Collaborative Modeling Enabled By Version Control

Dilshod Kuryazov [1], Andreas Winter [1], Ralf Reussner[2]

Abstract: Model-Driven Software Development is a key field in software development activities which is well-suited to design and develop large-scale software systems. Developing and maintaining large-scale model-driven software systems entail a need for collaborative modeling by a large number of software designers and developers. As long as software models are constantly changed during development and maintenance, collaborative modeling requires frequently sharing of model changes between collaborators. Thereby, a solid change representation support for model changes plays an essential role for collaborative modeling systems. This paper focuses on the problem of model change representation for collaborative modeling. It introduces a meta-model generic, operation-based and textual difference language to represent model changes in collaborative modeling. This paper also demonstrates a collaborative modeling application *Kotelett*.

Keywords: Model-driven Software Development and Evolution; Collaborative Modeling; Modeling Deltas; Model Difference Representation

1 Motivation

As a software engineering paradigm, Model-Driven Software Development (MDSD) 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 MDSD, software models which also comprise source code are the central artifacts. MDSD brings several main benefits 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 [KWB03, pp. 9ff].

Software models (e.g. in UML [RJB04]) are the key artifacts in MDSD activities. They 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 [KWB03].

[1] University of Oldenburg, Software Engineering Group, Uhlhornsweg 84, 26111 Oldenburg, Germany, {kuryazov, winter}@se.uni-oldenburg.de

[2] Karlsruhe Institute of Technology, Institute for Program Structures and Data Organization, Postfach 6980, D-76128 Karlsruhe, Germany, ralf.reussner@kit.edu

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 software evolution, models become large and complex raising a need for collaboration of several developers, designers and stakeholders (i.e. collaborators) on shared models i.e. *Collaborative modeling*.

Depending on the nature of interaction, collaborative modeling systems can be divided into two main forms, namely *sequential* and *concurrent* collaborative modeling [ESG91]:

- *Sequential Collaboration*. In sequential collaboration, collaborators design a shared model by checking out it into their distributed environment. After making changes, they merge their local changes into the main model. This scenario results in several subsequent revisions of the same shared central model. After each change, 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 models. *Modeling deltas* representing significant changes between subsequent revisions of models are first-class entities in storing the histories of model changes in model repositories. The sequential version control is referred to as *macro-versioning* in this paper.
- *Concurrent Collaboration*. 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 several concurrent instances 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 simple notations. Concurrent model instances can be differentiated by changes represented in small *modeling deltas*. The required performance of synchronization in real-time can be achieved by exchanging small modeling deltas [KW15]. Real-time synchronization of modeling deltas is referred to as *micro-versioning*.

Sequential and concurrent version control are the key activities of sequential and concurrent collaborative modeling, respectively. In both activities, *modeling deltas* are 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 modeling deltas is crucial.

An *efficient change representation notation* is needed for both micro-versioning and macro-versioning. Both versioning techniques might rely on the same base-technology to deal with modeling deltas. To that end, this paper introduces a difference language (DL) to the problem of model difference presentation in modeling deltas. The proposed DL is meta-model generic, operation-based, modeling tool generic, reusable, applicable, and extensible. Associated technical support also focuses on providing a catalog of supplementary services which allow for reusing and exploiting modeling deltas represented by DL. These supplementary services extend the application areas of DL.

The remainder of this paper is structured as follows: Section 2 investigates existing related approaches in the research domain. Several requirements for DL and its collaborative modeling application are defined in Section 3. The core concepts behind the DL-based collaborative modeling are depicted in Section 4 and the same section explains some DL services. Section 5 explains collaborative modeling applications. Section 6 discusses the evaluation of the collaborative modeling application in various modeling languages. This paper ends up in Section 7 by drawing some conclusions.

2 Related Work

The problem of collaborative modeling and its change representation is the actively discussed and extensively addressed topic among the research community of software engineering. There is a large number of research papers addressing to collaborative modeling (Section 2.1) and model difference representation (Section 2.2).

2.1 Collaborative Systems

Collaborative approaches are widely investigated in collaborative document writing and code-driven software development. Thus, this section also reviews some collaborative source code development and document editing systems in order to derive some general concepts and principals.

Sequential collaborative systems such as Git [Sw08], Mercurial [Ma10], Subversion [CSFP04] are used for sequential revision control in source code-driven software development. There are also sequential collaborative modeling systems such as EMF Store [HK13], SMoVer [Al07], AMOR [Br10] and [Ta12] that are discussed below.

Concurrent collaborative text editing systems record and synchronize change notifications during concurrent collaborative development. Collaborative documents writing systems like Google Docs [Go17] and Etherpad [Ap17] are widely used systems in concurrent document creation and editing. There are also several browser-based web modeling tools like GenMyModel [DMA13], Creately [Ci15] which exchange changes over WebSockets. 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.

