Research Journal of Applied Sciences, Engineering and Technology 8(4): 471-480, 2014
DOI:10.19026/rjaset.8.995
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: January 24, 2014 Accepted: March 08, 2014 Published: July 25, 2014

Research Article

Conflict Detection and Merging in Model based SCM Systems

Waqar Mehmood and Arshad Ali
COMSATS Institute of Informaion Technology, Wah Campus, Quaid Avenue Wah Cantt, Pakistan

Abstract: This study presents a fine-grained approach to the problem of conflict detection and merging in model-based Software Configuration Management (SCM) systems. Traditional SCM systems uses textual or structured data to represent models at fine-grained level. Our approach is based on defining graph structure to represent models data at fine-grained level. The approach is based on transforming the textual or structured data into graph structure and then performing the diff, merge and evolution control activities at the graph structure whereas versioning activities should remains at textual or structural representation. By doing so, at one hand we are getting the advantages of reusing the existing SCM systems for versioning purposes and on other hand avoiding the problems associated with textual or structured representation when performing rest of the SCM activities.

Keywords: Conflict detection, fine-granular representation, merging, model-based SCM, versioning

INTRODUCTION

Software Configuration Management deals with controlling the evolution of soft-ware systems. It is an indispensable part of a high-quality software development life cycle. Controlling the evolution requires many activities to perform such as construction and creation of versions, maintaining consistency between inter-dependent components, conflict detection and merging.

We categorize SCM systems into two areas i.e., Model-based SCM systems and Text-based SCM systems. Text-based SCM systems are traditional SCM systems that consider software artifact as a text files. By model-based SCM we means SCM system that consider software artifact as a graphical model. Funda-mentally, the main difference between text and model-based SCM occurs because of the different nature of their artifacts. Text-based SCM assumes an implicit tree structure with nodes being text files and with no relations. In contrast, in model-based SCM models are graphs, with nodes being complex entities and arcs (relations) containing a large part of model semantics. These dissimilarities clearly indicate that text and model-based SCM cannot be handled in the same way.

In this study the presented approach deals with conflict detection and merging activities in model-based SCM. At fine-grained level we represent our model as graph structure, which is an intermediate representation in the form of graph.

The approach is based on transforming the textual or structured data into graph structure. The diff, merge and evolution control activities are performed at the level of graph structure whereas versioning activities should remains at textual or structural representation such as XMI-files.

Model Driven Engineering (MDE) goal is to perform Software Engineering (SE) activities only on models, however, in reality models and files coexist and will have to be managed together consistently. This requires the reusability of traditional SCM systems for files. In our approach versioning activities should remains at textual or structural representation. By doing so, at one hand we are getting the advantages of reusing the traditional SCM systems for versioning purposes and on other hand avoiding the problems associated with textual or structured representation when performing rest of the SCM activities.

The approach present a three-way merge process, where a base and its de-rived versions are used for merging. The process of merging consists of comparison of version, conflict detection and resolution and merging. The comparison and merge operation are performed at fine-grained level on graph structure. The process of merging cannot be completely automated. Manual interaction is required in case of conflict detection. A conflict usually occurs if same element of an entity is modified in parallel. In order 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. We explain these concepts with the help of an example.

Corresponding Author: Waqar Mehmood, COMSATS Institute of Informaion Technology, Wah Campus, Quaid Avenue Wah Cantt, Pakistan

LITERATURE REVIEW

Odyssey-VCS (Oliveira *et al*., 2005) uses XMI as the protocol for communication between CASE tools and the VCS. When a conflict is detected, the developer receive conflict description and the original, user and current configurations. After performing the manual merge, the developer resubmit the UML model to the repository. Merge algorithm follows a 3-way merge approach which inputs are base version, source version and target version. Three main steps are existence analysis, attribute processing and relationship processing. The main problem is with per-forming Diff/Merge on structured data XMI which is not suitable for such kinds of operations as identified by Ohst and Kelter (2002).

The approach presented by Mehra *et al*. (2005) describe a generic approach for diff and merge via a set of plug-in components. Plug-ins are developed for Pounamu meta-CASE tool which support Version control, Visual differencing and Merging. Merging is realized interactively. Differences are shown graphically. The set of edit operations are offered to the user who decides which changes to apply. Diagram are transformed from XML representation into intermediate java object representation which represents a tree structure. Differences identified in two versions are converted into edit operations. The conversion can be considered as state-based to operation-based conversion. The approach is based on the reuse of existing SCM tools. However there are no inter/intra link information maintained between the elements of the models nor any evolution control policy is followed.

An approach for comparison and versioning of scientific workflows is presented in Ogasawara *et al*. (2009). The approach is based on modified 3-way merge algorithm named 3-way subgraph diff/merge algorithm which is based on graph theory. A 3-way subgraph diff/merge is a 3-way diff/merge in which instead of comparing a single vertex, a subgraph is analyzed as an atomic part and taken into consideration for merge decisions. The main problems with the approach are that it dealt only one specific kind of model i.e., workflows. Thus the approach is not generic. Furthermore, it doesn't reuse the existing SCM tools, which are helpful in case software documents consists of text files along with graphical models.

