FINE-GRANULAR MODEL MERGE SOLUTION FOR MODEL-BASED VERSION CONTROL SYSTEM

Waqar Mehmood[1], Nadir Shah[2], Majid Jamal Khan[3], Mukhriar Memon[4], M Ikramullah[5]

[1, 2]Department of Computer Science,
[3]Department of Management Science,
[1, 2, 3]COMSATS Institute of Information Technology, Wah Campus.
[4]Information Technology Center, Sindh Agriculture University, Tandojam.
[5]Department of Computer Science and IT, University of Sargodha.

Email: drwaqar@ciitwah.edu.pk[1], nadir@ciitwah.edu.pk[2], m4majid@gmail.com[3], mukhtiar.memon@sau.edu.pk[4], drikramullah@uos.edu.pk[5]

ABSTRACT

Software Configuration Management (SCM) aims to provide a controlling mechanism for the evolution of software artifacts created during software development process. Controlling software artifacts evolution requires many activities to be carried out such as, construction and creation of versions, computation of mappings and differences between versions, merging (i.e. combining of two or more versions) and so on. Traditional SCM systems are file-based SCM systems, which are not adequate for performing software configuration management activities. File-based SCM systems consider software artifacts as a set of text files, while today's software development is model-driven and models are the main artifacts produced in the early phases of software development process. New challenges of model mappings, differencing, merging, and conflict detection arise when applying file-based solution to model-driven software. The goal of this paper is to develop a configuration management solution for model merging and conflict resolution that overcomes the challenges faced by traditional SCM systems for model-based development. We represent models at fine-grained level as graph structures, which is an intermediate representation based on graph theory. Our approach follows a 3-way model merge process, where a base and its derived versions are used for comparison. To differentiate between conflicted and non-conflicted cases, we have defined different merge cases, and established a merge policy based on merge cases. Merge cases are used along with the comparison result in order to perform conflict resolution and merge operation. We performed a controlled experiment using open source eclipse modeling framework and compare our approach with an open source tool Eclipse Modeling Framework (EMF) Compare. The results proved the accuracy and efficiency of our proposed approach.

Keywords*: *fine-granular model representation, model diff, model merge, conflict detection, model driven engineering

1.0 INTRODUCTION

Large software projects, which involve more than one person, essentially need efficient management of software artifacts created during software development. In the absence of an efficient management mechanism, the software products are delivered later than its schedule, may cost more than anticipated, and would have been poorly designed and documented [20]. Software Configuration Management (SCM) aims to provide an efficient controlling mechanism to avoid such problems. SCM deals with controlling the evolution of software systems [21]. Controlling the evolution of software systems requires many activities to perform, such as, construction and creation of versions of the software artifacts, performing diff activity (i.e. the identification of mappings and differences between versions), conflict detection (i.e. identifying conflicting changes), and merge activity (i.e. combining two or more versions into single one) [22].

Traditional version control systems (VCS), such as Subversion [1], CVS [2], are file-based, i.e., these approaches consider a software system as a set of text files mainly in the form of source code. However today, software development is based on model-driven activities. Model-Driven Engineering

(MDE) [23] is a modern software development technique that aims to reduce the complexity of software development by assigning models as a central role in the development process. MDE emerges as a new paradigm which creates many challenges for traditional SCM systems. With the advent of MDE for software development, models become first-class artifact, to ensure the quality of the models, they must be designed, analyzed, maintained and subject to version control. Existing SCM systems (such as CVS [2], Subversion [1] etc.), are used during the later phases of software development, notably during implementation where the main artifact is source code in the form of text files. However, these systems will not be well suited for performing configuration management tasks on the models [3]. For instance, in MDE, software documents are not only text files, but also consist of models such as, different types of UML diagrams. These models are often stored as Extensible Markup Language (XMI) formats, such as a class diagram. The order of these sections of text is irrelevant in a file and the CASE tools can store the sections representing classes or other diagram elements in arbitrary order [3–5]. Therefore, applying diff and merge operations at the level of plain text will not produce meaningful results. On the other hand, efficient management requires a close interaction within the development team. Usually, each developer is responsible for one part of the whole software system. Therefore, it is necessary to let developers work independently without disturbing their teammates as well as allowing them to share their results at a certain time and to merge their developed software artifacts with the whole software system. In this paper, we provide a generic fine-granular model merge solution for a model-based control system. Our approach for conflict detection and merging is based on our previous work in [16] for model diff, since model merge component is built on top of model diff component. First part of our approach deals with how to avoid the problems of textual representation of models. For this, we represent models at fine-granular level as graph structure metamodel. The main concepts of the metamodel are Node, Edge, Link, Operation, Attribute, Parameter and DataType, one important benefit of this metamodel is that it is generic and can be used to represent different types of UML diagrams at fine-grained level, since most of the UML diagrams except sequence diagram is presented as a graph [5]. We present a 3-way merge process, where a base and its derived are used for merging. The process of merging consists of three steps: 1) comparison of versions, 2) merging versions, and 3) conflict resolution. Comparison of versions will be done by model diff component and the results will be reused in model merge component. The process of merging cannot be completely automated [3]. Manual interaction is required in case of conflict detection in software artifacts. A conflict usually occurs if same element of an entity is modified in parallel by different teammates. In order to differentiate conflicted and non-conflicted cases we define different merge cases to analyze the difference result from the merge operation for model merge.

The implementation is done using the open source EMF [6] framework using Java as a source language. The transformation component loads the inputs model conforming to source metamodel and transform it into graph structures conforming to target metamodel. The diff component then performs the diff algorithm on the graph structures for model comparison. To benchmark our approach, we performed different test and compare our approach with the open source tool EMF Compare [7]. EMF Compare [7] uses EMF technology project to compare models in EMF. It is realized by a package of Eclipse plugins that overwrites Eclipse's standard comparing behavior. We select EMF Compare for comparison with our approach because it is an available open source tool. The main assessment criteria of our evaluation are the quality of the calculated results and the execution time.

The rest of this paper organized as follows. In Section 2, related work is given. Section 3 briefly elaborates model diff which deals with comparing model versions to identify matching and differences between them during software development. Section 4 describes the proposed model merge approach in detail. Our experiment design, results, and performance evaluation of the proposed approach is explained in Section 5. Finally, Section 6 concludes the paper and a sketch of the future work is drawn.

2.0 RELATED WORK ON MODEL-BASED SCM SYSTEMS

Many solutions to model-based SCM exist in literature. In this section, to describe the existing solutions, we categories the existing solutions in two areas, i.e. a) existing solutions in model diff, and b) existing solutions in model merge. We first describe the comparison parameters for model diff and merge approaches and then describe the related work in both areas.

2.1 Model Comparison Parameters

In order to compare the existing model diff approaches we set an evaluation criterion. The evaluation parameters are based on following features.

- Delta computation method
  - State-based method
  - Operation-based method
- Delta matching Criteria
  - UID-based criteria
  - Language-based criteria
  - Signature-based criteria
- Independency
  - Tool-independency
  - Diagram-independency
- Merge

2.1.1 Delta computation methods

Delta in SCM means the value of differences between two versions of a model. There are two ways to compute the delta: (i) State-based approach and (ii) Operation-based approach. In the state-based approach, two states, such as a base version and its successor are compared to compute the differences. In an operation-based approach, changes are described by using the original sequence of the editor operations that caused the changes. Operation-based approach records a sequence of change operations (say op1 ,. . . ,opn) while these operations occur. Delta computation methods are further explained in section 3.2.

2.1.2 Delta matching method

The basis for identifying mapping and differences is called correspondence criteria. This is a metric which defines the information needs to be considered when comparing two models to obtain mapping and differences. Existing approaches of model comparison are based on different correspondence criteria [25]. These existing approaches can be categorized based on following criterion:

a) Unique Identifier-based (UID) Criteria
b) Similarity-based Criteria
c) Language-specific Criteria
d) Hybrid Criteria

a) Unique Identifier-based Criteria

In UID-based criteria, the assumption is that each element of the model has a universally unique identifier, which is assigned to newly created element by the model repository. The identity of the object remains the same, and it is only the structure of the object is different in different versions. Model comparison is performed based on these persistent identifiers. The main advantage of this criteria is its efficiency, i.e., it is fast and requires no configuration. The problem with such a criteria is that it can only be applied to two models that are subsequent versions and created in a same development environment. Such a solution can't be applied when two models are not subsequent versions or created by different development tools [26, 28].

b) Similarity based Criteria

The similarity or signature based criteria for model comparison is based on similarity of the syntactical information of the compared elements. Persistent identifier based approaches treat the problem of model matching as true/false identity matching, while similarity-based approaches attempts to identify matching elements based on the aggregated similarity of their features [25].

