model editors of model designing tools. There are several graphical modeling editors which display the modeling content with their layout representation information. Like models conform to their meta-models, layout information is represented by and conform to their graphical notations. In order to provide collaborative modeling on such graphical modeling tools, DL is aware of layout notation (i.e., concrete syntax) together with the meta-models (i.e., abstract syntax) of modeling languages. For instance, if a collaborator changes the position and size of a modeling artifact in a graphical editor, the same changes occur simultaneously in the modeling editors of other tool instances, as well. In concurrent collaborative modeling, this is provided by synchronizing forward modeling deltas consisting of the both, layout changes and content changes. The DL operations representing changes in layout part are also stored in backward modeling deltas in sequential collaborative modeling.

Genericity. There are several modeling languages and graphical notations following diverse formal specifications and concepts. DL is generic with respect to the meta-models of modeling languages and their layout without restricting itself to particular modeling languages or tools. Its collaborative modeling support is tailored to the wide range of modeling languages and tools. To this end, this thesis provides the *DL generator* service to derive specific DLs from the meta-models of modeling languages. Eventually, the application areas of DL can be extended by the specific orchestrations of the DL services.

Applicability. Overall collaborative modeling forms two scenarios such as *concurrent* and *sequential collaborative modeling*, whereas difference representation is a common and fundamental concern for both. The most approaches

investigated in Chapter 4 focus only on the some aspects of these collaborative modeling scenarios. DL supports the both scenarios of collaborative modeling by being applicable, persistent, implementable and expressive. The DL-based modeling delta repositories serve as the single point of truth for the both scenarios of collaborative modeling, as well as for model history analysis.

These contributions are reasonable advancements provided by DL for emerging an appropriate difference representation approach for collaborative MDSE. They are the efficient design of the data structure for representing model differences in modeling deltas. These significant principles are also the design foundations for DL that contribute to empower the qualification and solidity of difference representation. This thesis has developed the DL-based collaborative modeling infrastructure that can be further reused and applied to a wide range of domain-specific modeling languages and tools.

References

[A. Schmidt, 2018] A. Schmidt (visited on 22.07.2018). emfCollab: Collaborative Editing for EMF models. http://qgears.com/products/emfcollab/.

[Ahmed-Nacer et al., 2011] Ahmed-Nacer, M., Ignat, C.-L., Oster, G., Roh, H.-G., and Urso, P. (2011). Evaluating CRDTs for Real-time Document Editing. In *Proceedings of the 11th ACM Symposium on Document Engineering*, DocEng '11, pages 103–112, New York, NY, USA. ACM.

[Alanen and Porres, 2003] Alanen, M. and Porres, I. (2003). Difference and union of models. *In P.Stevens, J.Whittle, and G. Booch, editors, Proc. 6th Int. Conf. on the UML, Springer*, volume 2863 of LNCS:pages 2–17.

[Altmanninger et al., 2007] Altmanninger, K., Bergmayr, A., Schwinger, W., and Kotsis, G. (2007). Semantically enhanced conflict detection between model versions in SMoVer by example. In *Procs of the Int. Workshop on Semantic-Based Software Development at OOPSLA*.

[Altmanninger et al., 2009] Altmanninger, K., Seidl, M., and Wimmer, M. (2009). A survey on model versioning approaches. *International journal of Web Information Systems (IJWIS)*, 5:271–304.

[Appeldorn, 2018] Appeldorn, M. (2018). Collaborative Modeling Support for UML Activity Diagrams. Bachelor Thesis. University of Oldenburg.

[Appeldorn et al., 2018] Appeldorn, M., Kuryazov, D., and Winter, A. (2018). Delta-Driven Collaborative Modeling. In Berger, T. and Hebig, R., editors, *Models 2018*, Capenhagen. ACM SIG WEB.

[AppJet Inc., 2017] AppJet Inc. (visited on 01.02.2017). Etherpad. http://www.etherpad.com.

[Arendt et al., 2010] Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010). Henshin: advanced concepts and tools for in-place EMF model transformations. In *Model Driven Engineering Languages and Systems*, pages 121–135. Springer.

[Arnold, 1996] Arnold, R. (1996). *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA.

[Aßmann et al., 2006] Aßmann, U., Zschaler, S., and Wagner, G. (2006). Ontologies, meta-models, and the model-driven paradigm. In *Ontologies for software engineering and software technology*, pages 249–273. Springer.

[Bafoutsou and Mentzas, 2002] Bafoutsou, G. and Mentzas, G. (2002). Review and functional classification of collaborative systems. *International journal of information management*, 22(4):281–305.