The approach of merging UML documents given in Ohst *et al*. (2004) split the merging process into three steps. First a pre-merged document is created, then identified conflicts are solved manually and finally merged document is created. The pre-merged document is an extended unified document consisting of common parts, automatically merged parts and conflicts. Software document is transformed into abstract syntax tree at fine-grained level.

In Schneider *et al*. (2004) all edit operations that are executed on the diagrams are logged by the tool. The approaches uses three way merging but gives priority to the version that was committed first. The approach is based on operation-based deltas and thus dependent of the editor tool which logged the edit operations.

MATERIALS AND METHODS

Below we first describe some basic terminologies.

Conflict: A conflict occurs when the same attribute of an entity is modified parallel in both versions, or an entity or its components are deleted in one version and is modified in other version.

Merge: The process of combining two or more versions into a consolidated version.

Types of merging: Two types of merging are Two-way merging and Three-way merging.

Two-way merge: Two way merge compares two versions and perform merging. Every difference requires a user interaction. An example of two way and three way merge is given in Fig. 1.

Three-way merge: Three way merge compare three versions, a base version and two derived versions. It is more powerful than two way merge since more conflicts can be detected and user interaction is required only the case of conflict. It also increases the level of automation.

Versioning approach: There are two types of versioning approach pessimistic approach and optimistic approach.

Pessimistic approach: Pessimistic approach a.k.a lock-modify approach al-lows one developer to work on a model at a time. This approach ensure that no conflict occurs in case many developer work on the same model, since no parallel work is allowed.

Optimistic approach: In optimistic approach many developer can modify the same model in parallel. A merge to the changes are performed when the models are checked-in.

Software documents: In software development life cycle two main types of software documents as shown in Fig. 2 are text files and graphical models. Text files may contains source code, documentation etc whereas graphical models are in form of UML or domain specific models. However, these models are usually