For differentiating revisions and calculating differences, the most collaborative systems for source code-driven software development use Myer's LCS [My86] algorithm which is based on recursively finding the longest sequence of common lines in the list of lines of compared revisions. The core idea behind this algorithm is to represent differences documents by additions, removals, and matches of textual lines. Software models can also be represented in textual formats using XMI exchange formats, but it is commonly agreed that the collaborative approaches for source code cannot sufficiently fit to MDSD because of the paradigm shift between source code- and model-driven concepts [CRP07], [St08]. Differentiating the textual lines of the XMI-based software models can not provide sufficient information about the changes in associated and composite data structures of software models. As there are already outstanding collaborative systems for textual documents and the source code of software systems, MDSD also requires support for generic, solid, configurable and extensible collaborative modeling applications for their development, maintenance, and evolution.

2.2 Modeling Delta Representation Approaches.

The existing collaborative modeling approaches employ various techniques for model difference representation. Below, the existing approaches are classified and discussed.

Model-based approaches represent modeling deltas by differences models. Cicchetti et al. [CRP07] introduced a meta-model independent approach. The approach uses software models for representing model differences conforming to a differences meta-model. Cicchetti et al. also provides a service to apply differences models to differentiated models in order to transform them from one revision to another. A fundamental approach to sequential model version control based on graph modifications introduced by Taentzer et. al. [Ta12] is also used to represent model differences by differences models. The approach is validated in Adaptable Model Versioning System (AMOR) [Br10] for EMF models [St08]. The major focus of the approach is differentiation and merging of software models that serve as the main foundations for sequential model version control.

Graph-based approaches represent model differences using internal graph-like structures. It is usually similar to model-based representation, but relying on the low-level graph-like structures. A generic approach to model difference calculation and representation using edit scripts is introduced by the SiDiff approach [Ke13]. SiDiff consists of a chain of model differencing processes which include correspondence matching, difference derivation, and semantic lifting [Ke12]. SiDiff does not rely on a specific modeling language.

Standard ER Database represents model differences in standard entity-relational (ER) databases. Likely, SMOVER [Al07] is a sequential revision control system for software models using ER database.

Text-based approaches represent model differences by a sequence of edit operations in the textual forms embedding change-related difference information. An early approach is introduced by Alanen and Porres [AP03] in the text-based difference representation area. DeltaEcore [SSA14] is a delta language generation framework and addresses the problem of generating delta modeling languages for software product lines and software ecosystems. EMF Store [HK13] is a model and data repository for EMF-based software models. The framework enables collaborative work of several modelers directly via peer-to-peer connection providing semantic version control of models.

Operation-based. All of these approaches identify themselves as the operation-based difference representation. They usually use 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 of their difference representation techniques, they employ these basic operations only for recognizing the type of model changes, but information about model changes is generally stored in various forms as discussed above.

Lessons Learned.
This section discusses the existing related approaches based on the criteria whether representation by these approaches can provide small modeling deltas for both sequential and concurrent collaborative modeling. The most approaches focus on only some aspects of (or only one of) these collaborative modeling scenarios.

Software models themselves are too complex and structured for modeling tool developers and users. Modeling deltas represented by models or graphs usually consist of additional conceptual information for representing its modeling or graph concepts alongside actual difference/change information. In this case, model- or graph-based modeling deltas might not be as small as text-based modeling deltas. Moreover, if model differences are again represented by models or graphs, further extension of this class of approaches might require more knowledge and effort in developing further services on the top. *Model- and graph-based difference representations* are unlikely to show high performance in concurrent collaborative modeling because of their complex data structures which causes difficulties in rapid synchronization in real-time. During the evolutionary life-cycle, if all difference information is stored in a database, the database becomes very complex and huge with an associated and entity-relational data set. In *database-based representation*, modeling deltas may not easily be identified and reused.

Modeling deltas represented in *textual forms* are the most likely to be small, efficient and well-suited for collaborative modeling for many reasons: (1) directly executable descriptions of model differences; (2) easy to implement; (3) fully expressive, yet unambiguous providing necessary knowledge; (4) easy to synchronize with high performance; (5) minimalistic in comparison to model-, graph- and database-based representations.

Literature reviews show that research on difference representation for collaborative modeling is still in its infancy. Considering the aforementioned discussion, collaborative modeling requests a textual *difference language (DL)* to represent modeling deltas in sequential and concurrent collaborative modeling.

3 Requirements

As long as there are several modeling languages and model designing tools, collaborative modeling approaches should not rely on a specific modeling language or modeling tool. Collaborative modeling has to support both sequential (macro-versioning) and concurrent (micro-versioning) collaboration forms. Since difference representation lies at the core of collaborative modeling, this section defines several requirements that a proposed DL-based difference representation has to fulfill.

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 notation. In order to provide collaborative modeling on such graphical modeling tools, DL has to be aware of layout notation together with the meta-models (content) of modeling languages. For instance, if a designer changes the position and size of a modeling artifact in a graphical editor, the same changes have to simultaneously occur in the modeling editors of other tool instances as well.

Genericness. There are several modeling languages and graphical notations following diverse formal specifications and concepts. DL has to be generic with respect to the meta-models of modeling languages and their layout without restricting itself to a particular modeling language or tool. If DL is generic, its collaborative modeling support can then be tailored to the wide range of modeling languages and tools.