[Bailor et al., 2011] Bailor, J. B., Bernstein, E. J., Knight, M. R., Antos, C. J., et al. (2011). Document Merge. US Patent 8,028,229.

[Baudivs, 2014] Baudivs, P. (2014). Current concepts in version control systems.

[Beck et al., 2001] Beck, K., Beedle, M., Bennekum, A. V., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., et al. (2001). The agile manifesto.

[Berlin and Rooney, 2006] Berlin, D. and Rooney, G. (2006). *Practical Subversion*. Apress.

[Bezivin, 2005] Bezivin, J. (2005). On the Unification Power of Models. *SOSYM*, 4(2):171–188.

[Bézivin, 2006] Bézivin, J. (2006). Model driven engineering: An emerging technical space. In *Generative and transformational techniques in software engineering*, pages 36–64. Springer.

[Bézivin and Gerbé, 2001] Bézivin, J. and Gerbé, O. (2001). Towards a precise definition of the OMG/MDA framework. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280. IEEE.

[Bézivin and Kurtev, 2005] Bézivin, J. and Kurtev, I. (2005). Model-based technology integration with the technical space concept. In *Metainformatics Symposium*, volume 20, pages 44–49.

[Bieman et al., 2003] Bieman, J., Andrews, A. A., and Yang, H. (2003). Understanding change-proneness in OO software through visualization. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 44–53. IEEE.

[Booch and Brown, 2003] Booch, G. and Brown, A. W. (2003). Collaborative development environments. *Advances in computers*, 59:1–27.

[Borenstein, 1992] Borenstein, N. (1992). Computational mail as network infrastructure for computer-supported cooperative work. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 67–74. ACM.

[Bourque et al., 2014] Bourque, P., Fairley, R., et al. (2014). *Guide to the software engineering body of knowledge (SWEBOOK (R)): Version 3.0*. IEEE Computer Society Press.

[Breivold and Larsson, 2007] Breivold, H. P. and Larsson, M. (2007). Component-based and service-oriented software engineering: Key concepts and principles. In *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2007)*, pages 13–20. IEEE.

[Brosch et al., 2012] Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., and Wimmer, M. (2012). An Introduction to Model Versioning. In *Formal Methods for Model-Driven Engineering*, pages 336–398. Springer.

[Brosch et al., 2010] Brosch, P., Kappel, G., Seidl, M., Wieland, K., Wimmer, M., Kargl, H., and Langer, P. (2010). Adaptable Model Versioning in Action. *in: Proc. Modellierung 2010, Klagenfurt, Österreich*, Lecture Notes in Informatics 161:p.221–236.

[Brun and Pierantonio, 2008] Brun, C. and Pierantonio, A. (2008). Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34.

[Brunet et al., 2006] Brunet, G., Chechik, M., Easterbrook, S., Nejati, S., Niu, N., and Sabetzade, M. (2006). A Manifesto for Model merging. *Proceeding GaMMa '06 Proceedings of the 2006 international workshop on Global integrated model management ACM.*

[Buchegger et al., 2009] Buchegger, S., Schiöberg, D., Vu, L.-H., and Datta, A. (2009). PeerSoN: P2P social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, pages 46–52. ACM.

[Chen, 1976] Chen, P. P.-S. (1976). The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36.

[Chikofsky and Cross, 1990] Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE software*, 7(1):13–17.

[Cicchetti, 2008] Cicchetti, A. (2008). *Difference Representation and Conflict.* PhD thesis, University of L'Aquila, (Italy).

[Cicchetti et al., 2007] Cicchetti, A., Ruscio, D. D., and Pierantonio, A. (2007). A Metamodel independent approach to difference representation. *journal of Object Technology*, 6:9:165–185.

[Cinergix Pty., 2015] Cinergix Pty. (visited on 01.06.2015). CreateLy. http://www.creately.com.

[Clarence et al., 1991] Clarence, E., Simon, G., and Gail, R. (1991). Groupware: Some Issues and Experiences. *Commun. ACM*, 34(1):39–58.

[Cohen, 2003] Cohen, B. (2003). Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72.

[Collberg et al., 2003] Collberg, C., Kobourov, S., Nagra, J., Pitts, J., and Wampler, K. (2003). A System For Graph-based Visualization of the Evolution of Software. *Proc. 2003 ACM Symposium on Software Visualization SoftVis'03*, ACM:p.77.

[Collins-Sussman et al., 2004] Collins-Sussman, B., Fitzpatrick, B., and Pilato, M. (2004). Version Control with Subversion. *O'Reilly Media*.

