Collaborative Modeling Empowered By Modeling Deltas

Dilshodbek Kuryazov, Andreas Winter
Software Engineering Group
University of Oldenburg
{kuryazov,winter}@se.uni-oldenburg.de

ABSTRACT

Evolution and maintenance of the large-scaled software models require collaboration of several designers on the shared modeling artefacts. Since collaborators manipulate shared models in real-time, synchronization of the model changes is the main challenging aspect for collaborative modeling application.

In order to achieve efficient real-time synchronization of changes, these changes have to be properly identified, represented by appropriate notations and exchanged by *modeling deltas*. This paper presents a real-time collaborative modeling application based on exchanging model differences between collaborators. Modeling deltas are represented by an operational delta notation. The approach is validated by presenting a collaborative class diagram editor.

Keywords

Model Changes, Modeling Deltas, Real-Time Collaborative Modeling

1. MOTIVATION

Maintenance and development of the large-scaled evolving software models require real-time collaboration of several designers on the shared modeling artefacts. During the collaborative modeling process, collaborators apply various changes to the shared model in parallel. Since several users collaboratively work on shared artefacts, the collaborative modeling application needs to provide sharing of modeling artefacts and synchronize the user changes among collaborators when new artefacts are created or existing ones are deleted or changed.

Collaborative modeling has to be capable of handling a huge amount of the shared modeling artefacts. Changes of shared models have to be exchanged between collaborative modelers in real-time. These changes can be viewed as *modeling deltas*. A *modeling delta* might consist of one or many changes performed by a collaborator.

Real-time synchronization of modeling deltas is also referred to as *micro-versioning*. In the case of *micro-versioning*, small changes are identified and exchanged among various real-time copies of a model. Basically, the histories of such changes are not stored in a repository, but the change history of each collaborator is available on their copy which can be reverted by *Redo/Undo* actions.

On the other hand, constantly changing models results in several different revisions of the same modeling artefact. Obviously, model designers wish to store different versions of their model and manage model versions so that the previous model versions can be reverted or the change histories can be traced when needed. This kind of versioning is referred to as *macro-versioning*. *Macro-versioning* is the standard version control of models i.e. storing subsequent versions in a repository, reverting versions and other management activities [3].

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. In the case of *micro-versioning*, a modeling delta approach provides efficient detection, synchronization and application of changes made by designers. In *macro-versioning*, each model change needs to be represented by a sufficient notation so that it allows collaborators to efficiently store differences between subsequent model versions and provides a solid management of models and their versions. The general *Delta Operations Language (DOL)*, meta-model generic and operation-based approach which is introduced in [14] is applied to model difference representations in micro- and macro-versioning. This paper presents the application of DOL to *micro-versioning* whereas usage of DOL in *macro-versioning* is explained in [14].

DOL is a set of domain-specific languages to model difference representation by operations. A specific DOL for a specific modeling language is generated from the meta-model of a modeling language. A specific DOL is then fully capable of representing model changes in terms of operations. Only the changed model elements are identified and represented in a differences document which is called *Modeling Delta*. Each modeling delta consists of the sequence of change operations (create, delete and change) for changes on a model.

In DOL, exchanging model changes made by designers is eased by exchanging only modeling deltas which contain only change operations. Exchanging small deltas provides rapid synchronization of changes by reducing the capacity of exchange data and exchange time. Additionally, the approach provides several DOL services e.g. a *difference calculator* for detecting model changes and *change applier* which are used in realizing a collaborative modeling tool.

The paper is structured as follows: Section 2 gives an overview of collaborative modeling. Section 3 explains modeling delta based approach and gives an example of the DOL-based change representation. Section 4 illustrates modeling deltas in a collaborative class diagram modeling application including the main architecture of real-time collaborative modeling tools and their components. Evaluation of the modeling tool is explained in Section 5. The paper ends up

by drawing conclusions and future work in Section 6.

2. COLLABORATIVE MODELING

The real-time collaboration principles are already investigated in collaborative document creation. Google Docs [6] and Etherpad [1] are widely used in document creation and editing in real-time. But, these web-based tools are limited to text-based synchronization relying on lines of strings. There are also web-based, commercial real-time collaborative modeling tools such as *GenMyModel* [7] and *creately* [5]. Since they are web-based tools, exchanging changes occur over *WebSocket* using web browsers and users of these tools depend heavily on the web server with Internet connection. Since they are not open-source tools, the modeling notations are not accessible to extend or replace with the user defined meta-models.