The idea is that a pair of corresponding model elements needs to share a set of properties which can be a subset of their syntactical information. This syntactical information mainly includes name, type, and attributes. It may also include the context or structure similarity, in which the structure of model entities is also considered. The structure includes the number of edges connected to an entity and the end entities of a relationship. The syntactic information of an element also called signature, hence this criteria is also called signature-based matching [27]. As these approaches do not rely on persistent identifiers, they can be also used to compare models that have been constructed independently of each other. However, the developers need to specify a series of functions to calculate the identities of different types of model elements, while no such configuration effort is required in persistent identifier based approach.

c) Language-specific Criteria

The matching approaches in this category are tailored to a particular modeling language such as UML. The main advantage of this approach is that it takes into account the semantics of the target language to produce more accurate results, and it also reduces the search space. For instance, when comparing two UML models, two classes with the same name always constitute a match regardless of their location in the package structure. Similarly, two operations will be compared if they belong to the match classes, same is the case of properties and parameters, thus reducing the number of comparisons that need to be performed. However, this approach requires manual comparison algorithm, which can be challenging. As identified in [25], to ease the development of custom matching algorithms, approaches such as EMF Compare [7] and the Epsilon Comparison Language (ECL) [28] provide infrastructure that can automate the trivial parts of the comparison process, allowing developers to concentrate on the comparison logic only. Nevertheless, even with such tool support, the effort required to implement a custom matching algorithm is still considerably high.

d) Hybrid Criterion

Our approach is a hybrid criterion, which is a combination of both unique-identifier and signature-based criterion. Our justifications for the proposed approach are as follow: First, by using unique-identifier based criteria we obtained the efficiency requirement of the algorithm. In contrast to other approaches [3, 5, 17] which use this criterion at the cost of tool dependency, we handle this problem by using the name of the entity as unique-identifier. Thus even different tools are used for model development there is no dependency on the tools. Secondly, our approach also perform signature match to detect syntactic differences.

2.3 Tool-independency

In MDE, the same model can be developed using different CASE tools, e.g., a class diagram can be developed using CASE tool, like MagicDraw [29] or MS Visio [30]. There is always the problem of compatibility when a model developed using one CASE tool with other due to their different internal representation. For instance, the class diagram developed in MarigDraw is not supported by MS Visio. In such a scenario the goal of model diff tool is the independency from the tools which were used to create the diagram.

2.4 Diagram-independency

There are different diagram types that can be developed during software development lifecycle. The model diff tool should be applicable to a large set of diagram types [24]. However, in our study we have noticed that some approaches are tailored to some specific diagrams [5, 18, 31] and are not applicable to other diagrams while some approaches [24, 26, 7,17,18] offer generality.

2.5 Model Merge

Model merge deals with combining two or more versions of a model into a single one [21]. The process of merging consists of three steps: 1) comparison of versions, 2) merging versions, and 3) conflict resolution.

2.2 Existing Solutions of Model Diff

Alanen and Porres in [17] discuss the difference and union of models in the context of a version control system. Three meta-model-independent algorithms are given that calculate the difference between two models, merge, and calculate the union of two models. However, these algorithms crucially rely on the existence of a universally unique identifier for each model element. The output produced by the approach is in form of a sequence of edit operation, while in our approach the results are brought back into a model which is more comprehensible. Ohst et al. [5] addresses the problem of how to detect and visualize differences between versions of UML documents, such as, class or object diagrams. The approach assumes that each model element has a unique identifier for model comparison. To show the differences between two documents, the unified document is used that contains the common and specific parts of both base documents; the specific parts are highlighted. EMF Compare [7] is an open source tool to compare models in EMF. It is realized by a package of Eclipse plugins that overwrite Eclipse's standard comparing behavior. EMF Compare uses a generic algorithm for model comparison. The comparison is performed in two-phases: In the first phase the match engine tries to find similar elements and creates a match model. Based on this model, different engine is used to generate detailed information about the differences of certain model elements. A difference model is the result of the second phase. Both match and difference model are EMF models and therefore can be treated like any other model. Compared to our approach, the diff and match model produced by EMF Compare cannot be converted to graphical representation.

Table 1. Comparison chart with existing approaches

Approaches	Delta Computation Method		Calculation Criteria			Independency		Merge
	State-based	Operation-based	UID-based	Signature-based	Language-based	Tool	Diagram	
Alanen et al.	√	×	√	×	×	×	√	√
DSMDiff	√	×	×	√	×	√	√	×
D.Ohst	√	×	√	×	×	×	×	√
SiDiff	√	×	×	√	×	√	√	×
UMLDiff	√	×	×	×	√	√	×	×
Pounamu	×	√	√	×	×	×	√	√
Girschick	×	√	√	×	×	×	×	×
Workflow	√	×	√	×	×	√	×	√
Odyssey	-	-	-	-	-	-	×	√
AMOR	-	-	-	-	-	×	-	√
CoObRA	×	√	√	×	×	×	-	√
Unicase	×	√	√	×	×	×	-	√
Our Approach	√	×	√	√	×	√	√	√

Legends: √ Supported × Not supported - Unknown

Xing et al. [18] presented an automated UML-aware structural-differencing algorithm, UMLDiff, for automatically detecting structural changes between the designs of subsequent versions of object-oriented software. It takes as input, two class models of a java software system and reverse engineers from two corresponding code versions. This approach uses language-based matching criterion and identifies corresponding entities based on their name and structure similarity. If two objects have the same name, they are identified as equal, if not, their structural similarity is considered, computed from the similarity of names and other criteria specific of the considered entity type. Kelter et al. [19] presented a generic algorithm SiDiff which uses an internal data model comparable with simplified UML meta-model. A diagram is extracted from

an XMI file and is represented as a tree consisting of a composition structure. In this approach, the model elements are characterized by their elements and the difference algorithm start s with a bottom-up traversal at the leaves of the composition tree. This approach uses a signature-based matching criterion. The Pounamu approach presented in [8] describes a generic approach for diff and merge via a set of plug-in components. Plug-ins is developed for the meta-CASE tool Pounamu which support version control, visual differencing and merging. The approach uses operation-based method for different computation which results in the dependency of the tool in which diagrams are edited using a universal unique identifier (UID)-based matching criteria.

2.3 Existing Solutions of Model Merge

The Pounamu approach presented in [8] is a generic approach for diff and merge via a set of plug-in components. Plug-ins are developed for the meta-CASE tool Pounamu which support version control, visual differencing, and merging. The approach uses operation-based method for difference computation which results in the dependency of the tool in which diagrams are edited, contrary to our approach which uses State-based approach. The approach uses a universal unique identifier (UID)-based matching criteria.

An approach for comparison and versioning of scientific workflows is presented in [9]. A version model for workflow is presented as a directed graph. The approach is based on modified 3-way merge algorithm called 3-way subgraph diff/merge algorithm which is based on graph theory. A 3-way subgraph is analyzed as an atomic part and taken into consideration for merge decisions. The main problem with the approach is that it dealt only with one specific kind of model, i.e., workflows, thus, it is not generic. The approach uses UID-based matching criteria while our approach uses hybrid criteria. Merging UML documents as described in [3, 5] splits the merging process into three steps. First, a pre-merged document is created, then identified conflicts are solved manually and finally the merged document is created. Conflicts occur if the same attribute has been changed in both versions, or if an entity has been modified in one version and deleted in the other version. In case of change conflict the user has to decide which modification should be applied. Similarly, in case of deletion-modification conflicts, user has to decide whether the entity should be deleted or modified. The pre-merged document is an extended unified document consisting of common parts, automatically merged parts and conflicts. Software document is transformed into an abstract syntax tree at fine-grained level. These approaches work for UML class diagram specifically, as compared to our approach, which is generic and work for both UML and domain specific models. Furthermore, these approaches only consider unique identifiers for comparison while our approach uses a hybrid criterion. In CoObRA versioning framework [10], all edit operations that are executed on the diagrams are logged by the tool.