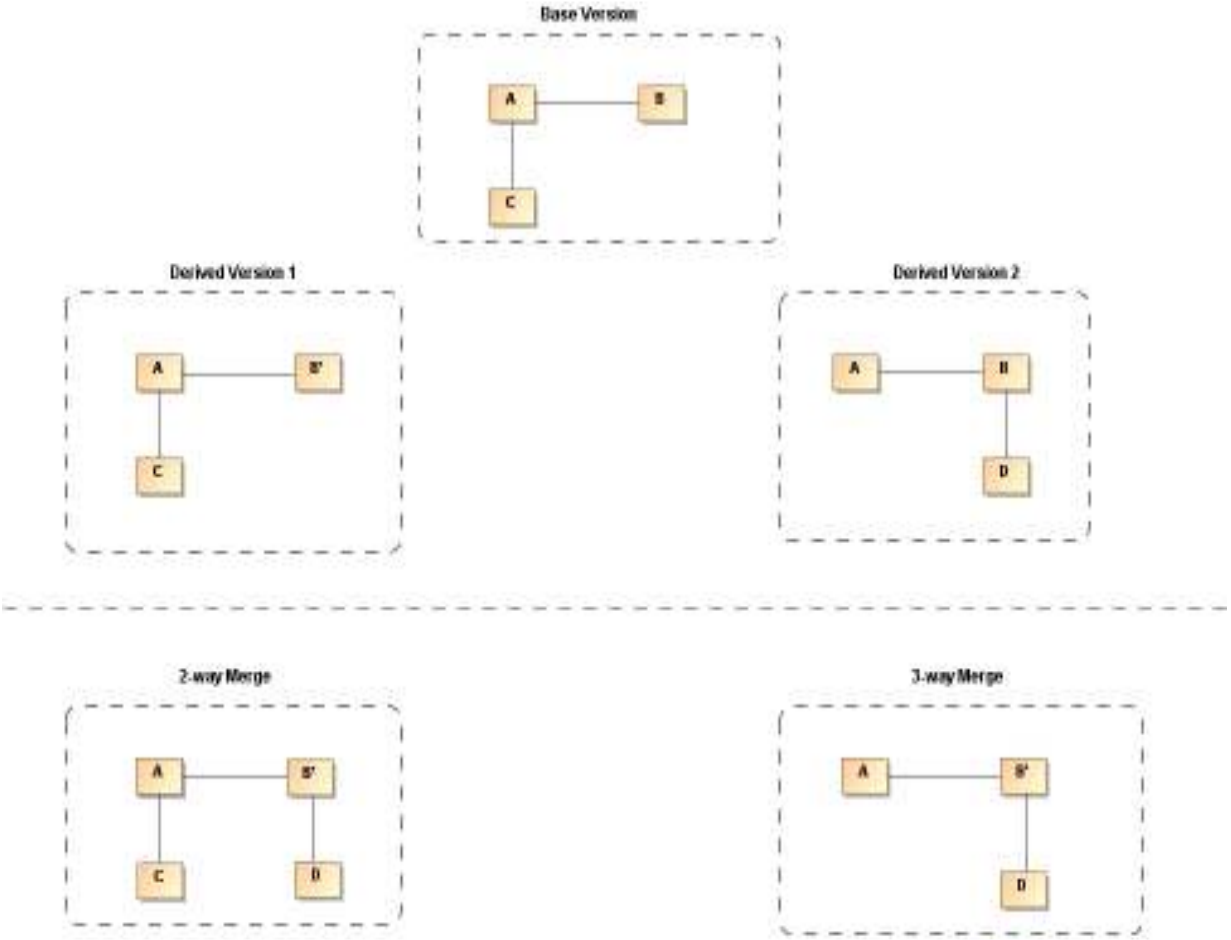

Fig. 1: 2-way and 3-way merge example

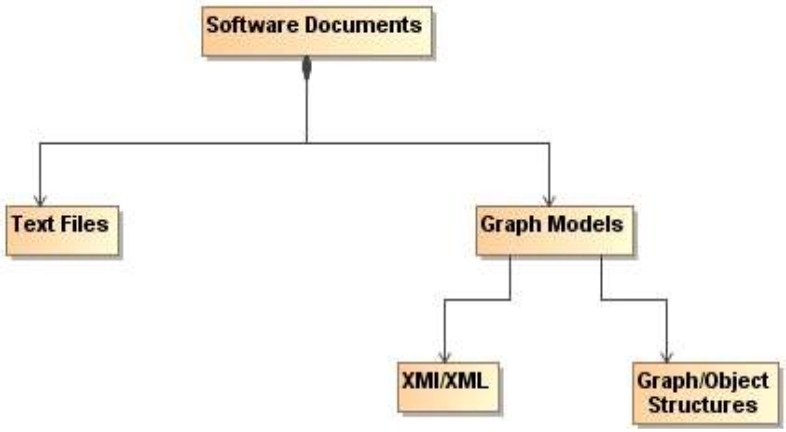

Fig. 2: Software document representations

represented in XMI at fine-grained level which is again a textual representation. The difference and merge operation performed on textual level yields many problem as identified in Ohst *et al*. (2003). Therefore we represent these models at fine-grained level in graph structure which is an intermediate representation in the form of graph. The meta model for this intermediate representation is given Fig. 3. By this representation we overcome the problems associated with textual representation of models.

Four main areas of SCM: Four main areas of SCM given in Fig. 4 are:

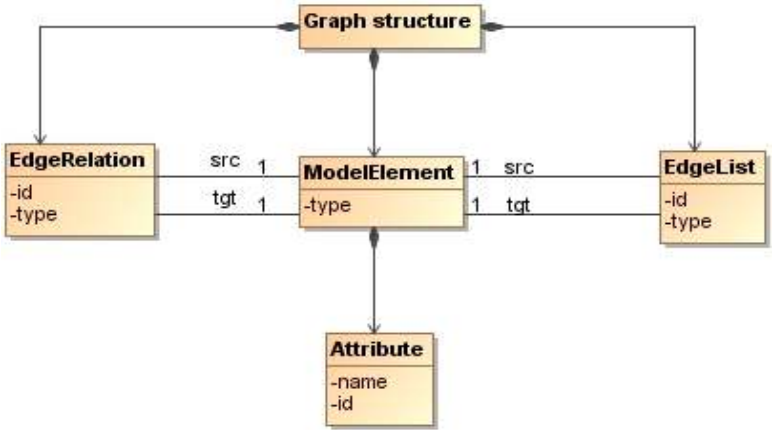

Fig. 3: Meta model of graph structure

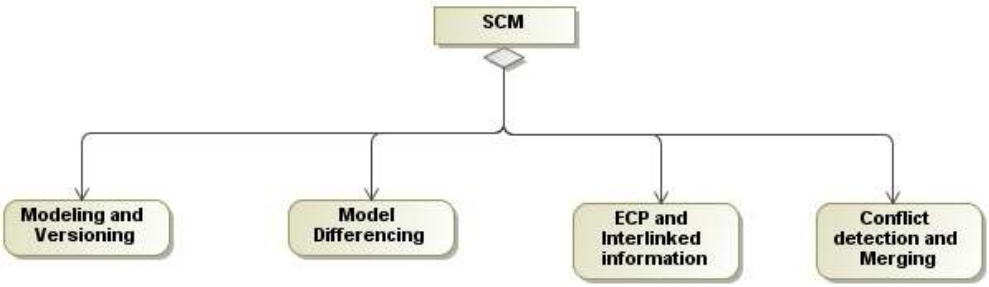

Fig. 4: Four main areas of model based SCM

- Modeling and versioning
- Model differencing
- Evolution control policy
- Conflict detection and merging

Modeling and versioning: Deals with creating and organizing versions of models, developing meta models version space and product space, defining approach pessimistic vs. optimistic.