In the following, the current features of the real-time collaborative modeling tool of the DOL approach are described in detail. Figure 1 depicts a screen-shot of the DOL-based collaborative modeling application. The figure consists of two independent tool instances working on the same model in parallel. The tool user interface (UI) called *Kotelett* [16] portrays *Model Tree*, *Model Editor Area*, *User List* (on the left) and *Log* (on the right) windows. At startup the tool displays the list of models which are currently available on the repository and asks the user which model to join as a collaborator. But, the users can also open multiple models during real-time collaboration.

Model Tree shows the list of diagrams, a user is currently working on. It also shows the list of model elements that are created in the current diagram. Users can work on several diagrams simultaneously and each model might consist of several diagrams. On the top of the model tree window, the pop-up menu allows for the selection of any version of the model previously saved. The menu lists all automatically and manually saved versions of the current diagram (also referred to as *macro-versioning*).

Model Editor Area is the main part which allows users to design the UML class diagrams in the graphical editor. This graphical modeling editor lists the most important notations of UML class diagram where the users can select and draw that element on the editor. These notations of the UML class diagram are created based on the meta-model depicted in Figure 2. The syntactic and semantic correctness of the model on this editor are analysed automatically according to that meta-model. Several modeling editor tabs can be opened at the same time.

User List (left instance) lists all users that are currently working on the initial diagram. These users are highlighted with different colors in order to show clear distinction of them and to recognize which change is made by which user on the editor. As the graphical editor displays, the model elements are highlighted with the same color of the creator of that element. If an element is created by one user and changed by another user, color of the last changed user is applied to that element. Additionally, each user can change their names and select necessary color that should appear on the *Kotelett UI*.

Log (right instance) constantly displays the modeling deltas that get exchanged among collaborators after each user action. Creating one model element on the graphical editor may result in one or several change operations that are contained in one modeling delta and synchronized with other copies. All of these changes are represented by DOL notation introduced in Section 3. 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*. Other tool instances receive these deltas as change requests to apply on their models.

3. MODELING DELTAS

A number of approaches are already introduced to represent model differences in modeling deltas. Model-based difference representation are introduced in [4] and [11]. Operation-based difference representation for software product lines is introduced in [9]. These approaches also use basic edit operations such as *create*, *delete* and *change* to describe model modifications, but model changes are again represented by software models. These model-based representation techniques are successfully applied in standard model versioning, but they are not applied to the real-time collaborative modeling so far.

One of the very few synchronization approaches is introduced in [13] which aims at model-based real-time synchronization of the model changes. The approach is presented as an extension of the EMF-Store platform [12]. In the EMF-Store platform, the model changes made by collaborators are directly attached to modeling objects and combined in the change packages. These change packages are synchronized among collaborators using the peer-to-peer connections.

The DOL approach provides synchronization of small DOL operations through a centralized and shared repository. EMF-Store uses the graph-like structures of EMF (Eclipse Modeling Framework) [19] modeling objects as internal model representation and the changes are directly made on the EMF models. The DOL approach provides separation of the graphical editor from the internal graph-like representation of models (Figure 5). The EMF-Store platform supports only the tree based model editing feature, whereas the *Kotelett* tool provides the graphical designing editor.

Conceptually, DOL is a family of operation-based languages to model change representations. A specific DOL for a particular modeling language is generated from the meta-model of a modeling language, but the approach is completely independent from any specific meta-model i.e. the approach is applicable to all modeling languages defined by their according meta-models.

Since the ideas behind the DOL-based model change representation approach is already described in [14] in detail, only some relevant parts of the approach and a simple example of DOL-based change representation is explained in the following subsections.

3.1 DOL Approach

In the parallel modeling tools, the huge and complex models are shared among several collaborators. Identification and synchronization of user changes on such models require the quite efficient and fast change detection and synchronization techniques. Every single change needs to be represented by a very simple notation so that the collaboration process does not perform any delays or inconsistencies in real-time. Considering these prerequisites, the DOL approach offers simple operation-based notation to exchange model changes.