The approach in [3, 5] uses 3-way merging but gives priority to the version that was committed first. A developer has to check the version v_1 of the repository into the local workspace to modify it by applying the operation sequence Δ_2. But if the operation sequence Δ_1 has been applied to the version in the repository, where the developer fails to commit his/her changes, then the developer has to update his/her local version first. This means applying the changes Δ_1 on the origin version v_1 to reach the actual version v_2 stored in the repository, then trying to apply the change operations in Δ_2 again. The difference between this approach and our approach is that the former is based on operation-based deltas and hence dependent on the editor tool which logged the edit operations. Furthermore, these approaches use Uid-based matching criteria and do not support conflict resolution. Oliveira et al. [11] have implemented Odyssey-VCS to provide configuration management support for CASE tools that work with UML models. This approach uses XMI as the protocol for communication between CASE tools and the VCS. Oliveira at al. [11] only describe the implementation of merge in their approach. When a conflict is detected, the developer receives a conflict description along with the original, user and current configurations. After performing manual merge, developer resubmits the model to the repository. Merge algorithm follows a 3-way merge approach. Their approach is limited to UML models only. Furthermore, this approach is based on performing diff/merge operation on structured data, i.e., XMI, which is not suitable for such operations.

Kogel et al. [12, 13] present a SCM approach for software engineering artifacts that manages change in graph-structured artifacts and supports traceability. Their approach is based on operation-based

deltas, change packages and product versioning. They have also developed Unicase [14], which is a CASE-tool integrating models that allows viewing and editing models in the form of textual, tabular, and diagrammatical visualization, stored in a repository and can be versioned. Three-way and directed delta approach is applied for the merge process, where the edit operations are obtained from the Unicase client. The difference between their approach and ours is that the former is based on operation-based deltas and hence dependent on editor tool used. Furthermore their approach uses UID-based matching criteria.

Altmanninger et al. [15] present AMOR (Adaptable Model Versioning), a semantic-based methods and techniques to leverage version control in MDE. It was claimed that AMOR supports precise conflict detection, i.e., previously undetected and wrongly indicated conflicts are avoided. This is because AMOR incorporates knowledge about the type of modifications and knowledge of the modeling concepts used. They also claim that AMOR focuses on intelligent conflict resolution by providing techniques for the representation of conflicting modifications as well as suggesting proper resolution strategies. AMOR targets an adaptable versioning framework, empowering modelers to flexibly balance between reasonable adaptation effort and proper versioning support while ensuring generic applicability in various domain-specific modeling languages and associated tools. AMOR uses the semantics of the modeling concepts. The main focus of the approach is on conflict detection and resolution, and it is not clear on the method used for model diff.

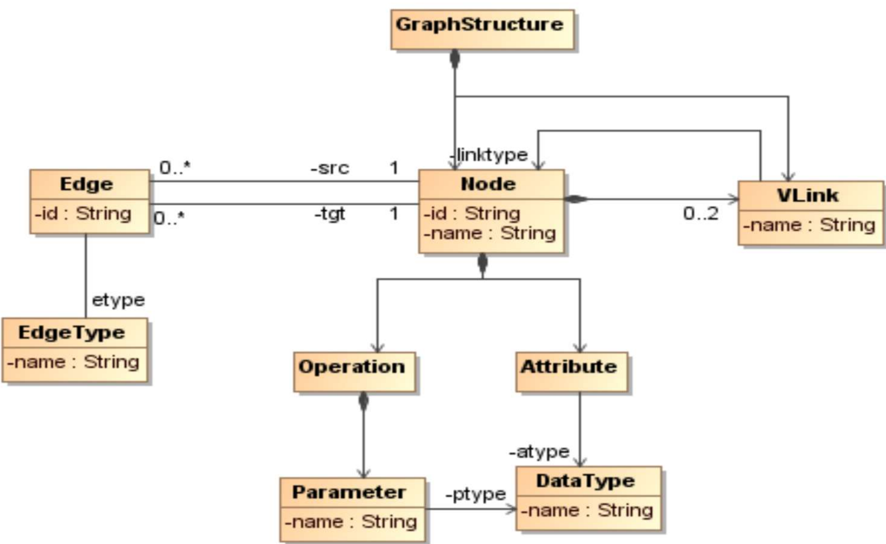

Fig. 1. Graph structure data model

3.0 MODEL DIFF

The goal of this work is to develop a generic model merge solution for merging two versions of a model. However since model merge component is built on top of model diff component, we need to perform diff activity first. Model diff deals with comparing two versions of a model to detect differences and matches between them. We address the problem of computing the mappings and differences between the models by exploring the issues of, a) how to represent models at a fine-grained level, b) how to compute deltas, namely the state-based or operation-based approaches, and c) designing algorithms that can be used to discover the mappings and differences between the models.

3.1 GRAPH STRUCTURE REPRESENTATION

In software development life cycle two main types of software documents are text files and graphical models. Text files may contain source code, documentation, software requirement specification (SRS) document, test

reports and so on, whereas graphical models can be UML models. A model can be represented in three different ways [28], i) the graphical representation i.e. the diagram itself, ii) the persistence representation, e.g. XMI, and iii) intermediate representation, e.g. syntax tree or graph structure. To avoid the problems mentioned in Section 2.1, we represent models at fine-granular level as graph structures. A graph structure data model defines the elements, attributes, and relationships between the elements at the fine-grained level [9]. The selection of an appropriate data model has a strong impact on the capabilities of the diff and merge tool. For instance, a simple data model could perform a simple and efficient diff and merge operations for versions of a model. In our proposed approach, at a fine-grained level, we represent models in an intermediate representation, as graph structures (e.g. as shown in Fig. 1). The proposed structure represents graph with typed elements that can be decorated with attributes. The basic elements of the metamodel are: Nodes, Edges, Links, Operations, Attributes, Parameters, and DataTypes. Besides other advantages, one more important benefit of the metamodel is that it is generic and can be used to represent various types of UML models, at the fine-grained level. This is an important issue, as most of the UML diagrams except that of the sequence diagram is represented by a graph [6].

Node: Node resembles an entity (e.g. a class in a Class diagram, or an activity in an Activity diagram) of a model. Nodes are identified by an id and may contain a number of attributes. A Node can be connected with other Nodes by different form of associations. In our graph structure the connection between the Nodes are represented by VLinks and Edges.
Attribute: Attribute represents data which represent features of node. They are identified by name and have a data type.
DataType: Data types model simple types such as Int, String, Boolean etc. They are identified by name and are most commonly used as attribute types.
Edge: Edge models the type of association between two nodes. Every edge has source and target node. Different types of association between nodes can be identified by Edgetype, which includes association, inheritance, containment etc.
Operation: Operation represents the operations of a Node. An operation is identified by a name and a list of zero or more typed parameters representing the overall signature. Like all typed elements, an operation specifies a type, which represents the return type; it may be null to represent no return type.
Parameter: Parameter models an operation's input parameters. A parameter is identified by a name and type of a value that may be passed as an argument corresponding to that parameter.
VLink: Node can have links which express unidirectional relationships between two Nodes. Vlink are used to connect all the nodes in a linear order. It is used as auxiliary element which do not map to any element of the source model.

Mapping between UML and Graph structure

The mapping between source models like UML and Graph structure will be done based on the concepts defined above. For instance, in Table 2.0 the mappings between UML class and activity diagram with the Graph structure elements are given.

Class-mapTo-Node: In UML the classifier Class defines a set of model entities. The corresponding concept in Graph structure is defined by Node. Therefore, we map Class onto Node.

Activity-mapTo-Node: Similar to the classifier Class, classifier Activity also defines a set of model entities. The corresponding concept in Graph structure is defined by Node. Since Activity is a supertype of classes InitialNode, ForkNode, MergeNode, JoinNode, DecisionNode, CallBehaviourAction, ActivityFinalNode and CentralBufferNode therefore, we map all the subtypes of Activity onto Node.

Operation-mapTo-Operation: Operations belonging to Class are defined in the UML as Operation. The corresponding concept in Graph structure is defined by Operation belonging to Node. Therefore, we map Operation onto Operation.

Attribute-mapTo-Attribute: Attributes belonging to Class are defined in the UML as Attribute. The corresponding concept in Graph structure is defined by Attribute. Therefore, we map Attribute onto Attribute.

Parameter-mapTo-Parameter: Parameters are defined in the UML as Parameter. The corresponding concept in Graph structure is defined by Parameter. Therefore, we map Parameter onto Parameter.

DataType-mapTo-DataType: Datatypes are defined in the UML as DataType. The corresponding concept in Graph structure DSL is defined by DataType. Therefore, we map DataType onto DataType.