Supportiveness. As defined in Section 1, collaborative modeling forms two scenarios such as concurrent *micro-versioning* and sequential *macro-versioning*, whereas difference representation is a common and fundamental concern for both. The most approaches investigated in Section 2 focus on only some aspects of these collaborative modeling scenarios. DL has to support both scenarios of collaborative modeling by being applicable, persistent, implementable and expressive.

These requirements are also partly addressed to in [CRP07], [HK10], [SBG08]. They are proper characteristics for emerging an appropriate difference representation approach for collaborative modeling in MDSD. 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 empowering the qualification and solidity of difference representation. DL aims at fulfilling these requirements throughout this paper.

4 Modeling Deltas

In collaborative modeling, changed modeling artifacts have to be properly identified and represented in modeling deltas for further processing of software models. A *model change* defines any kind of change made to a modeling artifact during its evolutionary life-cycle. Modeling deltas including model changes are the core entities in representing model changes and building collaborative modeling on top. DL focuses on representing model changes in modeling deltas. Section 4.1 demonstrates the overall conceptual idea of DL. Section 4.2 gives a very simplified example of DL-based change representation. Section 4.3 explains a subset of supplementary DL services required for collaborative modeling.

4.1 Conceptual Idea

In order to use the collaborative modeling environment, specific DLs have to be generated from the meta-models of modeling languages. DL is a generic approach with respect to the meta-models of modeling languages. It is conceptually a family of domain-specific languages. As long as the modeling concepts of any modeling language can be recognized by looking at the meta-model of that language, a specific DL for a particular modeling language is generated by the DL generator service (explain in Section 4.3) importing its meta-model. Then, modeling deltas can be represented in terms of DL for instance models conforming to that modeling language.

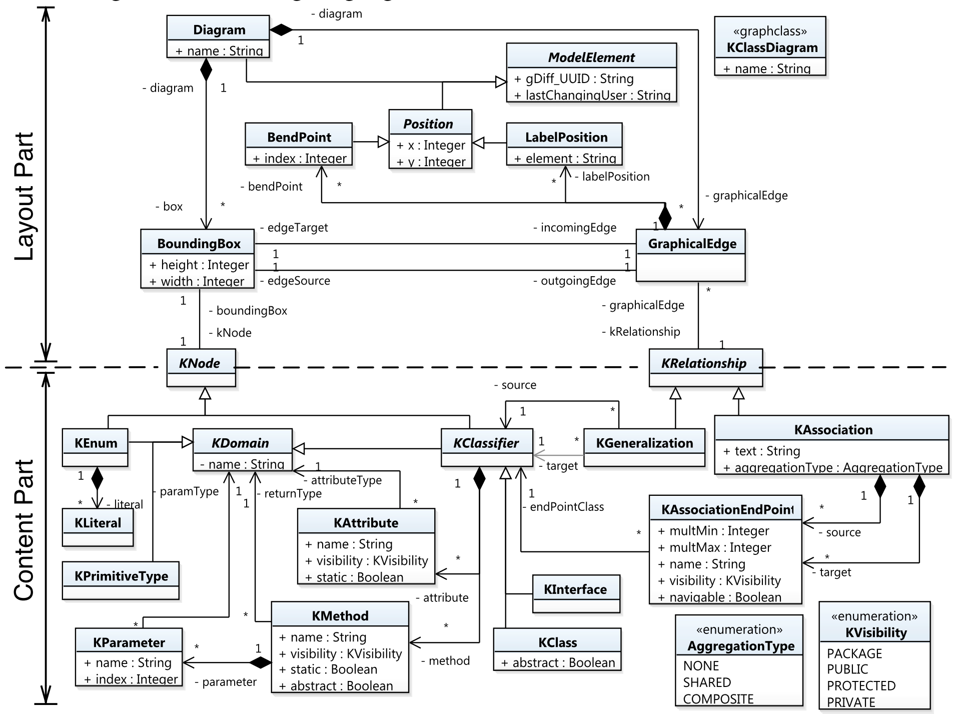


Fig. 1: UML Class Diagram Meta-model

Figure 1 depicts the meta-model of a subset of UML class diagrams which is used throughout this paper as an example. The meta-model is separated into two parts by a dashed line. Below of 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 notation for layout information for the conceptual part. This way of designing meta-models is another advantage of the DL approach satisfying the *Awareness of Content and Layout* requirement. This allows for using the same collaborative modeling environment for different modeling content. The complete meta-model is used for creating overall collaborative modeling application explained in Section 5.

4.2 Motivating Example

In order to explicitly explain how changes are represented in terms of DL, this section presents a simplified example of the DL-based change representation in modeling deltas. A very simple UML class diagram [RJB04, pp. 47ff] is chosen as a running example to apply the DL approach. Figure 2 depicts two subsequent revisions of the same class diagram namely `Rev_1` and `Rev_2`, they conform to the meta-model depicted in Figure 1. Figure 2 further portrays two concurrent instances of the latest revision, in this case, `Rev_2`. Two designers, namely `Designer_1` and `Designer_2`, are working on these concurrent instances.