Since constructions of modeling languages are defined

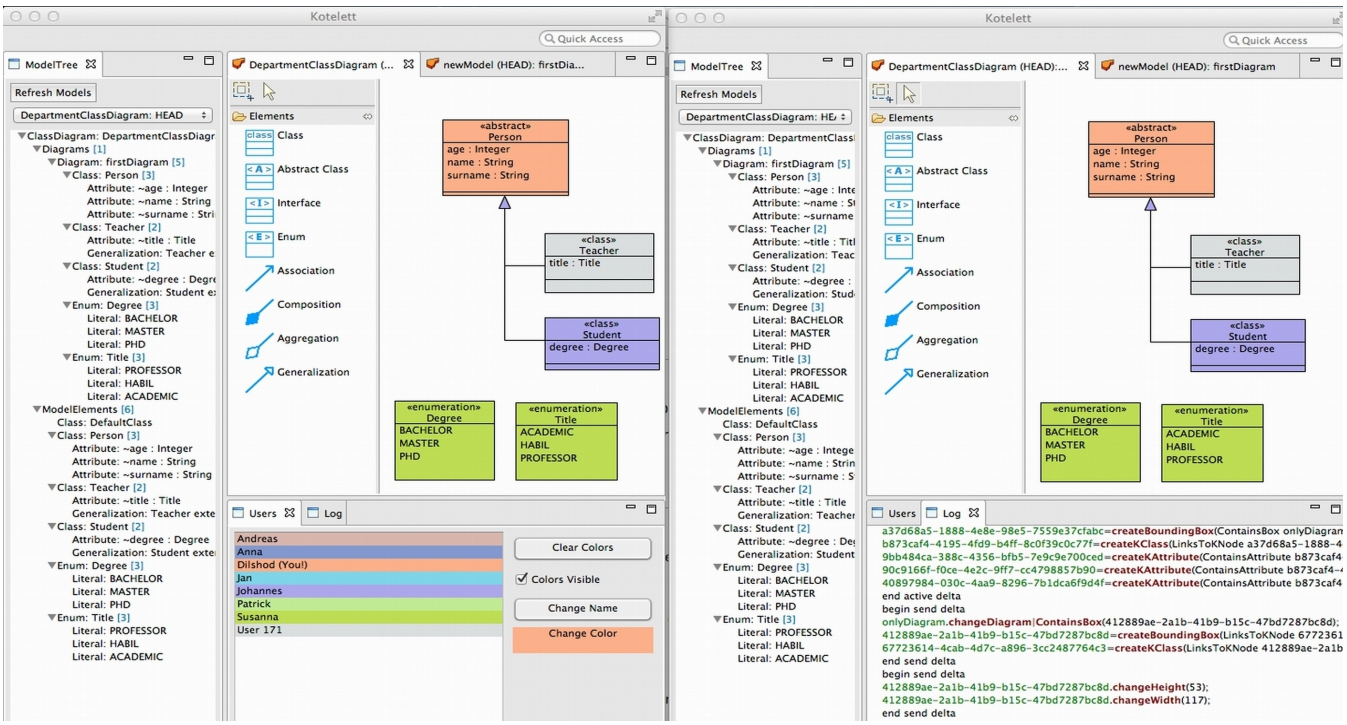

Figure 1: Kotelett Collaborative Modeling Tool

by their meta-models, the meta-model of that language is employed in order to derive a specific DOL notation for a particular modeling language. The *DOL generator* generates a particular DOL as the initial step in the DOL approach. After having a DOL notation to represent changes for the present modeling language, the user changes on instance models can be represented by DOL statements.

DOL Generation. The DOL generator receives the meta-model of a modeling language and generates a specific DOL for representing changes on the instance models conforming to that meta-model. A specific DOL is generated in the form of Java interface which receives change information as parameters and produces the DOL statements for that change.

Each object of any modeling language can be created, deleted or attributes of each object can be changed during the evolution process. Therefore, the DOL generator applies three basic edit operations such as `create`, `delete` to each concept of the given meta-model and `change` to each attribute of each concept. The DOL approach argues that these three atomic changes are sufficient to represent all kind of model changes [14]. The resulting DOL operations are also directly executable descriptions of model changes so that the changes represented by the DOL terms can easily be applied to models to transform the model from one state to another. Section 3.2 gives a concrete and simplified example of the DOL notation.