Association-mapTo-Edge: A relationship between two entities is described by Association in UML. The corresponding concept in Graph structure is defined by Edge. Therefore, we map Association onto Edge. The type of Association corresponds to the type of Edge, i.e., EdgeType.

Table 2: Mappings between UML and GraphStructure DSL

UML	**GraphStructure**
Class	Node
InitialNode Node	Node
ForkNode Node	Node
MergeNode Node	Node
JoinNode Node	Node
DecisionNode Node	Node
CallBehaviorAction Node	Node
ActivityFinalNode Node	Node
CentralBufferNode Node	Node
Reference	Edge
ControlFlow	Edge
ObjectFlow	Edge
Attribute	Attribute
Operation	Operation
Parameter	Parameter

3.2 DELTA COMPUTATION

When comparing two versions of a model, a model mapping defines those model entities that represent a single conceptual entity, while the unmatched entities represent the model differences. The difference between the two versions of a model is known as delta. There are two ways to compute delta or difference between two versions of a model:

i) State-based approach
ii) Operation-based approach

i) State-based Approach

In state-based approach two states, e.g., a version and its successor are compared to determine the differences. Deltas are reconstructed using a differencing algorithm that compares the different state representations. The delta in state-based approach is known as symmetric delta. In this approach changes are recorded after they occur. A symmetric delta of two versions v_1 and v_2 contains all elements which belong to v1 but not to v2 and vice versa. Using set notation loosely, the symmetric delta may be written as $\Delta(v_1,v_2) = (v_1 \setminus v_2) \cup (v_2 \setminus v_1)$ [24]. Since changes are derived after they occur, so change not considered at first level concept is state-based approach. A big advantage of state-based approach over an operation-based approach is a total separation of modeling tools and the version control system. This is because version control system needs not to observe the changes as they occur.

i) Operation-based approach

In operation-based approach, changes are described by using the original sequence of editor operations that caused the changes [24]. Operation-based approaches record a sequence of change operations $op_1, \ldots, op_n$ while they occur and when these operations are applied to one version v_1, yields another version v_2. The changes in an operation-based approach are considered as a first class concept. The difference calculation is not required, as the changes are already available by design. The delta in operation-based approach is known as directed delta. A directed delta may be formalized as a sequence $\Delta = op_1, \ldots, op_m$ such that $\Delta(v_1) = v_2$. The major drawback of operation-based approach is its

dependency on the editor tool. The approach needs the version control system to be present when the changes occur, i.e., when the models are manipulated by modeling tool. This requires the integration of version control system into the modeling tool.

A big advantage of the state-based approach over an operation-based approach is a total separation of modeling tools and the version control systems (VCS). As such, we also adapt a state-based procedure in our proposed approach, as it provides generality and independency of the tools.

3.3 MODEL DIFF COMPARISON ALGORITHMS

The model diff comparison algorithm takes two versions of a model as input and produces output in two sets MapSet{} and ChangeSet{}. MapSet{} that contains all of the pairs of model elements that are similar in both versions, having the same identifier. The ChangeSet{} that contains such entities, whose contents (e.g. the attributes of) are modified in the second version. The algorithm takes node-signature and edge-signature of elements for comparison. The Node-signature consists of node IDs, attributes, and operations; whereas, for the structural properties of the nodes, the algorithm compares the edge-signature of the nodes.

4.0 MODEL MERGE SOLUTION

This section presents a solution to the problem of merging in model-based software configuration management systems. Model merge deals with combining two or more versions of a model into a single one, based on model diff activity. As discussed earlier, traditional SCM systems use textual or structured data to represent models at fine-grained level, which is not a suitable representation to determine the differences or to merge software diagram produced in the early phases of software development, such as UML diagrams [3]. We present a 3-way merge process, where a base and its derived versions are used for merging. The process of merging consists of three steps: 1) comparison of versions, 2) merging versions, and 3) conflict resolution. The comparison process of versions is done by model diff component described in section 3. We reuse the results of model diff in merge activities as shown in Figure 2. The first step is to transform base and derived versions into graph structures. The base and derived versions are the instances of any source model, whereas the transformed models are the instances of graph structure model. After transformation, a model diff is applied on base version vs derived V1 and base version vs derived V2. Then a 3-way merge algorithm is applied using the merge policy to compares the versions for matched, unmatched, added and deleted elements. Based on difference result and merge policy, the possible actions can be categorize into add, delete, include changed and include unchanged elements. In case of the conflicted elements, a manual interaction is carried out to resolve the conflict. A conflict usually occurs if same element of an entity is modified in parallel. To differentiate conflicted and non-conflicted cases, we define different merge cases. Merge cases are used to analyze the difference result in order to perform the merge operation. Finally the merge diagram will be obtained.

4.1 Merge Policy

Merge policy is used for possible automation during merge process to differentiate conflicted and non-conflicting cases, and to identify the need for manual interaction. For 3-way merging we need to compare base version elements with derived version elements. We have identified 11 different merge cases (cf. table 1) based on which we created our merge policy. In case 1 the base element remains unchanged in both derived versions. Case 2 & 3 represent base element changed in one version while remains unchanged in second version. Case 4 & 5 represent base element deleted in one version while unchanged in other version. In case 6 base element is deleted in both versions. Case 7 & 8 represents an element is added in either version. Case 9 & 10 represent base element changed in one version while delete in other version. Case 11 represent base element is changed in both versions. Note that case 9, 10 and 11 are conflicted scenario, since same element is modified parallel in both versions.

Based on merge cases, we establish our merge policy as follows: If the base element is unchanged in both versions, then the unchanged element is included into the merge version. If the base element is changed in both versions, then both changed elements are included in merge version. Since this is a conflicted scenario, merge version will be manually updated to resolve the conflict. If the base element

is unchanged in one version and changed in other version then both changed and unchanged element will be included into merge version. If the base element is changed in one version and deleted in other version then the changed element will be included into merge version. Since this is also a conflicted scenario, merge version will be manually updated to resolve the conflict. If the element remains unchanged in one version and deleted in other version then the element will be considered deleted and should not be included in merge version. If the element is deleted in both version then it is also considered deleted and should not be included in merge version. All elements that are added in the derived versions are included into the merged version.

Table 3: Merge Cases

Cases	Base Vs V1	Base Vs V2	Action	Type
1	unchanged	unchanged	include unchanged	-
2	unchanged	changed	include changed	-
3	changed	unchanged	include changed	-
4	deleted	unchanged	deleted	-
5	unchanged	deleted	deleted	-
6	deleted	deleted	deleted	-
7	new	-	include new	-
8	-	new	include new	-
9	changed	deleted	include changed	conflict
10	deleted	changed	include changed	conflict
11	changed	changed	include both changed	conflict

4.3 Merging Algorithm

Our 3-way merge algorithm consists of three parts: mergeModelsP1, mergeModelsP2, and mergeModelsP3. First two parts deal with non-conflicting cases whereas the third part deals with conflicting cases. Furthermore, mergeModelsP2 given in algorithm 2.0 covers those cases which can be automated in the merge process.
Following notations are used in merge algorithm.
Base version V: Base version represents the original model.
Derived version V1: Derived version V1 represents the first modification to the base version.
Derived version V2: Derived version V2 represents the second modification to the base version.
MapSet{}: To represent the map elements of the base version and the derived versions.
ChangeSet{}: To represent the elements which are modified in the derived versions.
NewSet{}: To represent the elements which are added in the derived versions.
DeleteSet{}: To represent the elements which are deleted in the derived versions.