Model differencing: Deals with comparing two versions to detect matches, differences, defining fine-grained data model. Approaches are state based and operation based.

Evolution control policy: Deals with defining a policy for creating a new version, defining versioning granularity, defining intralink and interlink information. Evolution control policy based on version granularity and intra and inter-link information.

Conflict detection and merging: Conflict detection and merging deals with identifying and resolving conflict. In the next sections we will explain these issues in more details.

Conflict detection: A conflict occurs when the same attribute of an entity is modified parallel in both versions, or an entity or its components are deleted in one version and is modified in other version. Consider a conflicted scenario given in Fig. 5. Two users user 1 and user 2 perform a check-out operation to an entity Customer. The user 2 modifies the entity by refining the data type of attribute id from into string and adding an attribute name of type string. The user 2 then perform check-in operation and the Customer entity is updated in repository. At the same time user 1 also perform updation by adding a method setId (id) to the entity. Now when the user1 perform the operation check-in a conflict is raised, since entity Customer is updated in the repository and user 1 don't have this updated version of entity Customer. So user1 first check-out the updated version of the entity, check the conflicted attributes (in this case attribute id is conflicted attribute) and perform manual resolution.

Not every change to a model or entity causes a conflict e.g., adding methods or attributes to the same entity, changes two different entities, adding an entity, deleting an unmodified entity. The important point here is to note that higher the delta granularity higher will be the number of conflicts and vice versa. For instance, if the delta granularity is at class level then any change to the same class causes a conflict even different part of the class are modified, whereas, if the delta granularity is at attribute level then any change to the same attribute causes a conflict. No conflict will