DOL Services. In collaborative modeling, changes of these local models have to be calculated and have to be sent to other collaborators and changes of other models have to be applied to the local model. In order to fulfil these tasks, two of the DOL services are utilized in the collaborative modeling application: *Change calculator* and *Change applier*. The *change calculator* detects the model changes made by designers and produces modeling deltas representing those in DOL notation. The *change applier* is capable of applying modeling deltas to a given model. These DOL services are explained in Section 4 in detail.

3.2 Applying DOL to Collaborative Class Diagram Modeling

This section demonstrates a simplified example of the DOL-based model change representation. The meta-model (Figure 2) of UML class diagrams [17] is used as a running example in this section together with a simple instance model in two parallel versions depicted in Figure 3.

The meta-model has two main parts separated by the dashed line. In graphical modeling, every modeling object has design information such as color, size and position, also called *layout information*. The notation for layout information is provided by the meta-model on the upper part of the dashed line which is used in realizing the *Kotelett* tool. Every modeling object 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`.

Usually, layout information is not depicted in classical meta-models. 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 modeling data and layout data.

The bottom part of the meta-model depicts the UML class diagram notation. The complete meta-model is used for creating overall collaborative modeling application explained in Section 4.

Figure 3 displays two different parallel copies of the same simple UML class diagram conforming to the meta-model in Figure 2. In order to express the DOL-based change representation approach, the copy of the *Client A* is considered as the changed copy of the model and the copy of the *Client B* is unchanged copy of the model. The example in this section