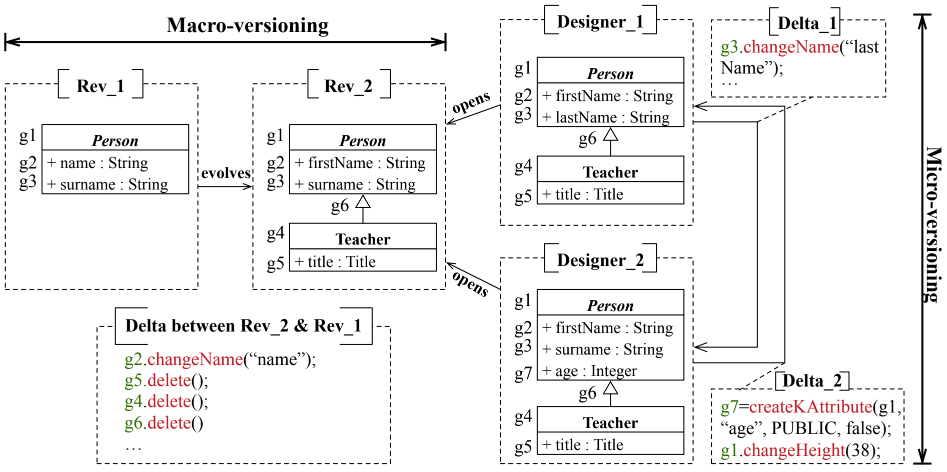


Fig. 2: Modeling Deltas in Collaborative Modeling

In the first revision, the model has only one class named `Person`. While evolving from the first revision to the second, the following changes are made: a class with the name `Teacher` is added and it is generalized to the existing class `Person`, the attribute `name` of the class `Person` is changed to `firstName`.

According to the `ModelElement` class of the meta-model in Figure 1, each model element has an attribute named `gDiff_UUID` which means all model elements are assigned to *universally unique identifiers (UUID)*. Therefore, each model element in this example also has a persistent identifier. Assigning UUID to modeling artifacts allows for identifying and keeping track of modeling artifacts over time. Any particular modeling artifact can be traced by detecting the predecessor and successor artifact of that modeling artifact (inter-delta references). The persistent identifiers are always used in the DL operations in order to preserve consistency of modeling deltas. They are also used as indicators to refer to modeling concepts from the DL operations in modeling deltas (delta-model references).

The DL operations representing modeling deltas in Figure 2 conform to a specific DL generated from the meta-model depicted in Figure 1. Each delta operation consists of a `Do part` (cf. *g2.changeName("name");*) which describes the *kind of change* by means of *operations* (one of create, change, delete) and an `Object part` (with attributes if required) (cf. *g2.changeName("name");*) which refers to the modeling concept. The modeling deltas between the subsequent revisions refer to macro-versioning and the modeling deltas between the two concurrent instances refer to micro-versioning.

Each object of any modeling language can be created, deleted or attributes of each object can be changed during the evolution process. DL defines model changes by three basic operations such as `creation`, `deletion` of modeling artifacts or `change` of the attributes of these artifacts. These basic operations are sufficient set of operations to represent any kind of model changes by DL.

Macro-versioning. In macro-versioning (cf. Subversion [CSFP04], Git [Sw08], Mercurial [Ma10]), the differences between subsequent revisions are usually identified and represented in reverse order i.e. they represent changes in the *backward deltas* [KW15]. Because these systems intend to store differences as directly executable forms which are more practical in retrieving the earlier revisions of software systems. They store the most recent revision of software systems and several differences deltas for tracing back to the initial revision. Hence, the latest revision is the most frequently accessed revision. DL-based difference representation also follows the similar art of delta representation. Thus, the modeling delta between the first and second revisions is represented in the backward delta (`Delta between Rev_2 and Rev_1`). Modeling deltas are directly executable descriptions of model differences i.e. application of the modeling delta to the second revision results in the first revision. When the delta `Delta between Rev_2 and Rev_1` is applied to the revision `Rev_2`, the first operation changes the attribute `firstName` of the class `Person` to `name`, and the following deletion operations delete the attribute of the class `Teacher`, the class `Teacher` itself and the generalization assigned to `g6`. For the sack of simplicity, the delta depicts the conditional full-stops (...) on the last line, meaning the delta consists of other DL operations for changing layout information.

Micro-versioning. In the second revision, the model is then being further developed by two designers concurrently. `Designer_1` changes the attribute of the class `Person` from `surname` to `lastName`. This change is then sent to the other instance as a `Delta`. On the other instance, `Designer_2` creates a new attribute with name `age` in the class `Person`. That change is sent to the instance, `Designer_1` is working on, as a `Delta`. When the new attribute is added, the height of the class `Person` is increased and this change of layout information is also described by DL operation (`g1.changeHeight(38);`) in the same delta. Both of these deltas are represented as *forward* deltas in this case. Because the latest changes have to be reflected on the parallel models, thus, changes have to be applied to models in forward order.

4.3 DL Services

In collaborative modeling, several services are involved in generating specific DLs, calculating and applying the DL-based modeling deltas. As all DL services are explained in [KW15], this section briefly revisits the services which are involved in collaborative modeling.