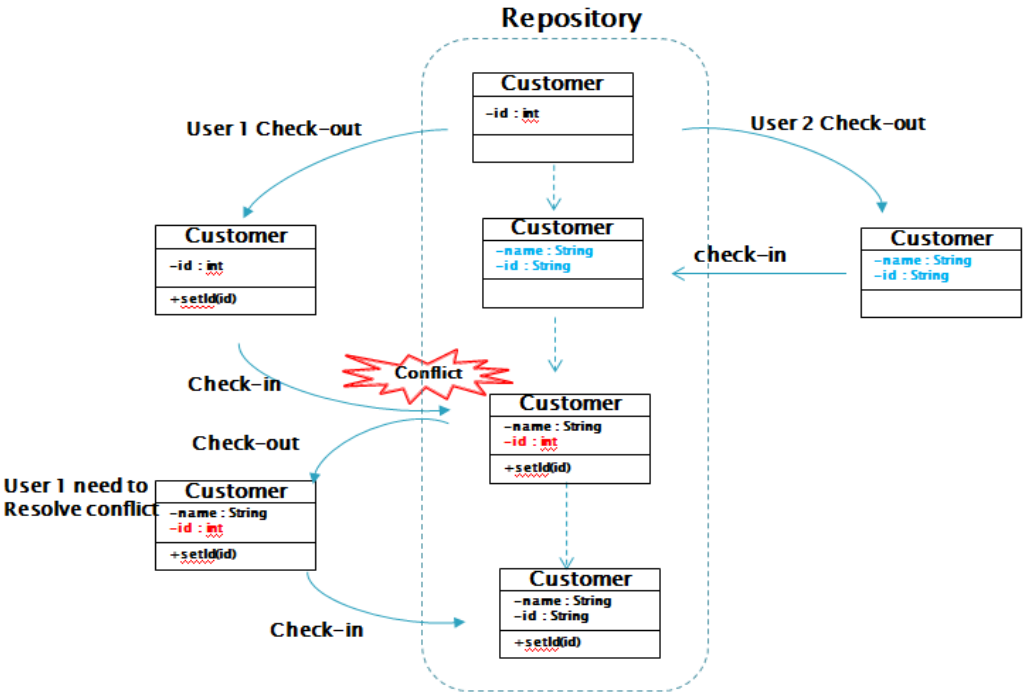

Fig. 5: Conflicted scenario

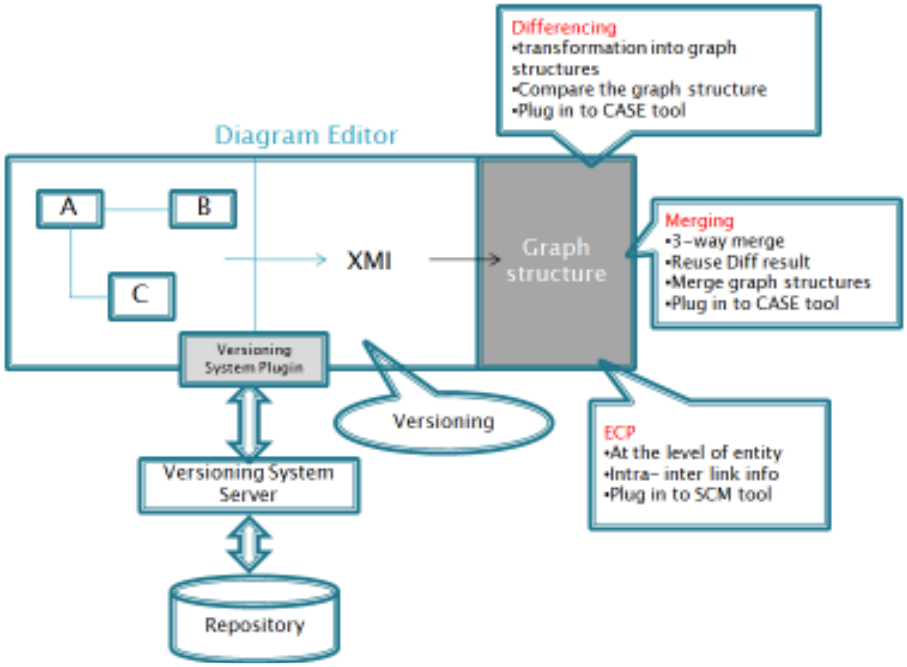

Fig. 6: Proposed solution

be raised if different attributes of the same class are modified.

RESULTS AND DISCUSSION

Merge process: Merge process consists of following four main steps.

Versions comparison: The process of comparing derived versions with the base version.

Conflict detection and resolution: The process of identifying the conflicted elements and resolving the conflicts either manually or automatically.

Merging: The process of combining two or more versions into a consolidated version.

Currently this problem is solved at the level of XMI along with the problems of versioning and difference calculation. A diagram editor is used to draw the graphical representation of the model which is stored as XMI format at fine-granular level. A plug-in of versioning system import/export these XMI data to the versioning system which perform versioning, differencing and merging operation on XMI. An extension to the current solution is given in Fig. 6. Our approach is based on transforming the XMI into graph structure and performing the differencing, merging and evolution control activities at the graph structure whereas versioning activities should remains at XMI representation. By doing so, at one hand we are getting the advantages of reusing the existing versioning system such as SVN (Michael, 2004) for versioning purposes and on other hand we overcome the problems associated with XMI when performing differencing, merging and evolution control activities.