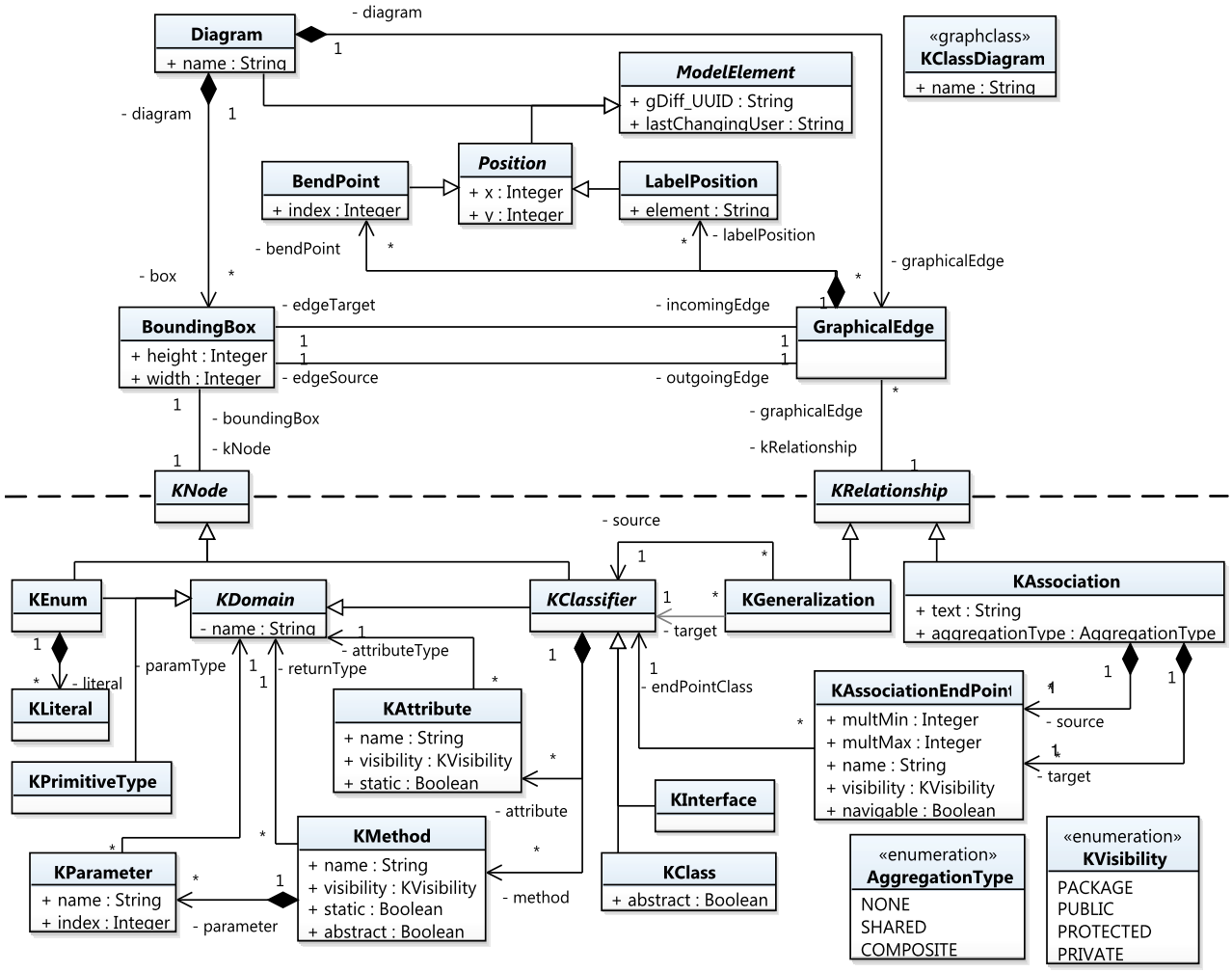

Figure 2: Simplified UML class diagram meta-model

shows how the changes on the copy of *Client A* are represented in modeling deltas by the DOL notation in order to apply these changes to the copy of *Client B*.

According to the `ModelElement` class of the meta-model in Figure 2, each model element has an attribute named `gDiff_UUID` which means all model elements are assigned to *persistent identifiers*. Therefore, each model element in this example also has a persistent identifier e.g. the class `Person` is assigned to `g1` and the class `Teacher` is assigned to `g4`, etc. Since the attributes of these classes are conceptualized as an independent meta-class `KAttribute` in the meta-model and treated as independent objects on the instance models, they are also assigned to the separate identifiers in the graph-like internal structures of the instance models.

Figure 4 illustrates the changes made on the copy of *Client A* that has to be applied to the copy of *Client B*.

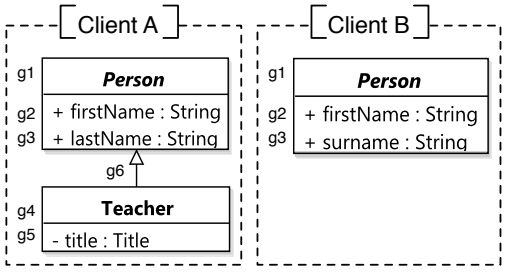

Figure 3: Two copies of the same class diagram

These changes are as follows: the attribute *name* of the class `Person` is changed from `surname` to `lastName` (line 2), the class `Teacher` is created with the attribute `title` (lines 4,5), the generalization `g6` is created connecting the class *Teacher* to `Person` (line 7). This is one way of representing changes on *Client A*. Other possible changes resulting in the same updated version are represented accordingly.

```
1 //-----------------------
2 g3.changeName("lastName");
3 //-----------------------
4 g4=createKClass(PUBLIC,"Teacher");
5 g5=createKAttribute(g4,"title",PROTECTED,
      false);
6 //-----------------------
7 g6=createKGeneralization(g4,g1);
```

Figure 4: Example modeling deltas

As mentioned before, each object of the model can be created or deleted, or the attributes of each object can be changed. The same concept applies to this example and the syntax of the DOL notation is directly derived from the meta-model depicted in Figure 2. For instance, construction of the operation on the line 5 is derived from the meta-class `KAttribute` by providing the create operation. Thereby, an attribute is created with name *title* and the visibility is `PROTECTED`, it is non-static and linked to the class `Person` by the parameter `g5`.

These changes are detected after each subsequent ac-

tions of a designer and synchronized with other copies of the model. Thus, the change list in Figure 4 is separated into three parts and exchanged in three independent modeling deltas. After the model changes are synchronized between these two instances of the same model by applying the delta from Figure 4 to the model given in Figure 3 on the right, the complete resulting models can be seen in Figure 1.

4. MODELING DELTAS IN COLLABORATIVE MODELING

This section depicts an overall architecture of the collaborative modeling application. The main components are explained in detail including the DOL services that are involved in building the collaborative modeling application.

Generally, the collaborative modeling application requests activities such as detecting changes made by users, synchronizing these changes among all other copies and applying them to the other copies of a model. Usually, collaborative applications and tools are built based on client-server architectures. Similarly, Figure 5 depicts the overall architecture of the collaborative modeling application including both client and server sides.