The whole merge algorithm works as follows; for the given model 'm', the diff results of base version 'm' and its derived version V 1 and the base 'm' and its derived version V 2 a merge model will be generated based on the cases given in Table 3. The algorithm 1.0 mergeModelsP1 starts from the first case of the 3-way merge, i.e., if a base element is unchanged in both derived versions, it is mapped and included into merge model as mapped node (lines 1 – 8) in Algorithm 1.0. For all the elements in V1's MapSet{} and V2's MapSet{} (both MapSet{} computed by model diff component) are compared, and the match element is added to merge model. Then the second case of the 3-way merge is covered (lines 9 – 17) in Algorithm 1.0, if a base element is unchanged in version V 1 and changed in version V 2 then both the unchanged and changed element are included into merge model. For this all the elements in V1's MapSet{} and V2's ChangeSet{} are compared and the match element are added to merge model as changed node in version 2 and mapped node in version 1. Then the third case of the 3-way merge is covered (lines 18 – 26) in Algorithm 1.0, if a base element is changed in

version 1 and unchanged in version 2 then both the unchanged and changed element will be included into merge model. For this all the elements in V1 ChangeSet{} and V2 MapSet{} will be compared and the match element will be added to merge model as changed node in version 1 and mapped node in version 2. Algorithm 2.0 mergeModelsP2 starts from the fourth case of the 3-way merge (lines 1 – 8), i.e., if a base element is deleted in version 1 and unchanged in version 2 then element is not included into merge model, i.e., considered deleted. For this, all the elements in V1 DeleteSet{} and V2 MapSet{} are compared and the match element is not added to the merge model. Then the fifth case of the 3-way merge is covered (lines 9 – 16), i.e., if a base element is unchanged in version 1 and deleted in version 2 then element is not be included into merge model, i.e., considered deleted. For this, all the elements in V1 MapSet{} and V2 DeleteSet{} are compared and the match element is not be added to merge model. Then the sixth case is covered (lines 17 – 24), i.e., if a base element is deleted in both versions. For this, all the elements in V1 DeleteSet{} and V2 DeleteSet{} are compared and the match element is not be added to merge model. Then the seventh and eighth cases are covered (lines 25 – 32), i.e., if an element is new in either version then the element are included into merge model. For this, all the elements in V1 NewSet{} and V2 NewSet{} are traversed and the new elements are added to merge model.

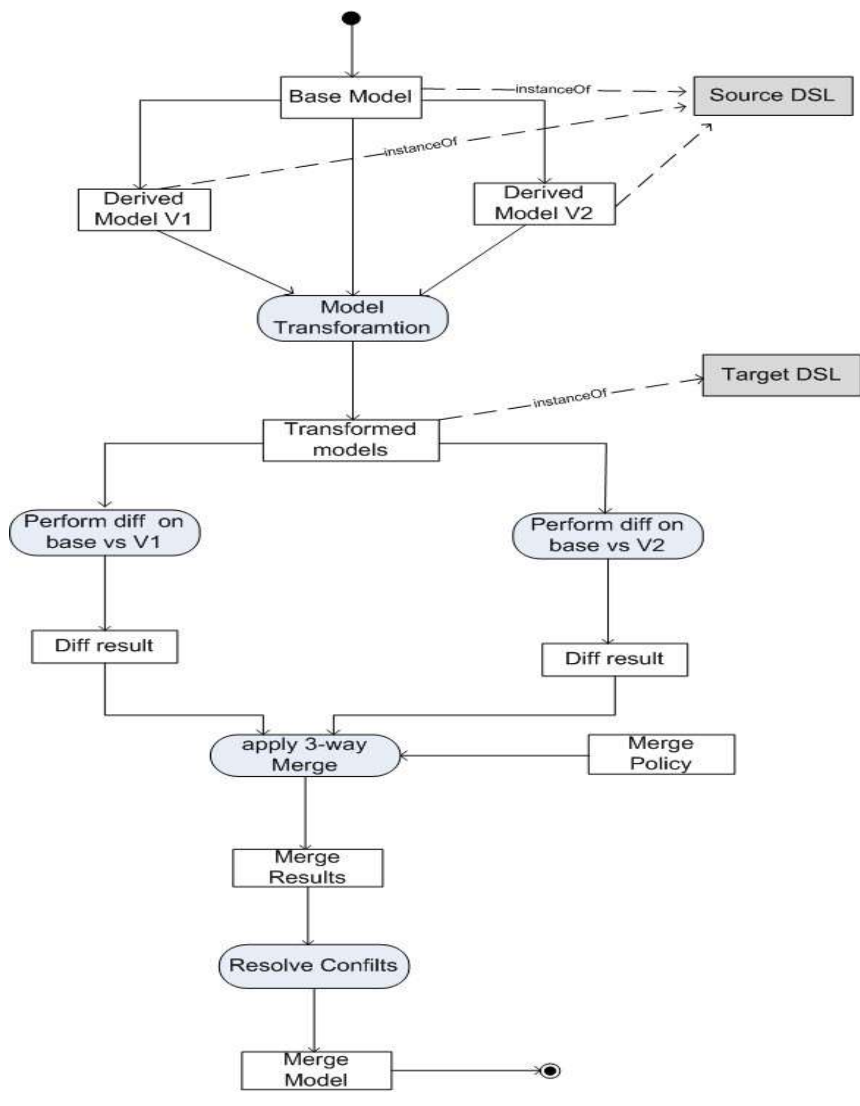

Fig. 2. Merging workflow

The conflicting cases are covered in mergeModelsP3 given in Algorithm 3.0. The algorithm starts from the ninth case of the 3-way merge (lines 1 − 9), i.e., if a base element is deleted in version 1 and changed in version 2 then both the deleted and changed element are included in the merge model. For this all the elements in V1 DeleteSet{} and V2 ChangeSet{} are compared and the match element is added to merge model as deleted node in version 1 and changed node in version 2. Then the tenth case is covered (lines 10 − 18), i.e., if a base element is changed in version 1 and deleted in version 2 then both the deleted and changed element are included into the merge model. For this all the elements in V2 DeleteSet{} and V1 ChangeSet{} are compared and the match element will added to merge model as deleted node in version 2 and changed node in version 1. Then the eleventh case is covered (lines 19 − 34), i.e., if a base element is changed in both version 1 and version 2 then both the changed elements will be included into merge model. For this all the elements in V1 ChangeSet{} and V2 ChangeSet{} will be compared and the match element will added to merge model as changed node in version 1 and changed node in version 2.

Algorithm 1.0 mergeModelsP1

```
Require: V1 MapSet{}, V2 MapSet{}, V1 ChangeSet{}, V2 ChangeSet{},
 1: V1 DeleteSet{}, V2 DeleteSet{}, V1 NewSet{},V2 NewSet{}
 2:                    //      Case 1: Unchanged + Unchanged
 3: for all elements of V1 MapSet{} traverse V1 MapSet{} do
 4:     for all elements of V2 MapSet{} traverse V2 MapSet{} do
 5:         if any element of V1 MapSet{} equals to V2 MapSet{} then
 6:             add element to the MergeModel as mapped node
 7:         end if
 8:     end for
 9: end for
10:                    //      . Case 2: Unchanged + Changed
11: for all elements of V2 ChangeSet{} traverse V2 ChangeSet{} do
12:     for all elements of V1 MapSet{} traverse V1 MapSet{} do
13:         if any element of V1 MapSet{} equals to V2 ChangeSet{} then
14:             add element to the MergeModel as mapped node in V1
15:             add element to the MergeModel as mapped node in V2
16:         end if
17:     end for
18: end for
19:                    //      . Case 3: Changed + Unchanged
20: for all elements of V1 ChangeSet{} traverse V1 ChangeSet{} do
21:     for all elements of V2 MapSet{} traverse V2 MapSet{} do
22:         if any element of V1 ChangeSet{} equals to V2 MapSet{} then
23:             add element to the MergeModel as changed node in V1
24:             add element to the MergeModel as mapped node in V2
25:         end if
26:     end for
27: end for
```

Algorithm 2.0 mergeModelsP2

```
1:                     //      Case 4: Deleted + Unchanged
2: for all elements of V1 DeleteSet{} traverse V1 DeleteSet{} do
3:     for all elements of V2 MapSet{} traverse V2 MapSet{} do
4:         if any element of V1 DeleteSet{} equals to V2 MapSet{} then
5:             do not include element in MergeModel
6:         end if
7:     end for
8: end for
9:                     //      Case 5: Unchanged + Deleted
```

```
10: for all elements of V2 DeleteSet{} traverse V2 DeleteSet{} do
11:     for all elements of V1 MapSet{} traverse V1 MapSet{} do
12:         if any element of V2 DeleteSet{} equals to V1 MapSet{} then
13:             do not include element in MergeModel
14:         end if
15:     end for
16: end for
17:                         //    Case 6: Deleted + Deleted
18: for all elements of V2 DeleteSet{} traverse V2 DeleteSet{} do
19:     for all elements of V1 DeleteSet{} traverse V1 DeleteSet{} do
20:         if any element of V2 DeleteSet{} equals to V1 DeleteSet{} then
21:             do not include element in MergeModel
22:         end if
23:     end for
24: end for
25:                               Case 7: New element in V1
26: for all elements of V1 NewSet{} traverse V1 NewSet{} do
27:     add element to the MergeModel as new node in V1
28: end for
29:                               Case 8: New element in V2
30: for all elements of V2 NewSet{} traverse V2 NewSet{} do
31:     add element to the MergeModel as new node in V2
32: end for
```