DL Generator. DL generator generates a specific DL for a particular modeling language by importing its meta-model. While generating the specific DL, it inspects all of the concrete (i.e., non-abstract) meta-classes of a given meta-model and the attributes of these meta-classes. For each meta-class, it generates interfaces with implementations for creation and deletion operations, as well as for change operations for each attribute. Modification of the attribute named `gDiff_UUID` (e.g. the class *ModelElement* in Figure 1 is not allowed by default as it tightly defines the identity of a modeling artifact, therefore, should not be changed as the part of model changes. A specific DL is always generated in the form of a Java Interface.

After generating a specific DL for the given meta-model, several DL services still have to be involved in collaborative modeling as depicted in Figure 3. It depicts an abstract architecture of collaborative modeling including *server* and *client* sides.

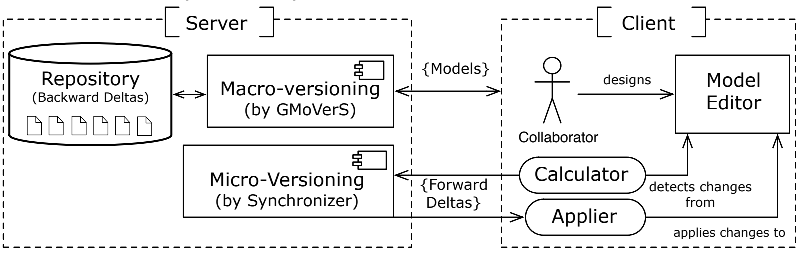


Fig. 3: Overall Architecture of Collaborative Modeling

The server-side consists of the concurrent synchronization service *micro-versioning* and sequential version control service *macro-versioning* which uses the DL-based modeling delta repository. On the client side, while collaborators are designing models using a model editor, their changes are constantly detected by the *calculator* service and sent to other clients as modeling deltas. Once these deltas arrive at other clients, they are applied to models by the *applier* service. During collaboration, designers may to store a particular state of their model whenever their model is complete and correct. They are able to manage models and load any revisions of their model which they have saved earlier. The macro-versioning feature is provided by the DL application GMoVerS (Generic Model Versioning System) [KW15].

Calculator. Calculation of modeling deltas depends on the form of collaboration whether it is sequential (macro-versioning) or concurrent (micro-versioning) collaboration. In micro-versioning, they are calculated by listening for changes in models by *change listeners* [HK10]. Because changes have to be synchronized in real-time providing sufficient performance. In macro-versioning, modeling deltas between subsequent model revisions are calculated using *state-based* comparison of subsequent revisions [Ke13]. The state-based comparison and change listener are two different features of the DL delta *calculator* service. For instance, all modeling deltas in Figure 2 are calculated by this service. The modeling delta `Delta between Rev_2 and Rev_1` is computed by the *state-based* comparison (implemented using SiDiff algorithm [Ke13]), whereas the modeling deltas on both instances of `Designer_1` and `Designer_2` is computed by the *change listener* in real-time. The state-based comparison feature can calculate both, *forward* and *backward* deltas, whereas change listener can calculate only *forward* deltas.

Applier. In collaborative modeling, modeling deltas are applied to the base model in order to transform them from one revision to another. Application of modeling deltas to the base models is provided by the DL delta *applier* service [KW15]. In case of loss or damage of information on the initial copy of a model, designers might feel a need to revert their model for obtaining earlier versions or undoing recent changes they made. For instance, in Figure 2, the applier applies the `Delta between Rev_2 and Rev_1` to `Rev_2` to revert it to `Rev_1`. The applier is also employed in micro-versioning in order to propagate model changes on the concurrent instances of a shared model. In Figure 2, the applier is used to apply both `Delta_1` and `Delta_2` to the instances of `Designer_1` and `Designer_2`, respectively.

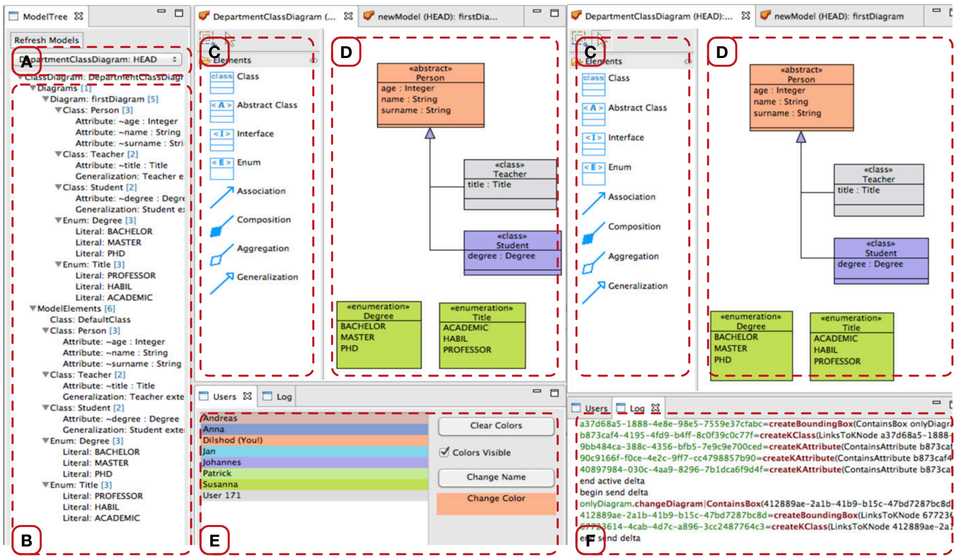
The both, DL calculator and applier, services follow the general principles and do the same core operating tasks of difference calculator and applier for textual version control systems. But, their realization follows modeling concepts i.e. the DL calculator and applier operate on the composite and graph-like software models, whereas the latter operate on the textual documents.