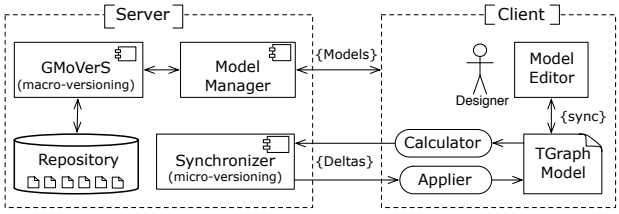

Figure 5: Architecture of Collaborative Modeling Tool

As shown in the figure, both client and server sides are developed as separated applications. Client and server store models according to their appropriate meta-models. It allows to check semantic and syntactical correctness of models while designing models and applying changes to models.

During real-time collaboration, modeling deltas are detected by the change calculator and sent to server. Server then sends modeling deltas to other clients and they are applied other parallel copies of a model by the change applier. The change applier performs based on the *first come first serve* technique. Therefore, possible conflicts are quite rare while applying changes to models because of high speed synchronization. Current experiments do not show any synchronization conflicts of changes.

Each component of the architecture is explained in the follow up subsections in detail.

4.1 Model Representation

On the client side, models are represented by graph-like structures using TGraphs [8]. The graphical modeling editor is realized using Eclipse GEF (Graphical Editing Framework) [15]. There is always bidirectional synchronization between the graphical modeling editor and graph-based model. Once any user makes changes to a model on the editor, these changes are propagated to the graph-based model and vice versa.

On the server side, the repository stores only modeling deltas representing the differences between subsequent model versions. Modeling deltas in the repository are also represented by the DOL notation. Loading models from the repository is done by applying *active delta* (and difference delta if necessary) to an empty model [14].

4.2 Delta Calculator

The state-based *change calculator* is used to detect changes once they are made on a model by collaborators. The inputs for the change calculator are two, changed and unchanged, copies of the same model and the outcome is a modeling delta representing the differences between these copies in the terms of DOL.

The change calculator is implemented using the existing generic difference calculator framework SiDiff [18]. The change calculator employs the *ID-based* model matching algorithm instead of a *change recorder* [13]. The ID-based matching algorithm has shown high speed and performance during real-time collaboration so far. Especially, in the case of the graph-like structured models, the ID-based matching algorithm performs very fast and efficient. Even if the models are represented using other structures like EMF [19] or XMI [2] models, model changes can easily be detected by this component. Since the change calculator approach is realized in generic way, the change calculator is directly applicable to various modeling languages with respect to the meta-model and any special implementation is not required.

4.3 Delta Applier

The *change applier* transforms a model from one state to another according to the change description in a modeling delta. After detecting the user changes, the synchronization component on the server side delivers them to all other clients of the shared model in the form of modeling delta. These changes are applied to other models by the *change applier*. The change applier receives a modeling delta and a model, and applies a given delta to a given model by executing DOL statements in that delta.

Applying changes to graph-based models are realized by *in-place* graph manipulations of JGraLab API [10].

4.4 Synchronizer

The *synchronizer* service on the server side supports exchanging modeling deltas among collaborators. It receives modeling deltas produced by the change calculator and sends them to other collaborators to apply on their copies. Modeling deltas represented by the DOL statements are efficiently exchanged as the string list of serialized Java objects. Since the synchronizer exchanges only real-time deltas, this service is referred to as *micro-versioning*.

4.5 Macro-versioning

Macro-versioning in the *Kotelett* tool is handled by GMoVerS (Generic Model Versioning System) which provides model versioning services based on DOL (cf. [14] for more details). On the server side, GMoVerS is utilized in order to manage models and their histories. The model versioning system also takes advantage of the DOL notation for representing model differences in modeling deltas.

Any version of a specific model can be stored by a user request at any moment during real-time collaboration and any model or/and any version of a specific model can be loaded from the repository.

5. EVALUATION

The DOL approach is applied to collaborative UML class diagram modeling. The collaborative modeling application, *Kotelett* itself was developed by a project group of students in Software Engineering Group at Carl von Ossietzky University of Oldenburg.

The collaborative modeling tool is used in Software Engineering lectures for teaching purposes by a group of students including more than ten collaborators in parallel. The tool is successfully presented by the project group on a study exhibition day for school children. Moreover, the tool is also experimented by the users located in long distance (Germany and Canada). During these experiments, the tool has not shown any inconveniences with agility, amount of users, etc. Hence, the users of the tool have not faced any change conflicts because of rapid synchronization of model changes by small modeling deltas as shown in Figure 1. The DOL-based change synchronization approach is fast enough to exchange all changes before conflicts occur so far.

