Representing Model Differences by Delta Operations

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

***Abstract*— Software models are subject to subsequent changes. Models evolve over time to meet changes resulting in several versions. The differences between subsequent model versions are represented in differences documents usually referred to as *Modeling Deltas*. The model differences are used in establishing several further services and operations of version control, collaborative modeling and history analysis tools for software models. Therefore, finding an adequate difference representation approach is essential for applicability, re-usability and analysis of the model differences.**

This paper introduces the *Delta Operation Language (DOL)*, a meta-model independent and operation-based concept to model difference representations. The proposed approach provides several operative *DOL-services* for utilizing the DOL-based differences. The application areas of DOL include *Model Versioning*, *Model History Analysis* and *Collaborative Modeling*. The approach is applied to UML activity diagrams as a running example.

I. Motivation

Software models are widely used in designing software systems on different abstraction levels. Like code of software projects, software models evolve over time undergoing various changes such as extensions, improvements and corrections. Constantly changing a model results in different versions of a model during its lifetime. The consistent recognition, specification and exploitation of the differences between subsequent model versions are crucial in comprehending, analysing and managing model-based software systems.

For dealing with modifications on models, the differences between model versions have to be properly identified and represented in difference documents i.e. *Modeling Deltas*. Representation of the model differences in modeling deltas is an important concern for *version control*, *collaborative modeling* and *history analysis* tools.

Model versioning systems used to handle and store the histories of evolving software models. They usually take advantage of modeling deltas to store the model differences. Each modeling delta represents the differences between subsequent versions of a model. A model versioning system reduces the difference storage space and improves simplicity by storing only deltas.

Changes are applied to software models by a group of designers using collaborative modeling tools. In the case of collaborative modeling, exchanging the model differences is eased by exchanging only modeling deltas which contain only the differences. Exchanging small deltas enables them to synchronize changes rapidly by reducing the capacity of exchange data.

Analysing the model histories helps to understand the evolutionary life-cycle of models. Information about the model histories is gathered by tracking the change histories of each model element. As long as only changed model elements are referred to in modeling deltas, tracing the change history from modeling deltas help to rapidly detect required history information. Afterwards, the users can browse or visualize that information to see how a model evolves.

Software models do not follow the similar principles and syntax as code-based projects. Therefore, the rich data and compound structure of models has be considered in representation of the model differences. Difference representation has to be capable of fully handling and carrying all necessary information about the change histories. Additionally, the differences represented in modeling deltas must be easy to access for further re-use and manipulations. Software models also can be represented as textual files using exchange formats like XMI (XML Metadata Interchange) [1]. But, it is commonly admitted that differentiating textual representation of models does not follow modeling concepts [2], [3]. The code-centric versioning systems like Git [4], Subversion [5] or RCS [6] can not fully handle software models in exchange formats and do not produce useful information about the model differences. To this end, this paper addresses the problem of difference representation for software models.

This paper introduces the general ***Delta Operations Language (DOL)*** of *Generic Model Versioning System (GMoVerS)* to model difference representations. DOL is a set of domain specific languages to model difference representation in terms of delta language operations. In order to derive a specific DOL for a specific modeling language, the *meta-model* of a modeling language is required. A *DOL Generator* used to generate a specific DOL for a certain modeling language using the meta-model. Then, a specific DOL is fully capable of representing all differences between subsequent versions of the instance models in terms of operation-based DOL in modeling deltas. The operations in modeling deltas are referred to as *Delta Operations*.

Moreover, DOL aims at supporting several *DOL-services* which can access and reuse delta operations. These operative services make the DOL-based modeling deltas quite handy in various application areas and enable application areas to utilize the DOL-based modeling deltas. Operation-based DOL has several use cases in this paper. These are *Generic Model Versioning System (GMoVerS)* (discussed in Section VIII-A), *Model History Analysis* (Section VIII-B), *Collaborative Modeling* (Section VIII-C) which is in process.

The paper is structured as follows: Section II lists requirements that difference representation has to satisfy. Several related approaches are discussed in Section III. Section IV gives an example of an evolving model and DOL-based difference

representation between subsequent model versions. Section V explains DOL generation from the given meta-model. The DOL-services are introduced in Section VI. The prototype implementations are explained in Section VII. Insight into application areas of DOL is given in Section VIII. The paper ends up by drawing conclusions and outlook for future work in Section IX.

II. Requirements

In the case of software models, difference representation has to satisfy a number of requirements regarding its re-usability, applicability and optimality. Difference information (i.e. history information of model changes) must be small and complete with a simple syntax and re-usable when needed. Besides, the difference representation approach has to be independent from modeling languages and modeling tools enabling applicability. Consequently, in order to have a solid difference representation technique in the end, several requirements and properties following concrete principles are defined targeting re-usability and applicability of modeling deltas and efficient access of model versions:

- *Meta-model Generic.* There are several modeling languages which follow different formal specifications and concepts. The abstract syntax of modeling languages i.e. the modeling concepts are defined by the languages meta-model. Models conforming that modeling language are subject to continuous changes and evolution. Thus, being meta-model independent makes a difference representation approach applicable to all modeling languages with respect to their meta-models.
- *Tool Independent.* There are several modeling tools and they have own internal model representations. To be able to handle models designed in different modeling tools, difference representation must not rely on an internal model representation of a specific model designing tool restricting itself for that modeling tool.
- *Operation-based.* The model differences are the collection of changes of modeling concepts. Changes of modeling artefacts can be viewed as operations that are applied to a previous version to form the later. These operations have to allow for representing all model changes embodying all necessary information about the change histories. Additionally, using an operation for each separate change results in a low number of operations with simple syntax and allows effortless implementations as well as follows actual modeling concepts. These operations have to be composite to enable representation of complex model changes.
- *Delta-based.* After representing changes in terms of operations, a certain set of operations between two states of a model should be grouped and stored in a modeling delta. A modeling delta consists of a set of operations which refer only to changed modeling concepts, meanwhile that delta performs the executable descriptions of the differences which allows to revert older model versions.
- *Completeness.* The representation must carry precise information about each change including the kind of change and the referenced modeling concept. The kind of change defines what kind of change is made (cf. creation or deletion of an object or change of object attributes), whereas the referenced modeling concept admits the changed model object.
- *Reusability.* The model differences represented in modeling deltas 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 fur further analysis and manipulations enabling applicability to various application areas.