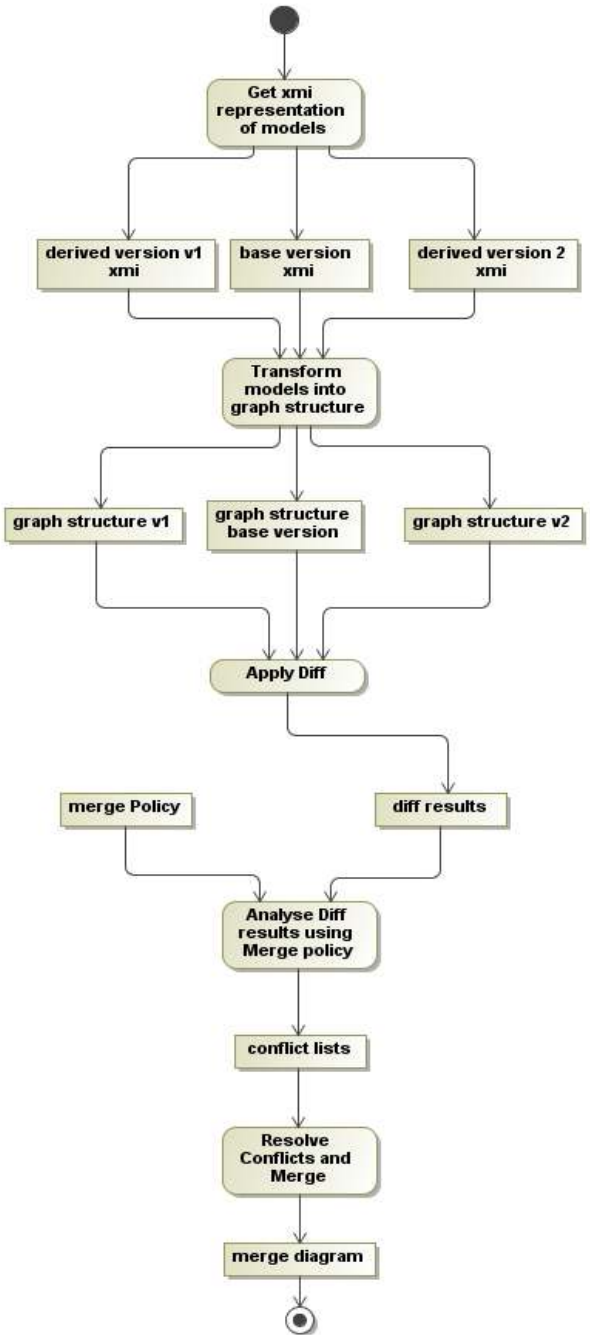

Fig. 7: Merging workflow

Merging workflow: Merging workflow is given in Fig. 7. The workflow works as follows. The diagram editor stores the diagram versions into xmi format. These xmi formats are inputs of the merging process. The first step is to transform these xmi inputs into graph structures. After transformation the Diff component compares the graph structures for matched, unmatched, added and deleted elements using a three-way merge approach. The result of this comparison will be analyzed ac-cording to the merge policy (Fig. 8). Based on difference result and merge policy the possible actions can be categorize into add, delete, include changed and include unchanged entities. The desired action will be performed. In case of conflict the conflicted elements will be identified. A manual interaction will be required to resolve the conflict. Finally merge diagram will be obtained.

Merge cases: We identified different merge cases (Table 1). Base version elements are com-pared with derived version elements. In case 1 the base element remains un-changed in derived versions. In case 2 base element is changed in both versions. In case 3 represent changed in one version while remains unchanged in second version. Case 4 represent changed in one version while deleted in other version. Case 5 represent base element deleted in one version while unchanged in other version. In case 6 element is deleted in both versions. Case 7 represent added in either version. Note that case 2 and case 4 are conflicted scenario, since same element is modified parallel in both versions. Based on these cases we apply our merge algorithm.

Merging algorithm: An abstract pseudo code of merge algorithm is given below:

- All the base version elements are taken into consideration. The corresponding element will be checked in both derived versions. If a match is found then the elements will analyzed according to the merge cases given in Table 1.
- If the base element is unchanged in both version then the unchanged element will be included into merge version.
- If the base element is changed in both version then the both the changed element will be included into merge version with the notification of conflict. Since this is a conflicted scenario, merge version will be manually updated to resolve the conflict.