Algorithm 3.0 mergeModelsP3

```
1:                          //      Case 9: Deleted + Changed
2: for all elements of V1 DeleteSet{} traverse V1 DeleteSet{} do
3:     for all elements of V2 ChangeSet{} traverse V2 ChangeSet{} do
4:         if any element of V1 DeleteSet{} equals to V2 ChangeSet{} then
5:             add element to the MergeModel as deleted node in V1
6:             add element to the MergeModel as changed node in V2
7:             note conflict for V1
8:         end if
9:     end for
10: end for
11:                         //        Case 10: Changed + Deleted
12: for all elements of V2 DeleteSet{} traverse V2 DeleteSet{} do
13:     for all elements of V1 ChangeSet{} traverse V1 ChangeSet{} do
14:         if any element of V2 DeleteSet{} equals to V1 ChangeSet{} then
15:             add element to the MergeModel as deleted node in V2
16:             add element to the MergeModel as changed node in V1
17:             note conflict for V1
18:         end if
19:     end for
20: end for
21:                         //        Case 11: Changed + Changed (V1)
22: for all elements of V1 ChangeSet{} traverse V1 ChangeSet{} do
23:     for all elements of V2 ChangeSet{} traverse V2 ChangeSet{} do
24:         if any element of V1 ChangeSet{} equals to V2 ChangeSet{} then
25:             add element to the MergeModel as changed node in V1
26:             note conflict for V1
27:         end if
28:     end for
29: end for
30:                         //        Case 11: Changed + Changed (V2)
```

```
31: for all elements of V2 ChangeSet{} traverse V2 ChangeSet{} do
32:     for all elements of V1 ChangeSet{} traverse V1 ChangeSet{} do
33:         if any element of V2 ChangeSet{} equals to V1 ChangeSet{} then
34:             add element to the MergeModel as changed node in V2
35:         end if
36:     end for
37: end for
```

5.0 EMPIRICAL PERFORMANCE EVALUATION

To benchmark our approach, we performed several tests and compare our approach with the open source tool EMF Compare[7]. The main assessment criteria of our evaluation are the quality of the calculated solutions and the required runtime.

Case Study 1:

We have performed a controlled experiment on a library system class model to benchmark the diff algorithm. We took six different cases to run our tests consisting different versions of the input models, in term of the size of the versions of the models. Since we are performing a 3-way merge, we took 3 versions of the model in every case, where one version is the base case and the other two versions are the derived versions. The tests were performed on a standard PC Intel Core Duo CPU P9400 with 4 GB memory. The results of the evaluation are shown in Figure 3 for our approach and Figure 4 for EMF Compare. Figure 5 shows the comparison table for both approaches and Figure 6 shows the comparison chart. The first column of the table shows different cases (Cases) on which the test were performed. The second column (Model Versions) shows the three versions of the model. The third column (Ele.) of the table shows the sum of the number of XMI elements of all versions. The fourth column (Cla.) shows the total number of classes. The fifth column (Diff Det. ∑) shows the total number of differences detected using 3-way merge. The sixth column (Ele. Ad. Md.) shows the number of elements which are either added or modified. The seventh column (Ele. Del.) shows the number of elements which are deleted. The eighth column (Conflict Changes) shows the number of conflicting changes. The ninth column (Diff. Ele. %) shows the percentage of changed elements between the versions. Finally, the last column (Exe. Time (ms)) shows the runtime of the diff operation in milliseconds. The runtime shown in the table are the average of five test runs.

5.1 Case A

In first case, we took the three versions Vb, V0 and V1 of the model of relatively small size, where Vb represents the base version and V0, V1 represent the derived version 0 and 1, respectively. The number of elements was 34 and the number of classes was 13 in three versions. From this test, the total number of differences detected was 4 element addition & modification changes. The percentage of the detected differences was 11% and the execution time was 470ms. For EMF Compare, the same results for element addition & modification changes were obtained but EMF Compare also showed the difference of one deleted element. Thus the total number of differences showed by EMF Compare was 6. Our approach also detected this difference but in the merge model we didn't show this difference because the deleted element was deleted in V1 and unchanged in V0 and according to our merge policy, a non-conflicting scenario and the deleted element is not included in the merge model. The execution time of EMF Compare was 632 ms, compared to 470ms for our approach.

5.2 Case B

In case B, we have increased the number of elements from 34 to 55 and the number of classes from 13 to 18. The two derived versions of the base version are V2 and V3. The total number of differences detected by our approach was 25 (21 element addition and modification changes, 2 element delete changes, and 2 conflicting changes). The conflicting changes exist because one of the attribute and reference of a

the class was modified in V1 while deleted in V2. The percentage of the detected differences is 45%, and the execution time is 481ms. For EMF Compare, the total number of differences detected by EMF Compare was 21, 4 less than our approach. By closely analyzing the output of EMF Compare, it was observed that EMF Compare did not record the reference addition of new elements' in the second version. Our approach identified 21 changes for element addition and modification, whereas EMF Compare only detected 17. This is because EMF Compare did not detect the 4 new references of 3 new classes added in both derived versions. EMF Compare takes 653ms to perform the merge operation for the given input, compared to ours of 481ms. Furthermore, the difference between the execution time in performing case A and case B was 11ms in our approach and 21ms for EMF Compare.

5.3 Case C

In case C, 119 elements and 33 number of classes was used. The two derived versions in case C were V4 and V5. The total number of differences detected by our approach is 40, (36 element addition and modification changes, 2 element delete changes, and 2 conflicting changes). The percentage of the detected differences was 33% in 517ms. For EMF Compare, the total number of differences detected was 27 (23 element addition and modification changes, 2 element delete changes, and 2 changes of reordering of the elements), 13 less than the differences calculated by our approach. The 2 conflicting changes were not identified by EMF Compare, which are layout change, shows the inaccuracy of the results execution time by EMF Compare was 693ms, whereas our approach took 517ms. Furthermore, the difference between the execution time in performing case B and case C was 36ms for our approach and 40 md for EMF Compare.

5.4 Case D

In case D, we have increased the number of elements from 119 to 181 and the number of classes from 33 to 50. The two derived versions in case D of the model were V5 and V6. The total number of differences detected by our approach is 67 (55 element addition and modification changes, 4 element delete changes, and 8 conflicting changes). The percentage of the detected differences is 37% in 535ms. For EMF Compare, the total number of differences detected is 50 (40 element addition and modification changes, 3 element deleted changes, 5 conflicting changes, and 2 changes of reordering of the elements), 17 less than the differences calculated by our approach, the 3 unidentified conflict changes, 1 unidentified delete change, and 2 reordering changes identification, errors showed the inaccuracy of the results. EMF Compare took 741ms to perform the merge operation whereas our approach took 535ms. The difference between the execution time in performing case C and case D was 18ms in our approach and 48ms for EMF Compare.

5.5 Case E

In case E, we have used 240 elements and 61 classes. The two derived versions in case E of the model were V6 and V7. The total number of differences detected by our approach for case E is 78 (61 element addition and modification changes, 7 element delete changes, and 10 conflicting changes).

Cases	Model Versions	Ele.	Cla.	Diff. Det. Σ	Ele. Ad. & Md.	Ele. Del.	Conflict. Changes	Diff. Ele. %	Exe.Time (ms)
A.	Vb, V0, V1	34	13	4	4	0	0	11%	470
B.	Vb, V2, V3	55	18	25	21	2	2	45%	481
C.	Vb, V4 ,V5	119	33	40	36	2	2	33%	517
D.	Vb, V5, V6	181	50	67	55	4	8	37%	535
E.	Vb, V6, V7	240	61	78	61	7	10	32%	546
F.	Vb, V7, V8	290	73	96	82	5	9	33%	560

Fig. 3: Test results of our approach

The percentage of the detected differences is 32% executed in 546ms. For EMF Compare, the total number of differences detected by EMF Compare is 64, (50 element addition and modification changes, 4 element deleted changes, 8 conflicting changes, and 2 changes of reordering of the elements) in 760ms. The difference between the execution time in performing case D and case E was 11ms for our approach and 19ms for EMF Compare.