These significant principles are also design foundation of the DOL-based representation approach that contribute to empower the qualification and solidity of difference representation. The approach aims at fulfilling these requirements throughout this paper.

III. Related Work

Before explaining operation-based DOL in detail, this section discusses several existing difference representation approaches. Also, these approaches are reviewed in the context of principles listed in Section II.

An early approach on operation-based difference representation is introduced by Alanen and Porres [7] which is meta-model independent as well. The approach addresses calculation of the differences and union of models. It represents the differences as a sequence of difference operations. Moreover, a number of merge conflicts are extended by the combination of the difference operations. The approach is meta-model generic, operation-based and provides a conflict resolution technique.

A meta-model independent approach for difference representation is introduced by Cicchetti et al. [2], [8]. The approach uses a *differences model* for representing the model differences. The differences model conforms to the differences meta-model derived from the base meta-model by *automatic transformations*. The approach provides a difference application component which requires *model matching* between the differences and base models to detect *weaving model*. The weaving model is used to contain references between the differences and base model elements. *Conflict resolution* is provided to detect and resolve conflicts while merging the differences and base models. Cicchetti et al. is meta-model generic, provides difference application and conflict resolution services, but it uses model-based way of difference representation.

SMOVER (Semantically enhanced Model Version Control System) [9] is a state-based approach, which provides several standard versioning activities such as *add* – to bring new models under version control, *checkout* – to obtain a copy of a model in the central repository into the local working space, *commit* – to commit local changes to the central repository and *update* – to update the local working space with the latest changes in the central repository. SMOVER mostly addresses flexible difference merging technique and uses *standard SQL databases* to store the model differences. Hence, the approach requires to have an adapter which lies between external modeling tools and SMOVER.

Haber et al. [10] addresses engineering delta modeling languages for software product lines. The approach aims at automatically deriving delta operations for software architectures of software product lines. But, it relies on text-based models and requires to specify the grammar of a modeling language to derive a delta language. The approach is operation-based, but focuses on software product lines.

An approach for specifying and recognizing model changes based on edit operations (MOCA) is introduced by Kehrer et al. [11], [12]. The approach consists of a chain of model differencing steps including model matching, deriving low-level changes and semantic lifting phases for EMF (Eclipse Modeling Framework) [3] models. The detected low-level changes are grouped and shifted up to a high-level syntax using the "semantic lifting" component. The authors define the set of recognition rules to recognize low-level changes and produce the user-level changes. These set of recognition rules cover 41 different edit operations for UML class diagrams [13] and 16 different edit operations for Matlab/Simulink models [14]. The resulting user/high-level changes are represented in models with different colours. MOCA is operation-based, uses model-based way for representations and focuses on EMF models.

Koegel and Helming [15] developed the EMF Store framework which is a data model repository for EMF models [3]. The framework allows collaborative work of several modellers directly via peer-to-peer connection providing semantic versioning of models, branching and conflict detection while merging. The EMF Store platform is extended by Krusche and Bruegge with Model-based Real-time Synchronization [16]. The EMF Store uses operation-based change tracking considering an atomic change of the instance models or a composite of several atomic changes. EMF Store is operation-based, re-usable, delta-based, provides both state-based and runtime operation recording but addresses EMF models.

Another model-based difference representation approach, EMF compare and merge [3] was introduced for differencing EMF models. The EMF compare provides model comparison and merging services and relies on EMF models.

These approaches already provide a support to deal with some aspects and principles of model difference representation. All the existing approaches provide a form of difference representation even some of them are strict to a specific modeling language, but most of them lack providing sufficient catalogue of additional services to manipulate their difference information. GMoVerS DOL aims at supplying a wide range of the DOL-services which can directly access and reuse the DOL-based repositories. Besides, the approach in this paper satisfies all the principles in Section II which are literally important for applicability and exploitation of model difference representations.

IV. Example

This section covers a simple example of model difference representation by the GMoVerS DOL. To express the idea behind the approach, a simplified UML activity diagram [13] is used as running example throughout this paper.

Figure 1 portrays a simplified meta-model of UML activity diagram. The current prototype implementation of the DOL Generator imports meta-models designed as UML Class diagrams. DOL is derived for this specific meta-model to explain the example in this section, but DOL Generation is completely independent from meta-models satisfying the *Meta-model Genericness* principle.

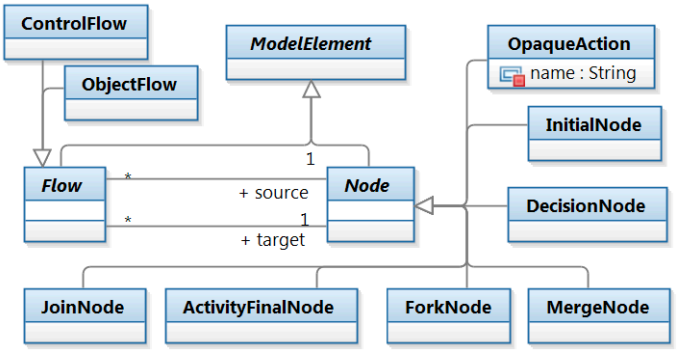

Figure 1: Simplified UML Activity Diagram meta-model

All meta-model classes are either *Nodes* or *Flows* (each flow having source and target attributes) and both are of type model *ModelElement*. Only *Action* has the *name* attribute whereas the other classes have no attributes.

Figure 2 depicts three consecutive versions of the same UML activity model performing an *Ordering System* example. All model versions conform to the same simplified meta-model shown in Figure 1.