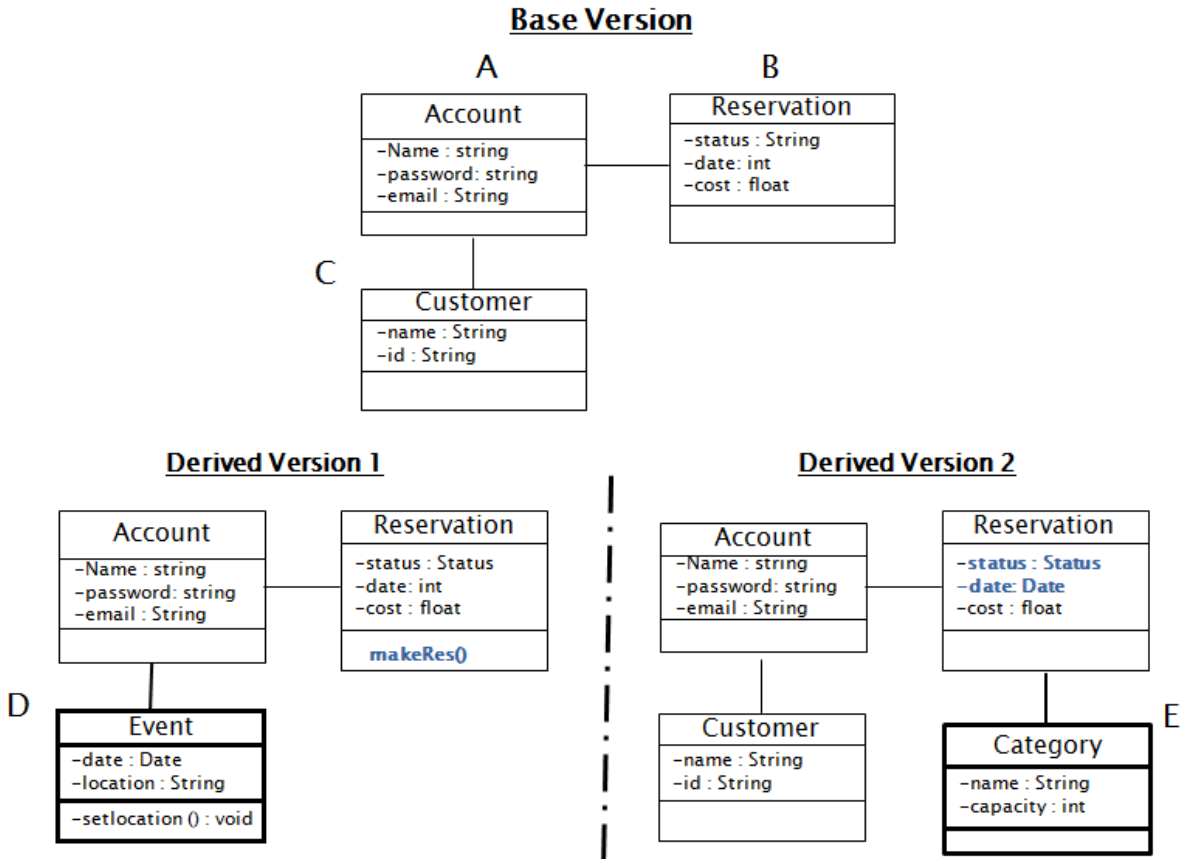

Fig. 8: Base and derived versions

Table 1: Merge cases

Cases	Base version vs. derived V1	Base version vs. derived V2	Action	Type
1	Unchanged	Unchanged	Include unchanged	
2	Changed	Changed	Include changed	Conflict
3	Changed	Unchanged	Include changed	
4	Changed	Deleted	Include changed	Conflict
5	Unchanged	Deleted	Delete	
6	Deleted	Delete	Delete	
7	Added	-	Add	

Table 2: Diff comparison results

Base version	Derived version 1	Derived version 2
A	Unchanged	Unchanged
B	Changed	Changed
C	Deleted	Unchanged
D	Added	-
E	-	Added
A-B	Unchanged	Unchanged
A-C	Deleted	Unchanged
A-D	Added	-
A-E	-	Added

- If the base element is unchanged in one version and changed in other version then the changed 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 with the notification of conflict. 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 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 present in either derived version but not in base version are considered added should be included merged version.

The same process will be repeated for relationships between entities.

Example: Consider the example given in Fig. 8, where a base version and two derived versions are given. In the base version we have three classes Account, Reservation and Customer. In derived version 1 Reservation entity is updated by adding makeRes () method, while entity Event is added and Customer entity is deleted. In derived version 2 Reservation entity is also updated by modifying the data types of attributes status and date, while entity Category is also added.

By comparing derived versions with the base version using a Diff algorithm and three-way merge approach we get the Diff result given in Table 2. After analyzing the result using the merge cases given in Table 1 and perform the merging we get the result given in Fig. 9. Note that entity Reservation is a conflicted entity since its updated in both derived versions, so user need to resolve the conflict manually.

Architecture: Figure 10 shows the reference architecture. There are six components namely, Model Editor, XMI/GS Converter, Merger, Diff comparator, Version Controller and Versioning System. The two repositories used in our approach are Policy repository and Version repository. Model Editor, Versioning System and Version repository are the reusable components of existing systems such as Magic draw and SVN (Michael, 2004) in our approach. XMI/GS Converter takes XMI files of diagrams developed in Model editor. It then transform XMI to graph structure and vice versa. The graph structures of different versions are inputs to the Diff comparator. The Diff comparator component perform differencing by comparing the graph structure and identifying the matched and unmatched elements. It will be a plugin to Model Editor. The output of Diff comparator i.e., the difference results will be input to both Merger and Version Controler component. Merge analyze