[Conradi and Westfechtel, 1998] Conradi, R. and Westfechtel, B. (1998). Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282.

[Constant, 2012] Constant, O. (2012). EMF Diff/Merge.

[Cover, 2001] Cover, R. (2001). XML Metadata Interchange (XMI).

[Dahl, 2004] Dahl, O.-J. (2004). The Birth of Object Orientation: The Simula Languages. In *From Object-Orientation to Formal Methods*, pages 15–25. Springer.

[Dahl et al., 1972] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured programming*. Academic Press Ltd.

[Dahm and Widmann, 1998] Dahm, P. and Widmann, F. (1998). Das Graphenlabor. Technical Report 11/98, Universität Koblenz-Landau, Institut für Informatik, Koblenz. Fachberichte Informatik.

[Date and Darwen, 1987] Date, C. and Darwen, H. (1987). *A Guide to the SQL Standard*, volume 3. Addison-Wesley New York.

[Davison, 1990] Davison, W. (1990). Unified Context Diff Tools.

[Dillenbourg, 1999] Dillenbourg, P. (1999). What do you mean by collaborative learning? *Collaborative-learning: Cognitive and computational approaches*, 1:1–15.

[Dirix et al., 2013] Dirix, M., Muller, A., and Aranega, V. (2013). GenMyModel: an online UML case tool. In *ECOOP*.

[Ducasse et al., 2005] Ducasse, S., Gîrba, T., and Favre, J.-M. (2005). Modeling software evolution by treating history as a first class entity. *Electronic Notes in Theoretical Computer Science*, 127(3):75–86.

[Ebert, 1987] Ebert, J. (1987). A versatile data structure for edge-oriented graph algorithms. *Communications of the ACM*, 30(6):513–519.

[Ebert and Franzke, 1995] Ebert, J. and Franzke, A. (1995). A declarative approach to graph based modeling. In *Graph-Theoretic Concepts in Computer Science*, pages 38–50. Springer.

[Ebert and Horn, 2014] Ebert, J. and Horn, T. (2014). GReTL: an extensible, operational, graph-based transformation language. *Software and Systems Modeling*, pages 1–21.

[Ebert et al., 2002] Ebert, J., Kullbach, B., Riediger, V., and Winter, A. (2002). Gupro-generic understanding of programs an overview. *Electronic Notes in Theoretical Computer Science*, 72(2):47–56.

[Ebert et al., 2008] Ebert, J., Riediger, V., and Winter, A. (2008). Graph technology in reverse engineering. The TGraph approach. In *Proc. 10th Workshop Software Reengineering. GI Lecture Notes in Informatics*, pages 23–24. Citeseer.

[Ebert et al., 1997] Ebert, J., Süttenbach, R., and Uhe, I. (1997). Meta-CASE in Practice: a Case for KOGGE. In *Advanced Information Systems Engineering*, pages 203–216. Springer.

[Eclipse Orion, 2014] Eclipse Orion (2014). Eclipse Orion.

[EMFCompare, 2017] EMFCompare (visited on 15.03.2017). EMF Compare. http://wiki.eclipse.org/EMFCompare.

[Engelbart, 1968] Engelbart, D. (1968). The mother of all demos.

[English et al., 2010] English, B., Alderman, B., and Ferraz, M. (2010). *Microsoft SharePoint 2010 Administrator's Companion*. Pearson Education.

[Erl, 2005] Erl, T. (2005). *Service-oriented architecture: concepts, technology, and design*. Pearson Education India.

[Esoteric Software, 2018] Esoteric Software (visited on 22.07.2018). KryoNet: TCP/UDP Client/Server library for Java. https://github.com/EsotericSoftware/kryonet.

[Fernández-Ramil and Lehman, 2000] Fernández-Ramil, J. and Lehman, M. M. (2000). Metrics of Software Evolution as Effort Predictors-A Case Study. In *ICSM*, pages 163–172. IEEE.

[Firebase Inc., 2017] Firebase Inc. (visited on 01.02.2017). Firepad. https://firepad.io.

[Fluegge, 2009] Fluegge, M. (2009). Entwicklung einer kollaborativen Erweiterung fuer GMF-Editoren auf Basis modellgetriebener und webbasierter Technologien. *Master's thesis, University of Applied Sciences Berlin*.

[Fluidbyte, 2014] Fluidbyte (2014). Codiad v.2.8.1 Web Based, Cloud IDE.

[Franzago et al., 2018] Franzago, M., Ruscio, D. D., Malavolta, I., and Muccini, H. (2018). Collaborative model-driven software engineering: a classification framework and a research map. *IEEE Transactions on Software Engineering*, pages 1–1.