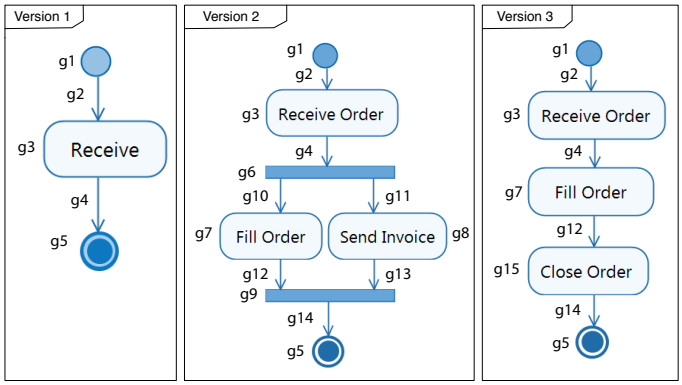

Figure 2: Example activity diagram in three concurrent versions

Each model element is assigned to a certain *persistent identifier* (g_x). The first version contains an *Action* named *Receive* as well as an *Initial* and a *Final* node. All elements are connected by *(Control) Flows*. The model evolves into the second version after several modifications such as *Fork, Join nodes*, two *Actions* named *Fill Order* and *Send Invoice* are created, the target end of *Control Flow* `g4` is reconnected to the fork node, the name of the *Receive* action `g3` is changed, and several control flows are created connecting these nodes. Finally, the model evolves again into the third version after creating a new action with name *Close Order*, reconnecting the target ends of `g4` and `g12`, and the source end of `g14` and deleting *Fork* and *Join*-nodes, the *Send Invoice* action, and the control flows `g10`, `g11` and `g13`.

To represent these differences, the specific DOL for UML activity diagrams is derived from its meta-model in Figure

1 by applying three atomic operations `create`, `delete` and `change` to each concept of the meta-model (deriving a complete set of DOL operations is explained in Section V). A delta-operation may create or delete an object/model element or change its attributes and associations. These three operations are accepted as sufficient operations to apply to the modeling concepts while deriving any specific DOL and the specific DOL is capable of representing all differences by these three operations. Other change operations like *moving* a group of elements from one place to another in a model can be achieved by changing one (or several) association(s) between a part that should be moved and the rest of a model.

Like classical version management systems for source code (cf. Subversion [5], Git [4], RCS [6]), GMoVerS DOL follows a *backward delta* approach, where the most current version and several differences documents are stored directly. The current version (the third version in this example) is also represented by operations which consists only of creation operations. Figure 3 depicts the modeling delta named *active delta* which gives directly the current model version by only create operations.

The resulting modeling deltas are directed in reverse order i.e. each change leads from the later state to the previous states. Because all application areas of difference representation usually demand the reversal operations in order to obtain earlier versions from later versions. Moreover, producing delta operations in reverse order is quite practical for implementation of applications.

```
1 g1 = createInitialNode();
2 g3 = createOpaqueAction("Receive Order");
3 g7 = createOpaqueAction("Fill Order");
4 g15 = createOpaqueAction("Close Order");
5 g5 = createActivityFinalNode();
6 g2 = createControlFlow(g1,g3);
7 g4 = createControlFlow(g3,g7);
8 g12 = createControlFlow(g7,g15);
9 g14 = createControlFlow(g15,g5);
```

Figure 3: Active delta

Each delta operation contains a ***Do part*** (cf. *g1=createInitialNode();*) which describes the *kind of change* by means of *operations* (one of create, change, delete, explained in Section V) and an ***Object part*** (with attributes if required) (cf. *g1=createInitialNode();*) which refers to the modeling concept. To refer to model elements from the operations in the modeling deltas, *unique persistent identifiers* are used as *references*. For instance, the operation on the sixth line in Figure 3 creates a *Control Flow* connecting `g1` and `g3` and assigns it to `g2`.

```
1  g6 = createForkNode();
2  g7 = createOpaqueAction("Send Invoice");
3  g9 = createJoinNode();
4  g4.changeTarget(g6);
5  g12.changeTarget(g9);
6  g14.changeSource(g9);
7  g10 = createControlFlow(g6,g7);
8  g11 = createControlFlow(g6,g8);
9  g13 = createControlFlow(g8,g9);
10 g15.delete();
```

Figure 4: Delta between active and the second versions

Likewise, the differences deltas are also represented using operation-based difference representation. The differences delta in Figure 4 depicts the differences between the third and second versions.

Finally, Figure 5 illustrates the delta including the differences between the second and first versions.

```
1 g3.changeName("Receive");
2 g4.changeTarget(g5);
3 g6.delete();
4 g7.delete();
5 g8.delete();
6 g9.delete();
```

```
7  g10.delete();
8  g11.delete();
9  g12.delete();
10 g13.delete();
11 g14.delete();
```

Figure 5: Delta between the second and first versions

The modeling deltas in these figures are executable descriptions of the differences. Each of the differences deltas allows to revert the previous model version from the latest version. The modeling delta in Figure 4 reverts the model to the second version from the third and the modeling delta in Figure 5 reverts the model to the first version from the second. The concatenation of the modeling deltas from Figure 4 and 5 leads to the differences to be applied to the current version resulting in the first version.

The model elements that have to be changed are addressed by the persistent identifiers in the delta operations. For example, the current model version contains an *Action* `g3` named *Receive Order*. It is not referred in the delta in Figure 4 because it is not changed. In the delta in Figure 5, the name of that action is changed to the previous name *"Receive"* (cf. Figure 5, line 1).

In Figures 4 and 5, the modeling deltas consist only of the list of changes including all necessary information about the differences fulfilling *completeness* and the unchanged model elements are not referred satisfying *Delta-based* properties.

V. Delta Operation Language

GMoVerS – Delta Operations Language (DOL) is a family of operation-based languages to represent the model differences. This section expresses generating a specific DOL from the meta-model of a specific modeling language by applying three basic operations to each concept of the given meta-model. After all, the resulting specific DOL allows to produce all the possible representation operations that needed to represent the entire model history in modeling deltas.

Figure 6 depicts the specification of the meta-model in Figure 1 with the specific DOL-operations. It is abstraction of the DOL interface in Figure 7.

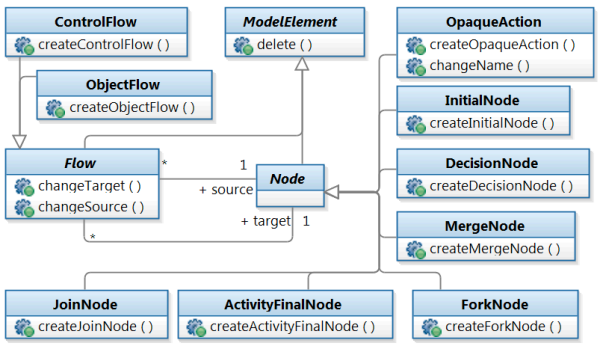

Figure 6: Meta-model of the specific DOL-operations

In Figure 6, all model elements including flows and nodes can be deleted, also, they can be created with parameters if they have attributes. Only attributes can be changed and all associations are associated with attributes. Attributes are created when their composing elements are created and deleted when their composing elements are deleted.