Cases	Model Versions	Ele.	Cla.	Diff. Det. Σ	Ele. Ad. & Md.	Ele. Del	Conflict. changes	Diff. Ele. %	Exe.Time (ms)
A.	Vb, V0, V1	34	13	6	4	2	0	17%	632
B.	Vb, V2, V3	55	18	21	17	2	2	38%	653
C.	Vb, V4 , V5	119	33	27	23	2	0	22%	693
D.	Vb, V5, V6	181	50	50	40	3	5	27%	741
E.	Vb, V6, V7	240	61	64	50	4	8	32%	760
F.	Vb, V7, V8	290	73	75	64	3	5	25%	775

Fig. 4: Test results of EMF Comp approach

5.6 Case F

For case F, we have increased the number of elements from 240 to 290 and the number of classes from 61 to 73. The two derived versions in case F of the model were V5 and V6. The total number of differences detected by our approach is 96 (82 element addition and modification changes, 5 element delete changes, and 9 conflicting changes). The percentage of the detected differences is 33% in 560ms. For EMF Compare, the total number of differences detected is 75 (64 element addition and modification changes, 3 element deleted changes, 5 conflicting changes, and 3 changes of reordering of the elements), 21 less than the differences calculated by our approach. The execution time taken by EMF Compare was 775ms and the difference between the execution time in performing case E and case F was 14ms in our approach and 15ms for EMF Compare.

Cases	Reordering differences	Unidentified differences	Exe.Time Our App.	Time Diff per Case Our App.	Exe.Time EMFComp	Time Diff per case EMFComp
1	0	0	470	-	632	-
2	0	4	481	11	653	21
3	2	15	517	36	693	40
4	2	19	535	18	741	48
5	2	16	546	11	760	19
6	3	24	560	14	775	15

Fig. 5: Results comparison

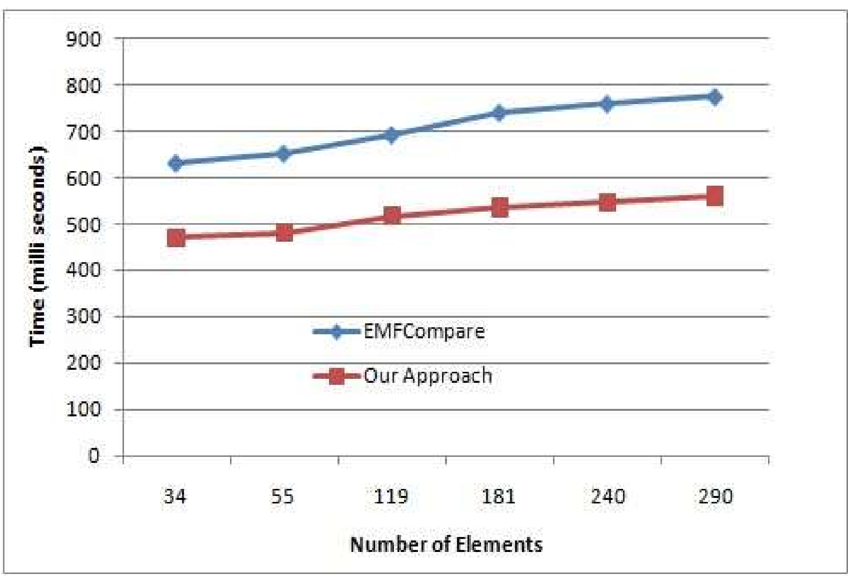

Fig. 6: Comparison chart

Case Study 2:
In this case study, we took the class diagram of ECore metamodel given in the plugins directories of eclipse installation folder [19]. ECore metamodel consists of hundreds of elements in form of classes and association between these classes. However, for our test purposes we have modified the model for different test cases. For instance, Vb, V0, and V1 are the three versions, where Vb represents the base version, V0, V1 represents the derived versions. In base version Vb, we have the classes, EClass, EReference, EOperation, EAttribute, EParameter, and EDataType. EClass class has association to EReference, EOperation, and EAttribute classes. The name of the associations are eOperations, eReferences, eAttributes, and eIDAttribute. There are six operations in the EClass. EReference class contains three attributes and having associations to EClass and EAttribute. EAttribute class has one and association relationship to EDataType class. EDataType class has one attribute and no relationship. EOperation class contains two operations and one association to EParameter class. EParameter class has association to EOperation class. In derived version V0 we have the following classes. EClass, EReference, EOperation, EClassifier, EParameter, EDataType, EGenericType, and EFactory. EClass class has association to EReference, EOperation, and EClassifier classes. The name of the associations are eOperations, eAllReferences, and eType. There are four operations in the EClass. EReference class contains one attribute and having association to EClass. EClassifier class has one attribute and association to EGenericType class. EDataType class has one attribute and no relationship. EOperation class contains two operations and associations to EParameter and EDataType class. EParameter class has association to EOperation class. EGenericType class has association to EFactory class. EFactory class has one attribute and no relationship. EGenericType, and EFactory classes has been added and EAttribute class has been deleted in second version. Similarly, in V1, we have further modified the base version Vb. Figure 7 and Figure 8 show the results of execution time taken in these experiments. The comparison chart for second experiment is given in Figure 9 which shows the execution time comparison for all the tests performed.

In case A we took the three versions Vb, V0 and V1 of the model of relatively small size. The number of elements was 74 and the number of classes was 19 in three versions. When we executed our proposed merge approach on the given versions, we got the following results given in Fig. 7. The execution time taken by our approach is 481ms. The results of test performed on EMF Compare are given in Fig. 8. The execution time taken by EMF Compare is 656ms.

In case B, we increased the number of elements of the three versions Vb, V2 and V3 as compared to case A, where Vb represents the base version and V2, V3 represent the derived version 2 and 3, respectively. The number of elements was 123 and the number of classes was 21 in three versions. The execution time taken by

our approach is 511ms. The execution time taken by EMF Compare is 674ms. The difference between the execution time in performing case A and case B was 30ms in our approach and 18ms for EMF Compare.

In case C, the number of elements was 139 and the number of classes was 39 in three versions. When we executed our proposed merge approach on the given versions, the execution time is 535m. The execution time taken by EMF Compare is 710ms. The difference between the execution time in performing case B and case C was 24ms for our approach while 36ms for EMF Compare.

In case D, the number of elements was 205 and the number of classes was 56 in three versions. The execution time taken by our approach was 575ms. The execution time taken by EMF Compare was 788ms to perform the merge operation for the given input. The difference between the execution time in performing case C and case D was 40ms ~~in~~ for our approach while 78ms for EMF Compare.

In case E, the number of elements was 275 and the number of classes was 75 in three versions. The execution time taken by our approach is 587ms and 857ms for EMF Compare. The difference between the execution time in performing case D and case E was 12ms for our approach and 69ms for EMF Compare.

In case F, the number of elements was 332 and the number of classes was 87 in three versions. When we executed our proposed merge approach on the given versions, the execution time taken by our approach is 617ms and 906ms for EMF Compare. The difference between the execution time in performing case E and case F was 30ms for our approach while 49ms for EMF Compare.

Cases	Model Versions	Elements	Classes	Exe.Time (ms)	Time Diff per Case
A.	Vb, V0, V1	74	17	481	-
B.	Vb, V2, V3	123	21	511	30
C.	Vb, V4 ,V5	139	39	535	24
D.	Vb, V5, V6	205	56	575	40
E.	Vb, V6, V7	275	75	587	12
F.	Vb, V7, V8	332	87	617	30

Fig. 7: Test results of our approach for case study 2

Cases	Model Versions	Elements	Classes	Exe.Time (ms)	Time Diff per case
A.	Vb, V0, V1	74	17	656	-
B.	Vb, V2 , V3	123	21	674	18
C.	Vb, V4 , V5	139	39	710	36
D.	Vb, V5, V6	205	56	788	78
E.	Vb, V6, V7	275	75	857	69
F.	Vb, V7, V8	332	87	906	49