[Fraser, 2009] Fraser, N. (2009). Differential synchronization. In *Proceedings of the 9th ACM symposium on Document engineering*, pages 13–20. ACM.

[García-Domínguez et al., 2018] García-Domínguez, A., Kolovos, D. S., Barmpis, K., Daniel, G., and Sunyé, G. (2018). Taming Large Models with Hawk and NeoEMF. MoDELS'2018, https://projects.eclipse.org/proposals/eclipse-hawk.

[GenMyModel, 2015] GenMyModel (visited on 01.06.2015). GenMyModel. http://www.genmymodel.com/.

[Ghosh, 2010] Ghosh, D. (2010). *DSLs in Action*. Manning Publications Co.

[Glasser, 1978] Glasser, A. L. (1978). The evolution of a Source Code Control System. *ACM SIGMETRICS Performance Evaluation Review*, 7.

[Gliffy, 2017] Gliffy (visited on 14.05.2017). Gliffy. https://www.gliffy.com/.

[GMF, 2018] GMF (visited on 22.07.2018). Graphical Modeling Project (GMP). http://www.eclipse.org/modeling/gmp/.

[Godfrey and Tu, 2002] Godfrey, M. and Tu, Q. (2002). Tracking structural evolution using origin analysis. In *Proceedings of the international workshop on Principles of software evolution*, pages 117–119. ACM.

[Goldman et al., 2011] Goldman, M., Little, G., and Miller, R. (2011). Collabode: collaborative coding in the browser. In *Proceedings of the 4th international workshop on Cooperative and human aspects of software engineering*, pages 65–68. ACM.

[Google Inc., 2017] Google Inc. (visited on 01.02.2017). Google Docs. http://docs.google.com.

[Grose et al., 2002] Grose, T. J., Doney, G. C., and Brodsky, S. A. (2002). *Mastering XML: Java Programming with XMI and UML*, volume 21. John Wiley & Sons.

[Gupta, 2000] Gupta, U. (2000). Done deals. *Venture Capitalists Tell Their Stories, Boston*.

[Hassan, 2008] Hassan, A. (2008). The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE.

[Hausladen et al., 2014] Hausladen, J., Pohn, B., and Horauer, M. (2014). A cloud-based integrated development environment for embedded systems. In *Mechatronic and Embedded Systems and Applications (MESA), 2014 IEEE/ASME 10th International Conference on*, pages 1–5. IEEE.

[Heinrich, 2013] Heinrich, M. (2013). Enriching Web Applications Efficiently with Real-Time Collaboration Capabilities.

[Hellmann, 2011] Hellmann, D. (2011). *The Python standard library by example*. Addison-Wesley Professional.

[Helming and Koegel, 2013] Helming, J. and Koegel, M. (2013). EMFStore. http://eclipse.org/emfstore.

[Henson and Garzik, 2002] Henson, V. and Garzik, J. (2002). Bitkeeper for kernel developers. In *Ottawa Linux Symposium*, page 197.

[Herbsleb and Moitra, 2001] Herbsleb, J. and Moitra, D. (2001). Global software development. *IEEE software*, 18(2):16–20.

[Herrmannsdoerfer and Koegel, 2010] Herrmannsdoerfer, M. and Koegel, M. (2010). Towards a Generic Operation Recorder for Model Evolution. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, IWMCP 10, pages 76–81, New York, NY, USA. ACM.

[Hindle and German, 2005] Hindle, A. and German, D. (2005). *SCQL: A formal model and a query language for source control repositories*, volume 30:4. ACM.

[Hoare et al., 2005] Hoare, G. et al. (2005). Monotone.

[Horn, 2013] Horn, T. (2013). Model querying with FunnyQT. In *International Conference on Theory and Practice of Model Transformations*, pages 56–57. Springer.

[Hudson, 2002] Hudson, G. (2002). Notes on keeping version histories of files. *Unpublished personal notes.*

[Hunt and MacIlroy, 1976] Hunt, J. W. and MacIlroy, D. (1976). *An algorithm for differential file comparison.* Bell Laboratories New Jersey.

[Jelschen, 2014] Jelschen, J. (2014). SENSEI: Software evolution service integration. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference*, pages 469–472. IEEE.

[John, 2010] John, A. (2010). Real-time content collaboration. US Patent App. 12/419,926.

[Jouault et al., 2008] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Science of computer programming*, 72(1):31–39.

[Jouault and Bézivin, 2006] Jouault, F. and Bézivin, J. (2006). KM3: a DSL for Metamodel Specification. In *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer.