The specific DOL is generated as Java Interface by a *DOL Generator* (Figure 8) importing the meta-model of a modeling language. The *methods* of the resulting interface are named and parametrized according to the meta-model concepts including one of the *create*, *delete* and *change* operations (Figure 7). Each implementation of these methods results in an analogous operation with relevant parameters.

Considering the activity diagram example presented in Section IV, the DOL Generator imports the simplified meta-model depicted in Figure 1.

```
1  //------ Creations ------
2  InitialNode createInitialNode();
3  OpaqueActionNode createOpaqueAction(String name);
4  ForkNode createForkNode();
5  JoinNode createJoinNode();
6  DecisionNode createDecisionNode();
7  MergeNode createMergeNode();
8  ActivityFinalNode createActivityFinalNode();
9  ControlFlow createControlFlow(Node source, Node
       Target);
10 ObjectFlow createObjectFlow(Node source, Node
       Target);
11 //------ Changes ------
12 void changeName(String newName);
13 void changeSource(Node newSource);
14 void changeTarget(Node newTarget);
15 //------ Deletion ------
16 void delete();
```

Figure 7: Interface generated from UML Activity Diagram meta-model

In case of the meta-model of any other modeling language, the DOL Generator follows the same principle to generate DOL i.e. only a meta-model is imported and three basic operations are applied to each concept of that meta-model. Eventually, each of the interface methods has the same structure as a *Do part* and an *Object part*. The delta operations are produced by implementations of that interface by assigning persistent identifiers.

VI. DOL Services

Representing the model differences by modeling deltas is not only focus of GMoVerS DOL. Besides, this paper introduces a reasonable set of *DOL-services* enabling reusability and applicability of the DOL-based deltas. Figure 8 displays the overall architecture of the DOL-services and the potential orchestration of them based on data-flows among these services.

The amalgamation depicts the DOL-services such as an ***Adapter***, a Difference ***Calculator***, a Delta ***Optimizer***, a ***Patcher*** and a ***Change Tracer***. Each DOL-service has a particular task and is involved in constructing the specific orchestration in the framework of DOL applications. For instance, models designed in external modeling tools are imported into the system through the *Adapter*. The adapter parses models in the exchange formats to internal representations and vice versa. The difference *calculator* is used to detect the differences between the compared models and produce the differences and active deltas by implementing a DOL interface. The difference calculator uses the *Optimizer* to produce the optimized modeling deltas and to write them into a repository. After all, other DOL-services such as the *Patcher* and the *Change Tracer* are capable of utilizing the DOL operations. The patcher is used to revert older model versions. The change Tracer tracks a specified model element and reports the change history of that element. Then, these change reports can be visualized to analyse the change history. All of these DOL-services are discussed in detail in the following sections.

These DOL-services are employed in developing the DOL applications such as *Model Versioning*, *Collaborative Modeling* and *Model History Analysis* (discussed in Section VIII). Architectures of these applications are built based on the specific orchestrations of several DOL-services in a certain order.

A. Adapter

Software models are created by using model designing tools and mostly these tools do not provide proper model versioning, collaborative modeling or model history analysis support. Most commercial modeling tools such as Visual Paradigm (VP) [17], Rational Software Architect (RSA) [18] or open source tool Eclipse Modeling Framework (EMF) [3] have their own internal model representation techniques. Thus, integrating version management or collaborative modeling tools with model designing tools is a challenge. But, these tools provide export and import of software models by XML Metadata Interchange (XMI) [1] format without layout information. Therefore, to exchange models between the external modeling tools and the GMoVerS version control system, the DOL-services provide the *Adapter* (Figure 9).

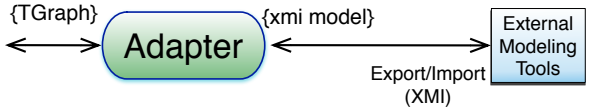

Figure 9: DOL-Service: Adapter

In GMoVerS, models are represented as TGraphs internally [19] (discussed in Section VII) to make them generally processable in implementation of the DOL applications. The *Adapter* can parse models in both directions: from the exchange formats to TGraphs and vice versa. The adapter also uses the meta-model of a modeling language to recognize all modeling concepts specified in the exchange format. Therefore, it is independent from model designing tools, modeling languages and exchange format versions in regard to a meta-model.

B. Difference Calculator

The difference calculator detects the differences between compared models using state-based comparison and produces the delta operations. A number of approaches already exist for difference calculation. Therefore, Küpker [20] investigated existing approaches such as UMLDiff [21] and SiDiff [22], [23] and combined them to *gDiff* (generic differentiating) tool.