Fig. 8: Test results of EMFComp approach for case study 2

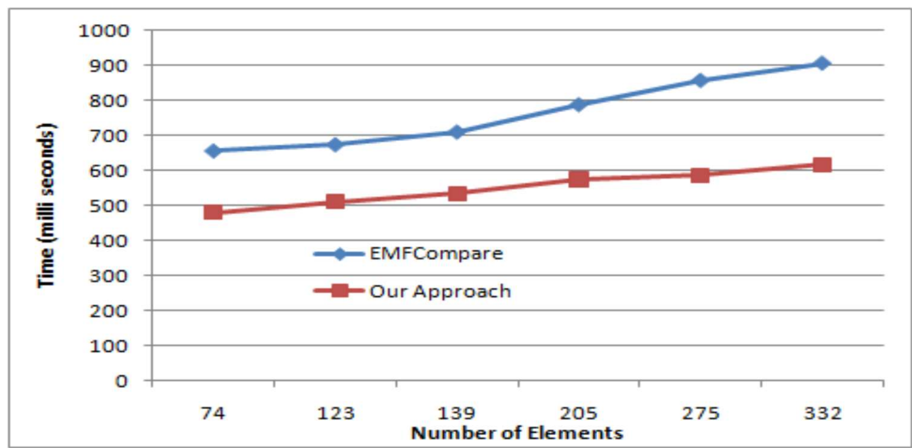

Fig. 9: Comparison chart for case study 2

6.0 CONCLUSION

Model versioning, differencing, merging are relatively young but key research areas in MDE. In this paper we presented a fine-granular model merge solution for model-based version control system in MDE. We represented models at fine-grained level as graph structures, which is an intermediate representation based on graph theory. Our approach followed a 3-way model merge process, where a base and its derived versions are used for comparison. To differentiate between conflicted and non-conflicted cases we defined different merge cases. A merge policy was established based on merge cases. Merge cases were used along with the comparison result in order to perform conflict resolution and merge operation. We performed a controlled experiment using open source eclipse modeling framework and compare our approach with an open source tool EMF Compare. The results proved the accuracy and efficiency of our proposed approach.

There are several future directions that we can consider. On one hand, this generality gives the advantage that different kinds of domain specific models can be compared with each other but on the other hand, as a consequence, diff and merge results are very generic too, only primitive changes, such as add or delete are recognized. Similarly, in model diff and merge activities, an appropriate visualization of differences between two versions is important for understanding the differences. In our approach, we used annotations to highlight the differences, while an alternative solution is the use of different colors. Different colors can be used to distinguish between different parts. However, doing so, one has to consider the editor tool dependency issue. In future we will work on these issues.

REFERENCES

[1] M. Pilato. "Version Control with Subversion". *O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004. ISBN 0596004486.*

[2] Cvs project. cvs web site. http://www.nongnu.org/cvs.

[3] D. Ohst, M. Welle. U. Keller "Merging UML documents." *Technical report, Universitt Siegen, 2004.*

[4] D. Ohst. " A fine-grained version and configuration model in analysis and design". *In ICSM'02: Proceedings of the International Conference on Software Maintenance (ICSM'02), page 521, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1819-2.*

[5] D. Ohst, M. Welle, U. Kelter. "Differences between versions of uml diagrams". *In ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, PP. 227–*

236, New York, NY, USA, 2003. ACM. ISBN 1-58113-743-5. doi: http://doi.acm.org/10.1145/940071.940102.

[6] Eclipse.org: Eclipse modeling framework (emf), http://www.eclipse.org/modeling/emf/.

[7] Eclipse foundation, emf compare, 2008. In http://www.eclipse.org/modeling/emft/?project=compare#comp

[8] A. Mehra, J. Grundy, J. Hosking. "A generic approach to supporting diagram differencing and merging for collaborative design*". In ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, PP. 204–213, New York, NY, USA, 2005. ACM. ISBN 1-59593-993-4. doi: http://doi.acm.org/10.1145/1101908.1101940.*

[9] A. Qazi, K. B. S. Syed, R. G. Raj, E. Cambria, M. Tahir, D. Alghazzawi, "A concept-level approach to the analysis of online review helpfulness", *Computers in Human Behavior*, Vol. 58, May 2016, PP. 75-81, ISSN 0747-5632, http://dx.doi.org/10.1016/j.chb.2015.12.028.

[10] C. Schneider, A. Zndorf, J. Niere. "CoObRA – a small step for development tools to collaborative environments*". In Workshop on Directions in Software Engineering Environments in 26th international conference on software engineering, Edinburgh, Scotland, UK, May 2004. URL http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/SZN04.pdf.*

[11] H. Oliveira, L. Murta, C. Werner. " Odyssey-vcs: a flexible version control system for uml model elements". *In SCM '05: Proceedings of the 12th international workshop on Software configuration management, PP. 1–16, New York, NY, USA, 2005. ACM. ISBN 1-59593-310-7. doi: http://doi.acm.org/10.1145/1109128.1109129.*

[12] M. Kogel. " Time – tracking intra- and inter-model evolution*". In Software Engineering (Workshops), PP. 157–164, 2008.*

[13] M. Kogel. "Towards software configuration management for unified models*". In CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models, PP. 19–24, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-045-6. doi: http://doi.acm.org/10.1145/1370152.1370158.*

[14] J. Helming, M. Koegel. Unicase. http://unicase.org.

[15] P. Brosch, P. Langer, M. Seidl, M. Wimmer. 2009. Towards end-user adaptable model versioning: The By-Example Operation Recorder. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models* (CVSM '09). IEEE Computer Society, Washington, DC, USA, 55-60. DOI=10.1109/CVSM.2009.5071723 http://dx.doi.org/10.1109/CVSM.2009.5071723

[16] W. Mehmood, N. Shah, Z. ur-din, E. U. Munir, "Fine-Granular Model Diff Solution in Model-based Version Control Systems", *Malaysian Journal of Computer Science, Vol. 28, No. 2, June 2015.*

[17] M. Alanen, I. Porres, "Difference and union of models," In *Proceedings of the UML Conference, Springer-Verlag LNCS 2863, San Francisco, California*, PP. 2–17, Oct.2003.

[18] E. Xing, Zhenchang, Stroulia, "Umldiff:An algorithm for object-oriented design differencing," *In Proc, IEEE/ACM International Con-ference on Automated* Software *Engineering(ASE'05), Nov.2005, Long Beach, California, USA, ACM*, pp 54–65.

[19] U. Kelter, J. Wehren, J. Niere, "A generic difference algorithm for uml models," *In Peter Liggesmeyer, Klaus Pohl, Michael Goedicke, editors, Software Engineering, volume 64 of LNI*, pp 105–116, GI, 2005, ISBN 3-88579-393-8.

[20] E. H. Berso, V. D. Henderson, S. G. Siegel. Software configuration management. Software configuration management: a tutorial. *IEEE Computer*, Vol. 12 No. 1, January 1979, pp. 6-14.

[21] R. Conradi, B. Westfechtel, "Version models for software configuration management". *ACM Comput. Surv.* Vol. 30 No. 2, June 1998, pp. 232-282.

[22] M. A. Shayegan, S. Aghabozorgi, R. G. Raj, "A Novel Two-Stage Spectrum-Based Approach for Dimensionality Reduction: A Case Study on the Recognition of Handwritten Numerals," *Journal of Applied Mathematics*, vol. 2014, Article ID 654787, 14 pages, 2014. doi:10.1155/2014/654787.

[23] M. Brambilla, J. Cabot, M. Wimmer, "Model-Driven Software Engineering in Practice" Morgan & Claypool, 2012.

[24] S. Foertsch, B. Westfechtel. Differencing and merging of software diagrams - state of the art and challenges. In ICSOFT (SE), PP. 90-99, 2007.

[25] D. S. Kolovos, D. D. Ruscio, A. Pierantonio, R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In CVSM '09: Proceedings of the 2009 ICSE Work-shop on Comparison and Versioning of Software Models, PP. 1-6, Washington,DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3714-6. doi:http://dx.doi.org/10.1109/CVSM.2009.5071714.

[26] Y. Lin, J. Gray, F. Jouault. Dsmdiff: A differentiation tool for domain-specific models. In European Journal of Information Systems (Special Issue on Model-Driven Systems Development), vol. 16, no. 4, PP. 349-361, 2007.

[27] A. Qazi, R. G. Raj, M. Tahir, M. Waheed, S. U. R. Khan, and A. Abraham, "A Preliminary Investigation of User Perception and Behavioral Intention for Different Review Types: Customers and Designers Perspective," *The Scientific World Journal*, vol. 2014, Article ID 872929, 8 pages, 2014. doi:10.1155/2014/872929.

[28] L. B. Huang, V. Balakrishnan, R.G. Raj, "Improving the relevancy of document search using the multi-term adjacency keyword-order model." *Malaysian Journal of Computer Science*, Vol. 25, No. 1, 2012, pp. 1-10.

[29] MagicDraw http://www.nomagic.com/products/magicdraw.html , last visited February 2016.

[30] MS Visio https://products.office.com/en-us/visio/, last visited February 2016.

[31] Eclipse Modeling Framework (EMF) [Online] Available: http://www.eclipse.org/modeling/emf/.