[Kagdi et al., 2007] Kagdi, H., Collard, M., and Maletic, J. (2007). A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, 19(2):77–131.

[Kay, 1996] Kay, A. (1996). The early history of Smalltalk. In *History of programming languages—II*, pages 511–598. ACM.

[Kehrer et al., 2012] Kehrer, T., Kelter, U., Ohrndorf, M., and Sollbach, T. (2012). Understanding model evolution through semantically lifting model differences with SiLift. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 638–641. IEEE.

[Kehrer et al., 2011] Kehrer, T., Kelter, U., and Taentzer, G. (2011). A Rule-based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 163–172, Washington, DC, USA. IEEE Computer Society.

[Kehrer et al., 2013a] Kehrer, T., Kelter, U., and Taentzer, G. (2013a). Consistency-Preserving Edit Scripts in Model Versioning. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 191–201. IEEE.

[Kehrer et al., 2013b] Kehrer, T., Rindt, M., Pietsch, P., and Kelter, U. (2013b). Generating Edit Operations for Profiled UML Models. In *ME@ MoDELS*, pages 30–39. Citeseer.

[Kim and Notkin, 2006] Kim, M. and Notkin, D. (2006). Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64. ACM.

[Kleppe et al., 2003] Kleppe, A., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Koegel et al., 2010] Koegel, M., Herrmannsdoerfer, M., von Wesendonk, O., and Helming, J. (2010). Operation-based conflict detection. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 21–30. ACM.

[Kofman and Perjons, 2004] Kofman, M. and Perjons, E. (2004). MetaDiff - a Model Comparison Framework. *http://metadiff.sourceforge.net/docs/metadiff.pdf, undated.*

[Kolovos et al., 2009] Kolovos, D., Ruscio, D. D., Pierantonio, A., and Paige, R. F. (2009). Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*, pages 1–6. IEEE.

[Kullbach et al., 1998] Kullbach, B., Winter, A., Dahm, P., and Ebert, J. (1998). Program comprehension in multi-language systems. In *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on*, pages 135–143. IEEE.

[Kurtev et al., 2002] Kurtev, I., Bézivin, J., and Aksit, M. (2002). Technological spaces: An initial appraisal. *CoopIS, DOA*, 2002.

[Kuryazov, 2014] Kuryazov, D. (2014). Delta Operations Language for Model Difference Representation. In Plödereder, E., Grunske, L., and Schneider Ull, E. D., editors, *44. Jahrestagung der Gesellschaft für Informatik e.V. (GI), ISBN 978-3-88579-626-8*, volume 232, pages 2221–2232, Stuttgart, Germany. Gesellschaft für Informatik.

[Kuryazov et al., 2012] Kuryazov, D., Jelschen, J., and Winter, A. (2012). Describing Modeling Delta By Model Transformation. In *Softwaretechnik Trends (Issue on International Workshop on Comparison and Versioning of Software Models (CVSM 2012))*, no. Band 32 Heft 4. Gesellschaft für Informatik.

[Kuryazov et al., 2013] Kuryazov, D., Solsbach, A., and Winter, A. (2013). Versioning Sustainability Reports. In *5.BUIS-Tage: IT-gestütztes Ressourcen- und Energiemanagement*, pages 409–419. Springer-Verlag.

[Kuryazov and Winter, 2015a] Kuryazov, D. and Winter, A. (2015a). Collaborative Modeling Empowered by Modeling Deltas. Lausanne, Switzerland. ACM. ISBN: 978-1-4503-3714-4.

[Kuryazov and Winter, 2015b] Kuryazov, D. and Winter, A. (2015b). Towards Model History Analysis Using Modeling Deltas. *Softwaretechnik-Trends*, 35(2):15–16.

[Kuryazov et al., 2018] Kuryazov, D., Winter, A., and Reussner, R. (2018). Collaborative Modeling Enabled by Version Control. In Schaefer, I., Karagiannis, D., and Vogelsang, A., editors, *Modellierung 2018*, volume P-280, pages 183–198, Bonn. Gesellschaft für Informatik (GI). ISBN: 978-3-88579-674-9.

[Küpker, 2013] Küpker, C. (2013). General Model Difference Calculation. Bachelor Thesis, Carl von Ossietzky University of Oldenburg.

[Langer, 2011] Langer, P. (2011). *Adaptable model versioning based on model transformation by demonstration*. na.

[Lanusse et al., 2009] Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., and Terrier, F. (2009). Papyrus UML: an open source toolset for MDA. In *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, pages 1–4.