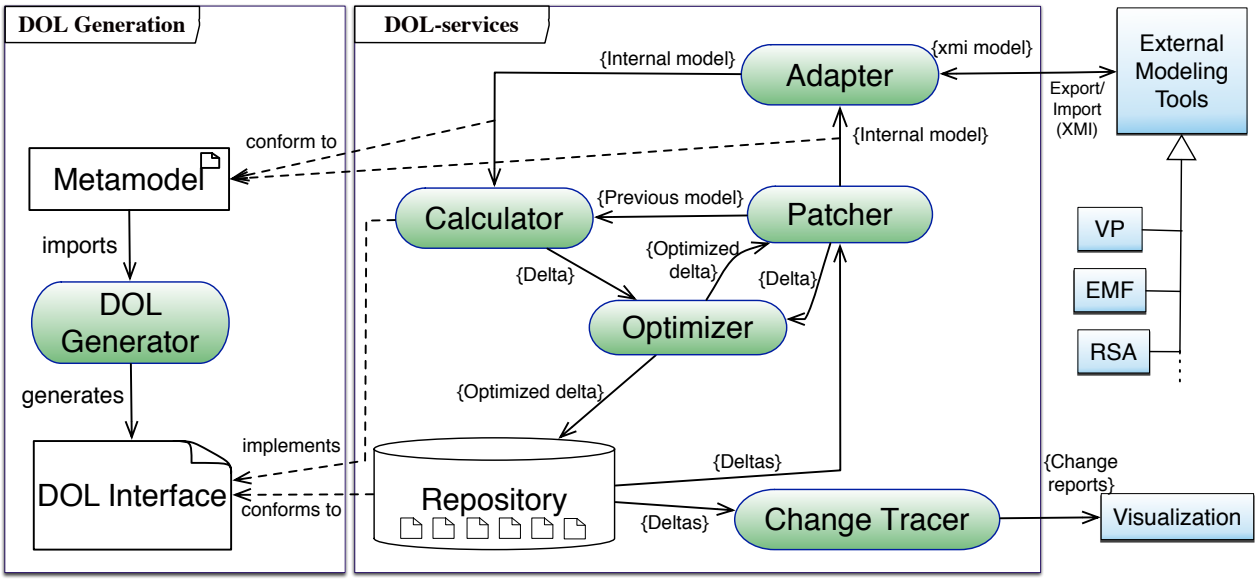

Figure 8: DOL generation and potential orchestration of the DOL-services

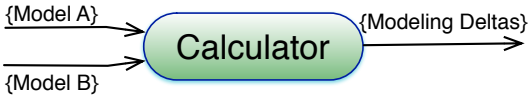

Figure 10: DOL-Service: Difference calculator

The calculator used in GMoVerS firstly detects possible matches between two *compared models* (Model A and Model B in Figure 10), and then calculates the model similarities using similarity metrics such as a name [21] or a structural similarity [23]. The candidate element with the highest similarity is selected as an unique match for the host element and created, deleted and changed elements are detected.

After detecting such modifications, the calculator produces two modeling deltas in the operation-based format by implementing a specific DOL Interface: the *active delta* representing the current model by create operations (cf. Figure 3) and the *differences delta* which leads the older version from the latest. If the first version of a model is given, the calculator produces only the active delta without the difference delta. Because, there is not previous versions to compare and detect the difference delta. Assigning the persistent identifiers to delta operations is also done by the difference calculator while matching model elements. These assignments are persistent over all sequences of the modeling deltas.

C. Delta Optimizer

In some cases, modeling deltas might contain some useless operations. To increase efficiency by reducing useless operations in modeling deltas, optimization of the delta operations is required. Optimization helps to receive more optimal modeling deltas, eventually. The *Delta Optimizer* (Figure 11) basically receives modeling delta as input.

There might be a lot of useless operations or redundancies in modeling deltas. For instance, if a particular model element is created and later deleted in the same delta (this might happen in the case of collaborative modeling at runtime), these two operations are registered for one element where both have no affect.

Figure 11: DOL-Service: Delta Optimizer

Another example might be changing one element 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 model element.

Moreover, the order of delta operations in modeling deltas is also important to avoid the lost and fuzziness of change information while applying deltas to models by the patcher. The creation operations are lifted to the top and deletions are dropped to the end and changes are placed in the middle in each modeling delta. Creations are made firstly, then changes are made and deletions are made in the end. For instance, if a control flow has to be change to a newly created action node in the case of Activity diagrams, a new action node has to be created first, then an existing control flow can be reconnected to that new node.

After completing optimization process based on aforementioned criterion, the delta optimizer produces optimal modeling delta in the end. Besides, the delta optimizer can be utilized by other DOL-services to optimize the DOL-based modeling deltas. For example, the *Patcher* uses delta optimization to ease delta application process so that it can skip some delta operations between subsequent modeling deltas.

D. Patcher

The *patcher* allows to revert to earlier versions by *applying* (sequences of) modeling deltas to a model. For instance, modellers may need to roll back a model to an older version/state in the case of loss or damage of information. The inputs to the patcher are a *model* and a *modeling delta*. An input delta is applied to an input model and a result is previous model version (Figure 12).

Figure 12: DOL-Service: Patcher

The delta operations are represented as executable descriptions in modeling deltas. For instance, execution of the active delta in the example in Section IV results in the current state of the model ($Version_3 = \emptyset.apply(\Delta_{active})$). Executing each differences delta leads to the previous state from the current ($Version_2 = Version_3.apply(\Delta_{(3,2)})$ and $Version_1 = Version_2.apply(\Delta_{(2,1)})$. To apply differences deltas to the current model version, formerly the current model version has to be reverted from the active delta.

Considering that there are fifty versions of a model and the patcher is requested to revert the tenth version from the fiftieth. In this case, the patcher runs through a sequence of deltas between the fiftieth and the tenth versions and optimizes patching process by concatenating and optimizing the delta operations in these deltas. Several operations in these steps can be skipped using the optimizer to have faster and efficient reverting in the end.

E. Change Tracer

Being aware of changes on models is essential in comprehending and analysing the histories of evolving models. To this end, GMoVerS DOL provides the *Change Tracer* service which contributes the history analysis to reveal necessary information about the change history. The change tracer copes with extracting necessary information about whole evolutionary life-cycle of a model by focusing on a specific model element.

The change tracer receives the set of modeling deltas from the repository as input (as shown in Figure 13). Then, it seeks change information of a specific model element based on its persistent identifier by verifying the given set of modeling deltas. After all, detected information is reported as a set of connected correspondences so that it can be used in further analysis by visualizing.

Figure 13: DOL-Service: Change Tracer

For instance, Figure 14 illustrates all history information of the control flow `g4` in the example in Section IV.

```
1 g4 = createControlFlow(g3,g7);
2 g4.changeTarget(g6);
3 g4.changeTarget(g5);
```

Figure 14: History information of Control Flow `g4`.

The traced element was referred to in three states. Because, it was in three different states in three model versions i.e it was changed subsequently. Information about these states is tracked by slicing three modeling deltas, the *active delta* in Figure 3 and two differences deltas in Figures 4 and 5.

VII. Implementation

This section discusses the prototype implementation of the DOL-services integrated into the Eclipse environment. First of all, this section indicates which tools and techniques are required to derive a specific DOL for a specific modeling language and to implement all the DOL-services. Then, the selected tools and techniques are explained based on their role in the prototype implementations.