The meta-model in Section 3.2 supports storage of *layout information* and *modeling language notation*. The collaborative modeling application or other applications of the approach can be extended for other modeling languages by extending or replacing the *modeling language notation* part of the meta-model. The layout information part always remains unchanged. In the case of real-time collaborative modeling application, if the graphical modeling editor part can be redeveloped for further modeling languages, all other background technologies and services such as the change calculator, applier, GMoVerS and DOL notation of the approach remain the same and do not require further implementation.

6. CONCLUSION

This paper expressed the collaborative modeling application of the DOL approach using the DOL statements for exchanging model differences among collaborators on a shared model. Exchanging model changes made by designers is eased by exchanging only modeling deltas which contain only the change operations. Exchanging small deltas allows for rapid synchronization of model changes by reducing the capacity of exchange data and exchange time. Moreover, the real-time collaborative work allows for maintenance of models and carrying out their evolution with lesser occurrence of conflicts between various development lines. Even though the approach do not completely ignore the presence of conflicts and it is planned to develop a conflict resolution service for the collaborative modeling application in future. The collaborative modeling application is planned to be extended for further UML diagrams as future work.

The DOL approach was also applied to the generic model versioning system (GMoVerS) in [14] which is also used in this collaborative modeling application for macro-versioning.

The client side of the tool is available in [16] to download and it can also be demonstrated in the workshop.

7. REFERENCES

[1] AppJet Inc. Etherpad. http://www.etherpad.com, visited on 01.06.2015.

[2] J. Bézivin. From object composition to model transformation with the mda. In *Technology of Object-Oriented Languages, International Conference on*, pages 0350–0350. IEEE Computer Society, 2001.

[3] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. An introduction to model versioning. In *Formal Methods for Model-Driven Engineering*, pages 336–398. Springer, 2012.

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

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

[6] S. Dekeyser and R. Watson. Extending google docs to collaborate on research papers. *University of Southern Queensland, Australia*, 23:2008, 2006.

[7] M. Dirix, A. Muller, and V. Aranega. GenMyModel: An Online UML Case Tool. *Joint Proceedings of Tools, Demos and Posters*, page 14.

[8] J. Ebert, V. Riediger, and A. Winter. Graph Technology in Reverse Engineering, The TGraph Approach. In *10th Workshop Software Reengineering (WSR)*, volume 126, pages 67–81. GI (LNI), 2008.

[9] A. Haber, K. Hölldobler, C. Kolassa, M. Look, B. Rumpe, K. Müller, and I. Schaefer. Engineering Delta Modeling Languages. In *Proceedings of the 17th International Software Product Line Conference*, pages 22–31. ACM, 2013.

[10] S. Kahle. JGraLab: Konzeption. *Entwurf und Implementierung einer Java-Klassenbibliothek für TGraphen*, 2006.

[11] T. Kehrer, M. Rindt, P. Pietsch, and U. Kelter. Generating Edit Operations for Profiled UML Models. In *Model Driven Engineering Languages and Systems (MoDELS 2013)*, pages 30–39, 2013.

[12] M. Koegel and J. Helming. EMFStore: a model repository for EMF models. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 307–308. ACM, 2010.

[13] S. Krusche and B. Bruegge. Model-based real-time synchronization. In *International Workshop on Comparison and Versioning of Software Models (CVSM'14)*, 2014.

[14] D. Kuryazov and A. Winter. Representing model differences by delta operations. In *18th International Enterprise Distributed Object Oriented Computing Conference, Worshops and Demonstrations (EDOCW), IEEE Computer Society Press, 2014*, pages 211–220, Ulm, 3-5 September 2014.

[15] B. Moore. *Eclipse development using the graphical editing framework and the eclipse modeling framework*. IBM, International Technical Support, 2004.

[16] Project Group. Kotelett: Collaborative Modeling Tool. https://pg-kotelett.informatik.uni-oldenburg.de:8443/build/stable/.

[17] J. Rumbauch, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, July 2004.

[18] M. Schmidt and T. Gloetzner. Constructing Difference Tools for Models Using the SiDiff Framework. *ICSE 2008*, pages 947–948, May 10-18 2008.

[19] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.