5 Application: Kotelett

The collaborative modeling applications are established by the specific orchestrations of the DL services and on the top of the DL-based difference presentation. This section explains the collaborative modeling application *Kotelett* and other ongoing work.

The collaborative modeling application entitled *Kotelett tool* was initially developed by a students project group in the Software Engineering Group at the Carl von Ossietzky University of Oldenburg. Kotelett takes advantage of the DL-based modeling deltas for exchanging changes among various collaborators of the shared model. Figure 4 depicts the overall user interface of Kotelett. It displays two independent tool instances working on the same model concurrently. Each Kotelett instance consists of several windows as explained below. 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. But, the users can open multiple models during collaboration.

In micro-versioning, modeling deltas are not stored for reverting changes. However, reversion of changes in micro-versioning happens on the client side of the editor and provided by the



```
ClassDiagram: DepartmentClassDiagr
  Diagrams [1]
    Diagram: firstDiagram [5]
      Class: Person [3]
        Attribute: ~age : Integer
        Attribute: ~name : String
        Attribute: ~surname : Stri
      Class: Teacher [2]
        Attribute: ~title : Title
        Generalization: Teacher e
      Class: Student [2]
        Attribute: ~degree : Degre
        Generalization: Student e:
      Enum: Degree [3]
        Literal: BACHELOR
        Literal: MASTER
        Literal: PHD
      Enum: Title [3]
        Literal: PROFESSOR
        Literal: HABIL
        Literal: ACADEMIC
  ModelElements [6]
    Class: DefaultClass
    Class: Person [3]
      Attribute: ~age : Integer
      Attribute: ~name : String
      Attribute: ~surname : String
    Class: Teacher [2]
      Attribute: ~title : Title
      Generalization: Teacher exte
    Class: Student [2]
      Attribute: ~degree : Degree
      Generalization: Student exte
    Enum: Degree [3]
      Literal: BACHELOR
      Literal: MASTER
      Literal: PHD
    Enum: Title [3]
      Literal: PROFESSOR
      Literal: HABIL
      Literal: ACADEMIC
```

```
a37d68a5-1888-4e8e-98e5-7559e37cfabc=createBoundingBox(ContainsBox onlyDiagra
b873caf4-4195-4fd9-b4ff-8c0f39c0c77f=createKClass(LinksToKNode a37d68a5-1888-
9bb484ca-388c-4356-bfb5-7e9c9e700ced=createKAttribute(ContainsAttribute b873ca
90c9166f-f0ce-4e2c-9ff7-cc4798857b90=createKAttribute(ContainsAttribute b873caf4
40897984-030c-4aa9-8296-7b1dca6f9d4f=createKAttribute(ContainsAttribute b873ca
end active delta
begin send delta
onlyDiagram.changeDiagram|ContainsBox(412889ae-2a1b-41b9-b15c-47bd7287bc8d
412889ae-2a1b-41b9-b15c-47bd7287bc8d=createBoundingBox(LinksToKNode 67723
67723614-4cab-4d7c-a896-3cc2487764c3=createKClass(LinksToKNode 412889ae-2a1
end send delta
```

Fig. 4: Kotelett Screenshot

Redo/Undo features of the editor. Kotelett can save the model at a particular time and by clicking a save button whenever the model is complete and correct. When the tool is asked to save the model, it calculates the differences (backward deltas for macro-versioning) between the last and current revisions. It then stores these differences in the form of the DL-based modeling delta. On the left instance of the Kotelett screenshot, the pop-up menu *model browser (A)* allows for selecting and opening any version of the model which was saved previously. The menu lists all automatically and manually saved versions of the current model. Once any previous revision is selected, that revision is reverted by the *DL applier* service by applying sequential modeling deltas to the base model.

The *model tree (B)* shows the list of diagrams the user is currently working on. Each diagram belongs to a specific model. It also shows the list of model elements that are created in the current diagram. The *model editor (D)* area is the main part which allows users for designing UML class diagrams. The *Elements (C)* part lists the most important notations of UML class diagram where the users can select and draw that element in the model editor. These notations of the UML class diagram are created based on the meta-model depicted in Figure 1. The correctness of the model on this editor is checked according to that meta-model, automatically. The *user list (E)* (left instance) window lists all users that are currently working on the initial diagram. These users are highlighted with different colors in order to show the clear distinction between them and to recognize which change is made by which user in the editor. The *log (F)* window (right instance) constantly displays the modeling deltas that are exchanged among collaborators after each user action. Creating one model element on the graphical modeling editor may result in one or several change operations that are contained in one modeling delta and synchronized between collaborators. Modeling deltas are represented by a specific DL generated from the meta-model in Figure 1.