All model designing tools have their own internal model representation technique. Therefore, *internal representations* are required for both instance and meta-models. This allows the DOL-services to manipulate models being a part of the specific orchestration of any DOL-application.

To manipulate models within the DOL-application platforms, *in-place* model transformations are needed. Model manipulations help to apply changes to models and shift a model between states which it undergoes during the evolution process. The DOL-services like the *Patcher* and the *Adapter* take advantage of *in-place* model manipulations to fulfil their missions. The *Difference Calculator* requests the model *traversal* system to compare the given models and calculate the differences between them.

The requirements for the implementation tools and techniques are solved by JGraLab (Java Graph Laboratory) API [24]. First of all, JGraLab provides means to handle meta-models defining *Graph Schema*. Thereafter, TGraphs conforming the given graph-schema can be manipulated and handled by JGraLab API. To represent models internally, GMoVerS DOL uses TGraphs [19] that are accessible by JGraLab API. Vertices and edges are the first-class objects in TGraphs and they can be attributed, are directed and typed. JGraLab supports management of TGraphs, and *in-place* graph transformations such as creation, deletion and change of graph elements. Finally, JGraLab has its own graph querying system which allows to filter, traverse and to compare the query results.

The meta-model of a modeling language is designed in RSA [18] as UML class diagrams and loaded into JGraLab. When a meta-model is loaded, the DOL Generator automatically generates the DOL Interface. Likewise, the meta-model is used by the adapter to parse models in the exchange XMI format to TGraphs. The instance models are designed in RSA as well and parsed to TGraphs by the *adapter*. Both, meta- and instance models are exported in the exchange formats without layout information. Eventually, all DOL-services like the calculator and the patcher directly operate on the TGraph-based models using JGraLab's *in-place* transformations and *querying* system.

The DOL-based difference representation approach is also implemented by VIATRA (VIsual Automated model TRAnsformations) in [25]. GMoVerS DOL is not strict only to JGraLab and can be realized by other model transformation and representation approaches such as QVT (Query/View/Transformation) [26]. But in some cases, model transformation approaches aim at transform from source model to target model lacking in-place manipulations which also makes implementations more complicated. For example, some experiments showed that deleting a model element is solved by an unspecified transformation rule in the case of ATL (ATLAS

Transformation Language) [27] i.e. if a model element has to be deleted during transformation, a rule for that deletion is not specified so that an element supposed be deleted is not copied into a new model.

VIII. Applications

In order to present applicability of the approach, several use cases are explained in this paper. These use cases are *model versioning*, *model history analysis* and *collaborative modeling* that are discussed in follow up sections in detail. The architectures of these applications are developed by the specific orchestrations of the DOL-services explained in Section VI and built on the underlying principles and properties listed in Section II.

A. Model Versioning

Model versioning aims at managing and manipulating models as well as storing and reusing the model differences. Therefore, version management systems are important to handle models and their histories. To this end, the *Generic Model Versioning System (GMoVerS)* is established adopting a specific orchestration of the DOL-services in this section.

The DOL-based difference representation approach is applied to versioning Sustainability Reports at companies in [28]. As sustainability reports at companies are also subject to constant changes and evolution, companies intend to report and analyse sustainability information to provide sustainable future. Thus, reports have to be versioned to analyse histories of sustainability reports and to present changes. To derive a specific DOL for representing differences of report versions, the approach employed a schema of sustainability reports.

The approach is applied to UML activity diagrams in this paper. In GMoVerS, the model differences are represented in terms of the DOL operations and the construction of version control activities is achieved by the amalgamation of the DOL-services as displayed in Figure 15.

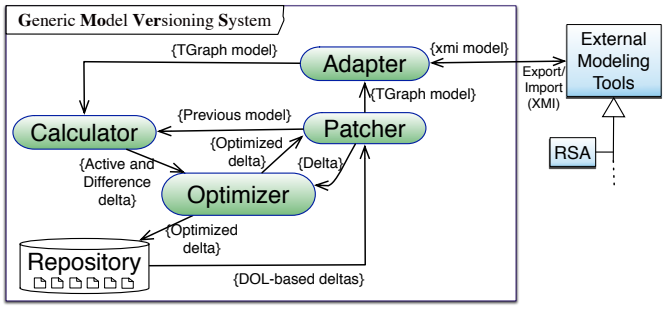

Figure 15: Specific orchestration of the DOL-services for GMoVerS

Current implementations of GMoVerS support *adding* models to the versioning system, *committing* changes and *reverting* older versions. In each of these services, relevant DOL-services are involved in a certain order based on data-flow.

For example, to start version control with a new model, a model has to be ***added*** under version control. It requires involvement of the *adapter* to parse a model in the exchange format to internal representation, the *calculator* to produce the *active delta* for the new model. Similarly, ***committing*** changes requires the *adapter* to parse new versions to internal representation, the *patcher* to revert the previous model version, again the *calculator* to calculate the differences between the previous and committed versions and produce the differences and active deltas. ***Reverting*** needs the *patcher* firstly to revert the recent model version, then it is involved as many times as needed to revert a requested version. For instance, to revert the first version form the third version, the patcher is called three time: one time for reverting the recent model from the active delta, two times to apply two the differences deltas to the current version.

Figure 16 displays a screen-shot of the GMoVerS development environment which is elaborated applying the DOL-services.

The package explorer on the left side shows the arrangement of the workspace including the meta-model in Figure 1, the models and modeling deltas of the example in Section IV. Particularly, all DOL-based modeling deltas belong to the GMoVerS repository. On the upper row of the right side, Figure 16 displays three subsequent versions of the example model in Figure 2 in the exchange format. The central row of the right shows two differences deltas and the active delta in Figures 3, 4 and 5. Finally, the most bottom of the screen-shot is a window to type and execute aforementioned activities such as add, commit and patch.

While committing local model changes to 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 the case of semi-automatic way, version control system has to provide interactive conflict resolution feature by browsing conflicts. To this end, implementation of *conflict resolver* DOL-services is an ongoing work.

B. Model History Analysis

Analyzing the model histories is the best aid in comprehending and understanding what changes are made by designers or to know how a model evolves. Also, observing the model history and its evolution process assists the users in making important decisions about next life of model-based software projects.