[Lehman, 1996] Lehman, M. M. (1996). Laws of software evolution revisited. In *LNCS, European Workshop on Software Process Technology*, pages 108–124. Springer.

[Lehman and Fernández-Ramil, 2001] Lehman, M. M. and Fernández-Ramil, J. (2001). Evolution in software and related areas. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 1–16. ACM.

[Leroux et al., 2006] Leroux, D., Nally, M., and Hussey, K. (2006). Rational Software Architect: A tool for domain-specific modeling. *IBM systems journal*, 45(3):555–568.

[Lientz and Swanson, 1980] Lientz, B. and Swanson, B. (1980). Software maintenance management.

[Lientz et al., 1978] Lientz, B., Swanson, B., and Tompkins, G. (1978). Characteristics of application software maintenance. *Communications of the ACM*, 21:6:466–471.

[Lin et al., 2007] Lin, Y., Gray, J., and Jouault, F. (2007). DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361.

[Lin et al., 2004] Lin, Y., Zhang, J., and Gray, J. (2004). Model comparison: A key challenge for transformation testing and version control in model driven software development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, volume 108, page 6.

[MacDonald, 2000] MacDonald, J. (2000). *File system support for delta compression*. PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley.

[Mackall, 2006] Mackall, M. (2006). Towards a better SCM: Revlog and mercurial. *Proc. Ottawa Linux Sympo*, 2:83–90.

[MacKenzie et al., 2003] MacKenzie, D., Eggert, P., and Stallman, R. (2003). *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd.

[Maurer and Wolfthal, 2016] Maurer, N. and Wolfthal, M. (2016). *Netty in Action*. Manning Publications.

[McAffer et al., 2010] McAffer, J., Lemieux, J.-M., and Aniszczyk, C. (2010). *Eclipse Rich Client Platform*. Addison-Wesley Professional.

[Meier and Winter, 2018] Meier, J. and Winter, A. (2018). Traceability enabled by metamodel integration. *Softwaretechnik-Trends*, 38(1):21–26.

[Mellor et al., 2003] Mellor, S., Clark, T., and Futagami, T. (2003). Model-driven development: guest editors' introduction. *IEEE software*, 20(5):14–18.

[Mens, 2002] Mens, T. (2002). A State-of-the-Art Survey on Software Merging. *IEEE Trans. Software Eng.*, 28:5:449–462.

[Meyer, 1988] Meyer, B. (1988). *Object-oriented software construction*, volume 2. Prentice hall New York.

[Miller and Mukerji, 2003] Miller, J. and Mukerji, J. (2003). MDA guide version 1.0. 1, Object Management Group. *Inc., June*.

[Modica et al., 2009] Modica, T., Biermann, E., and Ermel, C. (2009). An Eclipse Framework for Rapid Development of Rich-featured GEF Editors based on EMF Models. *GI Jahrestagung*, 154:2972–2985.

[MOF, 2003] MOF (2003). Meta Object Facility (MOF) 2.0 Core Specification, OMG Document ptc/03-10-04.

[Myers, 1986] Myers, E. W. (1986). An O (ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266.

[Naccarato, 2004] Naccarato, G. (2004). Template-based code generation with apache velocity.

[Naur and Randell, 1969] Naur, P. and Randell, B. (1969). Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.

[Nejati et al., 2007] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., and Zave, P. (2007). Matching and merging of statecharts specifications. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 54–64. IEEE.

[Nichols et al., 1995] Nichols, D., Curtis, P., Dixon, M., and Lamping, J. (1995). High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 111–120. ACM.

[Obeo Network, 2017] Obeo Network (visited on 02.10.2017). UML Designer. http://www.umldesigner.org/.

[Oliveira et al., 2005] Oliveira, H., Murta, L., and Werner, C. (2005). Odyssey-VCS: a flexible version control system for UML model elements. In *Proceedings of the 12th international workshop on Software configuration management*, pages 1–16. ACM.

[OMG, 2014] OMG (last visited on 2014). Object Management Group (OMG). http://www.omg.org.

[Ong, 1998] Ong, C.-M. (1998). *Dynamic simulation of electric machinery: using MATLAB/SIMULINK*, volume 5. Prentice hall PTR Upper Saddle River, NJ.

[Oster et al., 2006] Oster, G., Urso, P., Molli, P., and Imine, A. (2006). Data consistency for P2P collaborative editing. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 259–268. ACM.

[Preguica et al., 2009] Preguica, N., Marques, J. M., Shapiro, M., and Letia, M. (2009). A commutative replicated data type for cooperative editing. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 395–403. IEEE.