The JGraLab (Java Graph Laboratory) [DW98] technical space is used for realizing DL in Kotelett. TGraph [ERW08] is used to represent software models internally, whereas the TGraph schema used to define meta-models. JGraLab further provides generic features for defining in-place model manipulations to implement the DL-based operations. The architectural and implementation concepts of Kotelett is explained in [KW15] in detail.

DL in UML Designer.
As ongoing work, the collaborative modeling approach is being applied to UML designer which is an EMF- and Sirius-based domain-specific modeling tool [Ob17]. DL will be applied to UML designer using the EMF technical space. To represent model changes in UML designer, a specific DL will be derived from the EMF-based Ecore meta-model [St08]. The delta calculator and applier services will be extended by EMF transactional editing domain and command stack extensions. All other underlying technologies remain the same and unchanged.

6 Evaluation

The collaborative modeling approach in this paper is validated to achieve its intended goals by fulfilling the requirements identified in Section 3. More specifically, these requirements are revisited in this section for presenting their fulfillment by DL. All DL concepts, services and applications provide evidence to the fulfillment of each of these requirements. Finally, the extend-ability of the approach is shown in this section.

6.1 Validation

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 server located in Germany.

During these experiments, the tool has shown sufficiently high performance by synchronization of small DL-based modeling deltas. So far Kotelett did not face any change conflicts in micro-versioning. This probably is attributed to the rapid synchronization of small modeling deltas. Exchanging binary formats of deltas can be faster, but so far it was not necessary. It can be experimented for optimization purposes. But, it probably might not be expressive enough making it difficult for further extentions. For merging various model revisions in macro-versioning, Kotelett aims at employing the existing *merge* feature provided by the *JGraLab* technical space [DW98].

In order to accomplish a solid, appropriate, generic and efficient collaborative modeling approach, this paper has kept the requirements defined in Section 3 in its focus and thoroughly attempted to satisfy them. Based on the layout notation part and modeling language notation part of meta-models (e.g. Figure 1), the collaborative modeling environment is aware of content and layout of models. The same collaboration environment can be used for modeling language and layout notations (*Awareness of Content and Layout*), simultaneously. The approach is generic with respect to the meta-models of modeling languages. The collaborative modeling environment can be applied to wide range of modeling languages and tools by generating specific DLs by the DL generator explained in Section 4.3. It then can import any meta-model defining modeling concepts and layout notation and generates specific DL for it (*Genericness*). If meta-models are changed, they can be re-imported and specific DLs can be regenerated newly for changed meta-models.

The collaborative modeling approach is applied to Kotelett and other applications under development. These applications support concurrent (by *micro-versioning*) and sequential (by *macro-versioning*) collaborative modeling (*Supportiveness*). They are developed on top of DL which provides more efficient ways of representing, exchanging and synchronizing modeling deltas improving the performance of data processing. DL is utilized as solid and common syntactic ground for representing modeling deltas in these applications.

6.2 Extendability

In the approach, existing functionality can be extended, new features can be added by extending underlying technologies and concepts such as the DL services. The collaborative modeling approach can represent software models under collaboration using internal graph-like structures [ERW08] and graphical editors can be separated from that graph representation. This allows for extending graphical editors without changing the underlying concepts and technologies of collaborative modeling. This enables tool developers to define further graphical notations, shapes, and views as they want and reuse existing collaborative modeling environment without any further development effort. As depicted in Figure 1, the layout notation part (above the dashed line) of the meta-models enables extendability of the approach for further modeling languages with the same layout notation. The modeling concept part (below the dashed line) of the meta-model should be replaced by the meta-model of another modeling language. Eventually, the same layout information can be reused for further modeling languages.

The catalog of the DL services can also be replaced by other implementations and extended with further services or features. Any service, component or plug-in required for collaborative modeling can easily be developed and registered in the service catalog which is practically useful for further service-oriented modeling tool development. The only prerequisite for these services is to recognize the syntax of DL. Eventually, these services can again be involved in service orchestrations for establishing the collaborative modeling applications.

7 Conclusions

The approach proposed in this paper is meta-model generic, aware of both content and layout of models and supports sequential and concurrent collaborative modeling. As *modeling deltas* are first-class entities and play an essential role in storing, exchanging and synchronizing changes between the subsequent and concurrent revisions of evolving models, representation of modeling deltas is very crucial in both forms of collaboration. Thus, the collaborative modeling approach is elaborated on top of the efficient difference language notation which is employed in both micro-versioning and macro-versioning. Both forms rely on the same base difference representation language for modeling deltas.