The history analysis is built on the top of the change tracer (Figure 17). In order to detect history information of a specific model element, the change tracer fetches a set of modeling deltas from the repository and runs throughout these deltas based on a persistent identifier by gathering required information from each delta. After detecting necessary history information, it is visualized in a tabular view.

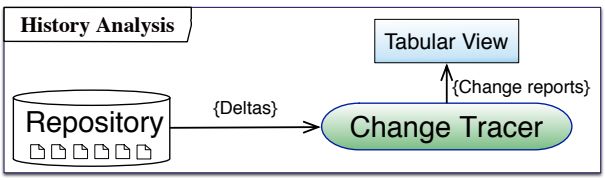

Figure 17: Specific orchestration of the DOL-services for history analysis

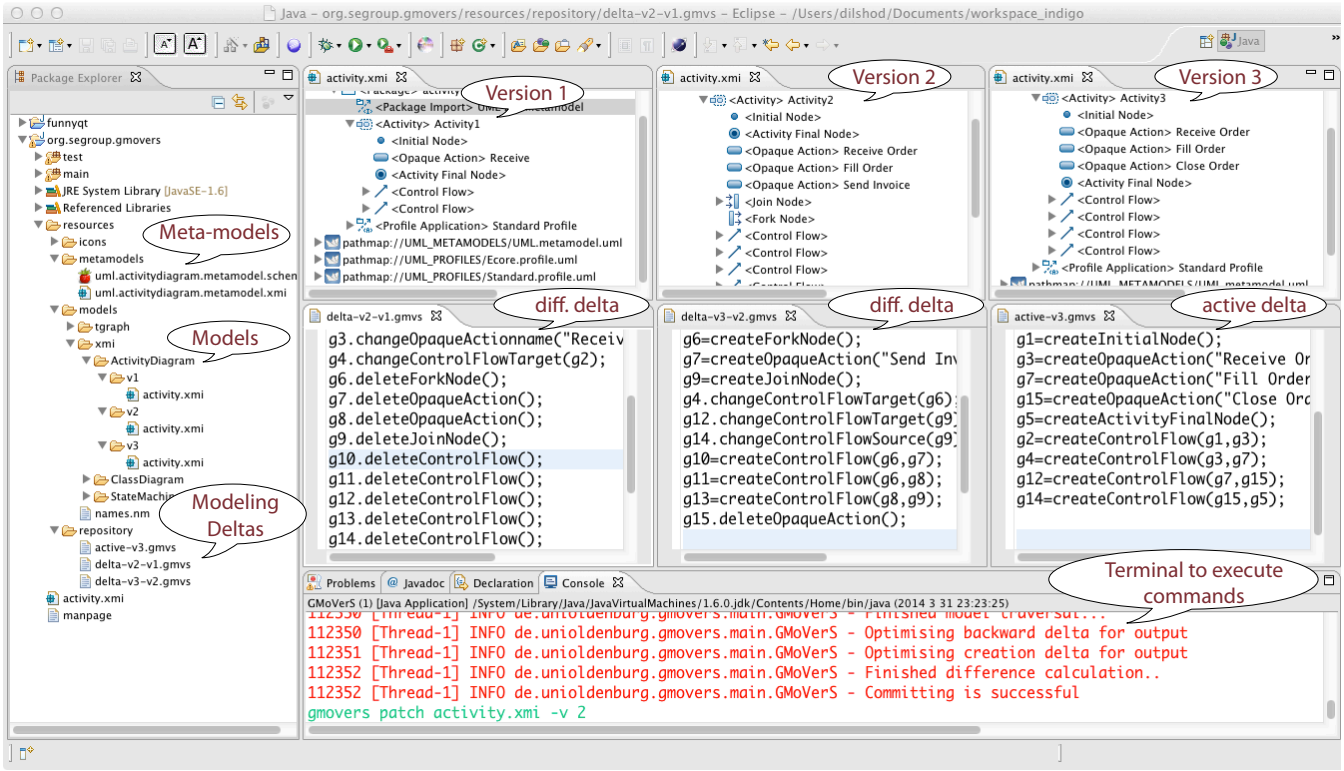

Figure 16: A screen-shot of GMoVerS

The screen-shot in Figure 18 displays the example model in Section IV. The user interface of the model history analyser has two sections: The *Model Tree* section shows model versions in a tree view including their history information and the *tabular view* on the right side lists the history information for a specific model element. To see the history information of any model element, it has to be selected from the model tree and click the *Show History* button. On the same line with the show history button, the label shows the name of a model element which is being traced. The history table contains three columns: *Versions* – the model version number which the change is made, *Modification Type* – the kind of the change and *Date* – the date which the change is made.

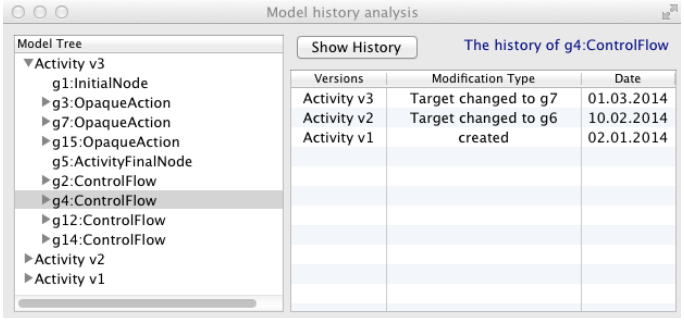

Figure 18: A screen-shot of the history analysis application

The history analysis application firstly builds a tree for each model version running throughout all modeling deltas to show models as a general tree. For example, on the left side of Figure 18, all model versions are outlined in a tree view including all model elements with their attributes. Then, the *change tracer* service traces the history of selected model element based on its persisted identifier. The table on the right side shows the history of the *g4:Control Flow* which is selected from the left tree. The change tracer queries states of the selected element in each version and lists the detailed history of that element including the kind of change and associated objects.

C. Collaborative Modeling

Another prominent application of DOL is collaborative modeling which facilitates a teamwork of several designers on a shared model at runtime. Currently, the implementation of this application is planned as a students project. It is intended to deliver a collaborative modeling environment which is built on top of the DOL-based difference representation approach.