[Proceedings, 2006] Proceedings, O. ., editor (2006). *Towards a Better SCM: Revlog and Mercurial*, volume 2, Ottawa.

[Project Group, 2014] Project Group (2014). Kotelett: Collaborative Modeling Tool. https://pg-kotelett.informatik.uni-oldenburg.de:8443/build/stable/. University of Oldenburg.

[QGears, 2018] QGears (visited on 22.10.2018). CoolRMI: High performance Java remote method invocation library. http://qgears.com/products/coolrmi/.

[Rama and Bishop, 2006] Rama, J. and Bishop, J. (2006). A survey and comparison of CSCW groupware applications. In *Proceedings of the 2006 annual*

research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, pages 198–205. South African Institute for Computer Scientists and Information Technologists.

[Raumbaugh et al., 2004] Raumbaugh, J., Jacobson, I., and Booch, G. (2004). *Unified Modeling Language Reference Manual*. Pearson Higher Education.

[Rentsch, 1982] Rentsch, T. (1982). Object Oriented Programming. *SIGPLAN Not.*, 17(9):51–57.

[Ressel et al., 1996] Ressel, M., Nitsche-Ruhland, D., and Gunzenhäuser, R. (1996). An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 288–297. ACM.

[Robbes, 2007] Robbes, R. (2007). Mining a Change-Based Software Repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 15–, Washington, DC, USA. IEEE Computer Society.

[Robles et al., 2004] Robles, G., González-Barahona, J., and Ghosh, R. (2004). Gluetheos: Automating the retrieval and analysis of data from publicly available software repositories. In *Proceedings of the international workshop on mining software repositories*, pages 28–31. IET.

[Rocco et al., 2015] Rocco, J. D., Ruscio, D. D., Iovino, L., and Pierantonio, A. (2015). Collaborative repositories in model-driven engineering [software technology]. *IEEE Software*, 32(3):28–34.

[Rochkind, 1975] Rochkind, M. (1975). The source code control system. *Software Engineering, IEEE Transactions on*, 4:364–370.

[Roh et al., 2011] Roh, H.-G., Jeon, M., Kim, J.-S., and Lee, J. (2011). Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368.

[Roundy, 2005] Roundy, D. (2005). Darcs: distributed version management in haskell. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4. ACM.

[Roundy, 2009] Roundy, D. (2009). Theory of Patches. *darcs.net/manual/node8.html*.

[Rubel et al., 2011] Rubel, D., Wren, J., and Clayberg, E. (2011). *The Eclipse Graphical Editing Framework (GEF)*. Addison-Wesley Professional.

[Saeki, 2006] Saeki, M. (2006). Meta-modeling based version control system for software diagrams. In *IEICE Transactions on Information and Systems*, volume E89-D, pages 1390–1402. IEICE.

[Schipper et al., 2009] Schipper, A., Fuhrmann, H., and von Hanxleden, R. (2009). Visual comparison of graphical models. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 335–340. IEEE.

[Schmidt, 2006] Schmidt, D. C. (2006). Model-Driven Engineering. *IEEE Computer Society*, 39(2):25.

[Schmidt and Gloetzner, 2008] Schmidt, M. and Gloetzner, T. (2008). Constructing Difference Tools for Models Using the SiDiff Framework. *in: Companion volume, ICSE 2008*, pages 947–948.

[Schneider et al., 2004] Schneider, C., Zündorf, A., and Niere, J. (2004). CoObRA-a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments*.

[Schooler, 1996] Schooler, E. (1996). Conferencing and collaborative computing. *Multimedia Systems*, 4(5):210–225.

[Seidewitz, 2003] Seidewitz, E. (2003). What models mean. *IEEE software*, 20(5):26–32.

[Seidl et al., 2014] Seidl, C., Schaefer, I., and Aßmann, U. (2014). DeltaEcore-A Model-Based Delta Language Generation Framework. In *Modellierung*, pages 81–96.

[Singer et al., 2005] Singer, J., Elves, R., and Storey, M.-A. (2005). Navtracks: Supporting navigation in software maintenance. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 325–334. IEEE.

[Solsbach et al., 2011] Solsbach, A., Süpke, D., vom Berg, B. W., and Gómez, J. M. (2011). Sustainable online reporting model: A web based sustainability reporting software. In *Information Technologies in Environmental Engineering*, pages 165–177. Springer.

[Sriplakich et al., 2008] Sriplakich, P., Blanc, X., and Gervais, M.-P. (2008). Collaborative Software Engineering on Large-scale models: Requirements and Experience in ModelBus. *Proceedings of the 2008 ACM symposium on Applied computing 2008*, ACM:p.674–681.