A proposed operation-based DL serves as a common change representation and exchange format for collaborative modeling, making software models commonly available to multiple users in different locations. Micro-versioning achieves high performance of synchronization by exchanging small DL-based forward deltas, whereas macro-versioning can effectively store the change histories of evolving modeling artifacts in model repositories as the DL-based backward deltas. These and further collaborative modeling scenarios can be developed by specific orchestrations of the DL services which can produce, store, exchange, synchronize and apply modeling deltas in collaborative modeling. Kotelett can be downloaded at `https://pg-kotelett.informatik.uni-oldenburg.de:8443/build/stable/` and presented at the conference as well.

References

[Al07] Altmanninger, K.; Bergmayr, A.; Schwinger, W.; Kotsis, G.: Semantically enhanced conflict detection between model versions in SMoVer by example. In: Procs of the Int. Workshop on Semantic-Based Software Development at OOPSLA. 2007.

[AP03] Alanen, M.; Porres, I.: Difference and union of models. In P.Stevens, J.Whittle, and G. Booch, editors, Proc. 6th Int. Conf. on the UML, Springer, LNCS 2863:2–17, 2003.

[Ap17] AppJet Inc.: , Etherpad. http://www.etherpad.com, visited on 01.02.2017.

[Br10] Brosch, P.; Kappel, G.; Seidl, M.; Wieland, K.; Wimmer, M.; Kargl, H.; Langer, P.: Adaptable Model Versioning in Action. in: Proc. Modellierung 2010, Klagenfurt, Österreich, LNI 161:221–236, March 24-26 2010.

[Ci15] Cinergix Pty.: , CreateLy. http://www.creately.com, visited on 01.06.2015.

[CRP07] Cicchetti, A.; Ruscio, D.; Pierantonio, A.: A Metamodel independent approach to difference representation. Journal of Object Technology, 6:9:165–185, October 2007.

[CSFP04] Collins-Sussman, B.; Fitzpatrick, B.; Pilato, M.: Version Control with Subversion. O'Reilly Media, June 2004.

[DMA13] Dirix, M.; Muller, A.; Aranega, V.: GenMyModel: UML case tool. In: ECOOP. 2013.

[DW98] Dahm, P.; Widmann, F.: Das Graphenlabor. Technical Report 11/98, Universität Koblenz-Landau, Koblenz, 1998. Fachberichte Informatik.

[ERW08] Ebert, J.; Riediger, V.; Winter, A.: Graph technology in reverse engineering. The TGraph approach. In: 10th Workshop Software Reengineering. GI LNI. pp. 23–24, 2008.

[ESG91] Ellis, C.; Simon, G.; Gail, R.: Groupware: Some Issues and Experiences. ACM, 34(1):39–58, 1991.

[Go17] Google Inc.: , Google Docs. http://docs.google.com, February 2017. online.

[HK10] Herrmannsdoerfer, M.; Koegel, M.: Towards a generic operation recorder for model evolution. In: Proceedings of the 1st International Workshop on Model Comparison in Practice. ACM, pp. 76–81, 2010.

[HK13] Helming, J.; Koegel, M.: , EMFStore., 2013. http://eclipse.org/emfstore.

[Ke12] Kehrer, T.; Kelter, U.; Ohrndorf, M.; Sollbach, T.: Understanding model evolution through semantically lifting model differences with SiLift. In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on. IEEE, pp. 638–641, 2012.

[Ke13] Kehrer, T.; Rindt, M.; Pietsch, P.; Kelter, U.: Generating Edit Operations for Profiled UML Models. In: MoDELS. pp. 30–39, 2013.

[KW15] Kuryazov, D.; Winter, A.: Collaborative Modeling Empowered by Modeling Deltas. ACM, Lausanne, Switzerland, 09 2015.

[KWB03] Kleppe, A.; Warmer, J.; Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2003.

[Ma10] Mackall, M.: The mercurial scm. Internet Website, last accessed 05.07.2017, 2010.

[My86] Myers, E. W.: An O (ND) difference algorithm and its variations. Algorithmica, 1(1):251–266, 1986.

[Ob17] Obeo Network: , UML designer. http://www.umldesigner.org/, visited on 02.10.2017.

[RJB04] Rumbaugh, J.; Jacobson, I.; Booch, G.: Unified modeling language reference manual. Pearson Higher Education, 2004.

[SBG08] Sriplakich, P.; Blanc, X.; Gervals, M.: Collaborative software engineering on large-scale models: requirements and experience in modelbus. In: Proceedings of the 2008 ACM symposium on Applied computing. ACM, pp. 674–681, 2008.

[SSA14] Seidl, C.; Schaefer, I.; Aßmann, U.: DeltaEcore-A Model-Based Delta Language Generation Framework. In: Modellierung. pp. 81–96, 2014.

[St08] Steinberg, D.; Budinsky, F.; Merks, E.; Paternostro, M.: EMF: Eclipse Modeling Framework. Addison-Wesley Longman Publishing Co., Inc., 2008.

[Sw08] Swicegood, T.: Pragmatic version control using Git. Pragmatic Bookshelf, 2008.

[Ta12] Taentzer, G.; Ermel, C.; Langer, P.; Wimmer, M.: A fundamental approach to model versioning based on graph modifications: from theory to implementation. journal: Software and Systems Modeling, April 25 2012.