The implementation of the application will rely on representing changes (made by collaborators) in terms of the DOL-operations and exchanging these changes by the appropriate modeling deltas. To this end, it is planned to develop extra DOL-service named *runtime operation recorder* to detect operations embodying changes. The operation recorder will be integrated into end-user model designing tools and it detects changes made by modellers. While manipulating a model, the DOL-based change operations are recorded in modeling deltas. Since a model is manipulated by a group of modellers at runtime, an evolving model has several development branches. In this case, *synchronization* is required among various development branches. Synchronization of changes basically is synchronization of the DOL-based modeling deltas which stored in various workspaces. Synchronization of changes will be eased by exchanging small DOL-based deltas. Eventually, various model designing tools which are running on different platforms can communicate with each other in terms of delta operations language.

IX. Conclusion and Future Work

This paper introduced the operation-based approach to model difference representation facilitating several additional services. DOL and its services satisfy all the requirements mentioned in Section II:

- *Meta-model Generic.* DOL is applicable to several modeling languages with respect to their meta-models. The DOL approach is applied to UML activity diagram in this paper, to versioning Sustainability Reports in [28] and the prototype experiments cover UML state machines.
- *Tool Independent.* The approach has its own internal model representations. But, the DOL-services provide an *Adapter* for integration with external modeling tools by the exchange format.
- *Operation-based.* A specific DOL is generated by the DOL generator from the meta-model of a modeling language. The model differences are represented in terms of the operation-based DOL embodying changes between subsequent model versions. Each DOL operation encloses all necessary information about a change, has a meaningful syntax and completely follows compound modeling concepts.
- *Delta-based.* Only changed model objects are referred in modeling deltas. Also, modeling deltas are the executable descriptions of the model differences which allow to form older versions of a model. These descriptions are implemented by *in-place* model transformations.
- *Completeness.* The DOL-based representation carries precise information about each change including the kind of change and the referenced modeling concept. The changed model objects are referred to according to their persistent identifiers stored in modeling deltas.
- *Reusability.* DOL provides several DOL-services which can directly access and manage the DOL-based modeling deltas. The DOL-services make modeling deltas straightforward and accessible fur further analysis and manipulations enabling applicability to various application areas.

A specific DOL is derived for a specific modeling language and several DOL-services are provided which can manage DOL-based deltas. Services relying on representation of the GMoVerS DOL approach are applied to *model versioning* and *model history analysis* so far, using *state-based* difference calculation. The DOL-services list will be extended with *a runtime operation recorder*, a difference merger and a *synchronizer* for collaborative modeling.

References

[1] Object Management Group, "XMI Specification, v1. 2."

[2] A. Cicchetti, "Difference Representation and Conflict." Ph.D. dissertation, University of L'Aquila, (Italy), April 2008.

[3] EMF: Eclipse Modeling Framework (Compare), http://www.eclipse.org/emf/compare.

[4] J. Loeliger, *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development.* O'Reilly Media, 2009.

[5] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion.* O'Reilly Media, 2004.

[6] W. F. Tichy, *RCS — A System for Version Control. Software – Practice Experience.* Volume 15, Issue 7, 1985.

[7] M. Alanen and I. Porres, "Difference and union of models." in *UML 2003. LNCS.* Springer, 2003, pp. 2–17.

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

[9] K. Altmanninger, A. Bergmayr, W. Schwinger, and G. Kotsis, "Semantically enhanced conflict detection between model versions in SMoVer by example." in *Proceedings of the Int. Workshop on Semantic-Based Software Development at OOPSLA*, 2007.

[10] 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*. ACM, 2013, pp. 22–31.

[11] T. Kehrer, U. Kelter, and G. Taentzer, "A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning." in *ASE*, 2011, pp. 163–172.

[12] T. Kehrer, M. Rindt, P. Pietsch, and U. Kelter, "Generating Edit Operations for Profiled UML Models." in *ME@MoDELS*, 2013, pp. 30–39.

[13] UML: Unified Modeling Language, http://www.uml.org.

[14] Mathworks: Matlab/Simulink, http://mathworks.com/simulink.

[15] J. Helming and M. Koegel, "EMFStore." 2013, http://eclipse.org/emfstore.

[16] S. Krusche and B. Bruegge, "Model-based Real-time Synchronization." in *International Workshop on Comparison and Versioning of Software Models (CVSM'14)*, February 2014.

[17] Visual Paradigm, "Visual Paradigm for UML." *UML tool for software application development*, 2010.

[18] IBM Rational Software Architect, http://www.ibm.com.

[19] J. Ebert, V. Riediger, and A. Winter, "Graph Technology in Reverse Engineering, The TGraph Approach." in *10th Workshop Software Reengineering (WSR 2008)*, R. Gimnich, U. Kaiser, J. Quante, and A. Winter, Eds., vol. 126. GI (LNI), 2008, pp. 67–81.

[20] C. Küpker, "General Model Difference Calculation." Bachelor Thesis, Carl von Ossietzky University of Oldenburg, June 2013.

[21] Z. Xing and E. Stroulia, "UMLDiff: An Algorithm for Object-Oriented Design Differencing." ser. 6. ACM, 2005, pp. 54–65.

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

[23] C. Treude, S. Berlik, S. Wenzel, and U. Kelter, "Difference Computation of Large Models." in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference*. ACM Press, 2007, pp. 295–304.

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

[25] D. Varró and A. Balogh, "The Model Transformation Language of the VIATRA2 Framework." *Sci. Comput. Program.*, vol. 68, no. 3, pp. 187–207, October 2007. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2007.05.004

[26] Object Management Group., "Meta object facility (mof) 2.0 query/view/transformation (QVT) specification." *Final Adopted Specification (November 2005)*, 2008.

[27] J. Frédéric, A. Freddy, B. Jean, K. Ivan, and V. Patrick, "ATL: a QVT-like transformation language." in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 719–720.

[28] D. Kuryazov, A. Solsbach, and A. Winter, "Versioning Sustainability Reports." in *5.BUIS-Tage: IT-gestütztes Ressourcen- und Energiemanagement*. Springer-Verlag, 2013, pp. 409–419.