[Steinberg et al., 2008] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse Modeling Framework*. Addison-Wesley Longman Publishing Co., Inc.

[Stephan and Antkiewicz, 2008] Stephan, M. and Antkiewicz, M. (2008). Ecore: A tool for editing and instantiating class models as feature models. *University of Waterloo, Tech. Rep*, 8:2008.

[Stepper, 2018] Stepper, E. (visited on 22.07.2018). EMF-based Model Repository: Corrected Data Objects (CDO). Eclipse Project Website. http://eclipse.org/cdo.

[Sun and Ellis, 1998] Sun, C. and Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. ACM.

[Suvanaphen and Roberts, 2004] Suvanaphen, E. and Roberts, J. (2004). Textual difference visualization of multiple search results utilizing detail in context. In *Theory and Practice of Computer Graphics, 2004. Proceedings*, pages 2–8. IEEE.

[Swicegood, 2008] Swicegood, T. (2008). *Pragmatic version control using Git.* Pragmatic Bookshelf.

[Szyperski, 2000] Szyperski, C. (2000). Component software and the way ahead. *Foundations of component-based systems*, pages 1–20.

[Taentzer et al., 2012] Taentzer, G., Ermel, C., Langer, P., and Wimmer, M. (2012). A fundamental approach to model versioning based on graph modifications: from theory to implementation. *journal: Software and Systems Modeling.*

[TeamEdit, 2011] TeamEdit (2011). A collaborative text editor.

[Tichy, 1985] Tichy, W. F. (1985). RCS — a system for version control. *Software – Practice Experience*, 15:Issue 7.

[Tolvanen, 2016] Tolvanen, J.-P. (2016). MetaEdit+ for collaborative language engineering and language use (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 41–45. ACM.

[Treude et al., 2007] Treude, C., Berlik, S., Wenzel, S., and Kelter, U. (2007). Difference Computation of Large Models. 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*, ESEC-FSE, pages 295–304, New York, NY, USA. ACM.

[van Deursen et al., 1998] van Deursen, A., Klint, P., et al. (1998). Little languages: Little maintenance? *Journal of software maintenance*, 10(2):75–92.

[Van Deursen et al., 2000] Van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages. *Centrum voor Wiskunde en Informatika*, 5:12.

[Visual Paradigm, 2013] Visual Paradigm (2013). Visual Paradigm for UML. *Visual Paradigm for UML-UML tool for software application development.*

[Viyović et al., 2014] Viyović, V., Maksimović, M., and Perisić, B. (2014). Sirius: A rapid development of DSM graphical editor. In *Intelligent Engineering Systems (INES), 2014 18th International Conference*, pages 233–238. IEEE.

[Weiser, 1981] Weiser, M. (1981). Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press.

[Weiss et al., 2009] Weiss, S., Urso, P., and Molli, P. (2009). Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 404–412. IEEE.

[Wenzel, 2008] Wenzel, S. (2008). Scalable Visualization of Model Differences. *Proc. 2008 ICSE Workshop on Comparison and Versioning of Software Models*, Leipzig:41–46.

[Wenzel, 2010] Wenzel, S. (2010). *Unique identification of elements in evolving models : towards fine-grained traceability in model-driven engineering*. PhD thesis, Universität Siegen.

[Wenzel and Kelter, 2008] Wenzel, S. and Kelter, U. (2008). Analyzing Model Evolution. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 831–834, Leipzig, Germany.

[Wingerd and Seiwald, 1998] Wingerd, L. and Seiwald, C. (1998). High-level best practices in software configuration management. *System Configuration Management*, pages 57–66.

[Winter, 2000] Winter, A. (2000). *Referenz-Metaschema für visuelle Modellierungssprachen*. PhD thesis, Universität Koblenz-Landau, Wiesbaden. zugl. Dissertation, Institut für Informatik. Universität Koblenz-Landau.

[Xing and Stroulia, 2005a] Xing, Z. and Stroulia, E. (2005a). Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31(10):850–868.

[Xing and Stroulia, 2005b] Xing, Z. and Stroulia, E. (2005b). UMLDiff: An Algorithm for Object-Oriented Design Differencing. *in: Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, IEEE Computer Society:54–65.

[Yang et al., 2000] Yang, Y., Sun, C., Zhang, Y., and Jia, X. (2000). Real time cooperative editing on the Internet. *IEEE Internet Computing*, 4(3):18–25.

[Zimmermann et al., 2005] Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.

Declaration of Authorship

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichungen, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Dilshodbek Kuryazov, 20.02.2019