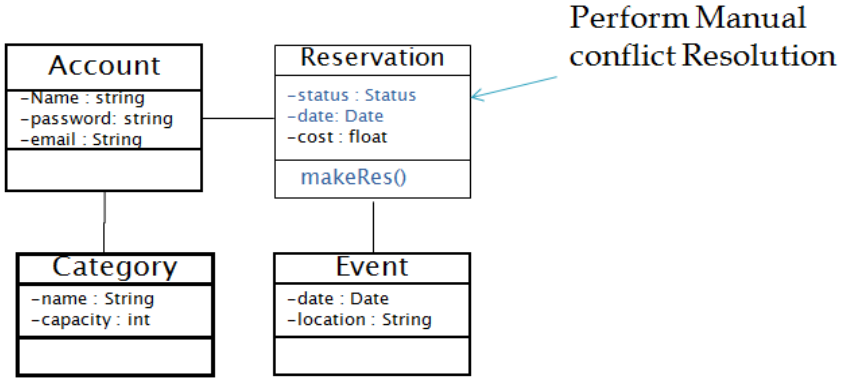

Fig. 9: Merge algo results

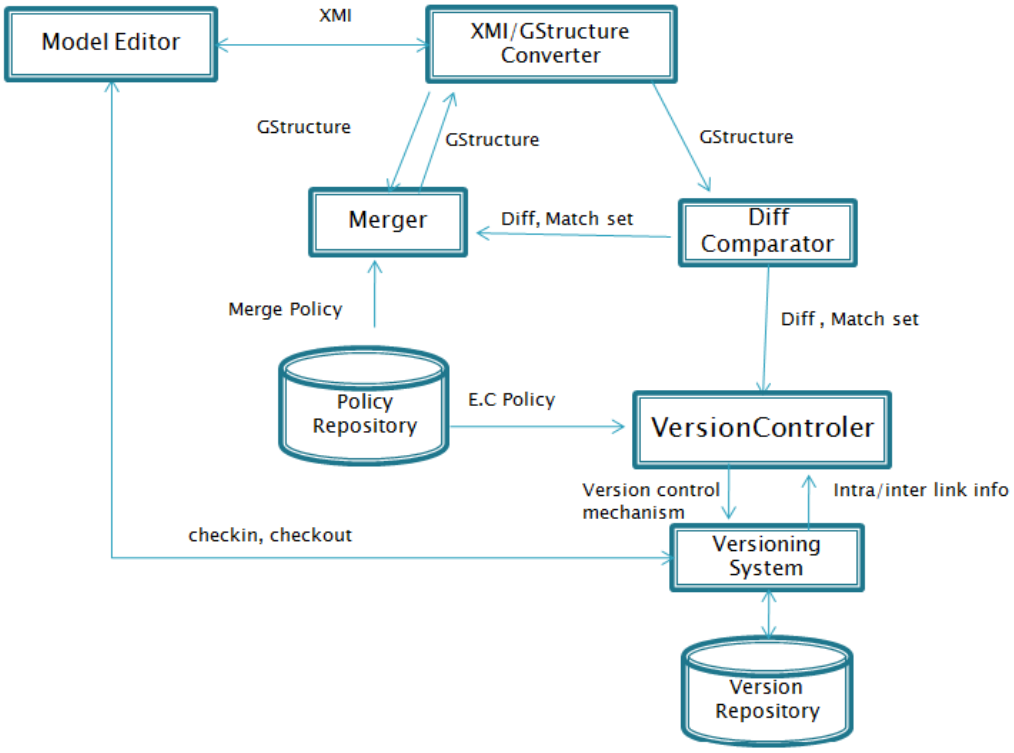

Fig. 10: Reference architecture

difference result based on the merge policy and perform a three-way merge. The merge result is in form of graph structure. This result is converted back into XMI by XMI/GS Converter and then the merge result can be rendered in Model Editor. Merge component will also be a plugin to Model Editor component. Finally evolution control mechanism will be implemented by Version Controler based on the inter/intra link information. Version Contorler component takes three kinds of inputs difference results, intra/inter link information and evolution control policy. Based on difference results and intra/inter links information Version Controler implements the evolution control policy. Version Controler component will be a plugin of versioning system since versioning is performed by the versioning system.

CONCLUSION

This study presents a fine-grained approach to the problem of conflict detection and merging in model-based Software Configuration Management (SCM) systems. Existing SCM systems uses textual or structured data to represent models at fine-grained level. Representing models as textual or structured data at fine grained level is not suitable for performing diff, merge and evolution control activities. In these representations changing the order of some text lines implies changing the file which produces a difference result for the same file when using traditional SCM systems. Secondly these files also contain layout information, which are not relevant for diff, merge etc activities of the model. Therefore our approach is based on defining graph structure to represent models data at fine-grained level. By doing so, at one hand we are getting the advantages of reusing the existing SCM systems for versioning purposes and on other hand avoiding the problems associated with textual or structured representation when performing rest of the SCM activities.

The presented approach is generic in a sense that it is neither dependent on any specific tool nor on any specific model type. Graph structure can be used to represent any kind of model data either domain specific or UML models. Similarly XMI/GS Converter can be generalized to convert any kind of textual data representing model data into graph structure. As a future work, we work on the prototype implementation of the proposed solution.

REFERENCES

Mehra, A., J. Grundy and J. Hosking, 2005. A generic approach to supporting diagram differencing and merging for collaborative design. Proceeding of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05), pp: 204-213.

Michael, P., 2004. Version Control with Subversion. O'Reilly and Associates, Inc., Sebastopol, CA, USA.

Ogasawara, E., P. Rangel, L. Murta, C. Werner and M. Mattoso, 2009. Comparison and versioning of scientific workflows. Proceeding of the ICSE Workshop on Comparison and Versioning of Software Models (CVSM '09), pp: 25-30.

Ohst, D. and U. Kelter, 2002. A fine-grained version and configuration model in analysis and design. Proceeding of the International Conference on Software Maintenance (ICSM'02), pp: 521-527.

Ohst, D., M. Welle and U. Kelter, 2003. Differences between versions of UML diagrams. Proceeding of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11), pp: 227-236.

Ohst, D., M. Welle and U. Kelter, 2004. Merging UML documents. Technical Report, Department of Electrical Engineering and Computer Science, University of Siegen, Germany, 2004. Internal Report.

Oliveira, H., L. Murta and C. Werner, 2005. Odyssey-VCS: A flexible version control system for UML model elements. Proceeding of the 12th International Workshop on Software Configuration Management (SCM'05), pp: 1-16.

Schneider, C., A. Zu¨ndorf and J. Niere, 2004. CoObRA-a small step for development tools to collaborative environments. Proceeding of 26th International Conference on Software Engineering Workshop on Directions in Software Engineering Environments. Edinburgh, Scotland, UK.