FINE-GRANULAR MODEL DIFF SOLUTION FOR MODEL-BASED SOFTWARE CONFIGURATION MANAGEMENT

Waqar Mehmood[1], Nadir Shah[2], Zahoo- ud-din[3], Ehsan Ullah Munir[4]

[1, 2, 3, 4] COMSATS Institute of Information Technology, Wah Campus, Pakistan

Email: [1]drwaqar@ciitwah.edu.pk, [2]nadirshah82@gmail.com, [3]zahooruddin79@gmail.com, [4]ehsanmunnir@gmail.com

Tel: 92-51-9314382, Fax: 92-514-546850

ABSTRACT

Software Configuration Management (SCM) aims to provide a controlling mechanism for the evolution of software artifacts created during software development process. Controlling the evolution requires many activities to perform, such as, construction and creation of versions, computation of mappings and differences between versions, combining of two or more versions and so on. Traditional SCM systems are file-based SCM systems. File-based SCM systems are not adequate for performing software configuration management activities because they consider software artifacts as a set of text files while today software development is model-driven and models are the main artifacts produced in the early phases of software life cycle. New challenges of model mappings, differencing, merging (combining two or more versions), and conflict detection (identifying conflicting changes by multiple users) arise while applying file-based solution to models. The goal of this work is to develop a configuration management solution for model representation, mappings and differences which overcomes the challenges faced by traditional SCM systems while model being the central artifact. Our solution is two-folded. First part deals with model representation. While traditional SCM systems represent models as textual files at fine-granular level, we represent models as graph structure at fine-granular level. In second part we are dealing with the issue of model diff, i.e., calculating the mappings and differences between two versions of a model. Since our model diff solution is based on our fine-granular model representation therefore we overcome not only the problem of textual representation of model but produce efficient results for model diff in terms of accuracy, execution time, tool independency and other evaluation parameters. 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 efficiency of our approach.

Keywords*: *model comparison, model difference, fine-granular model representation, model diff, model driven engineering

1.0 INTRODUCTION

To develop large software projects (in which more than one person participate), it essentially needs the efficient management of software artifacts created during software development. In the absence of efficient management, the software products that the industry has to produce can be delivered much later than scheduled, may cost more than anticipated and would have been poorly designed and documented [22]. Software Configuration Management (SCM) aims to provide an efficient control mechanism for such problems. It deals with controlling the evolution of software systems [24]. Controlling evolution requires many activities, 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) [23].

Model-driven engineering (MDE) is a modern software development technique that aims to reduce the complexity of the software development by assigning models a central role in the software development process [21]. With the advent of MDE, models become as a first-class artifact during the software development lifecycle. To ensure quality of models in MDE, models must be designed, analyzed, and maintained, subject to a version control mechanism. MDE emerges as a new paradigm that creates many challenges for the traditional configuration management systems. For instance, traditional VCS systems, such as Subversion [1] and Concurrent Versioning System (CVS) [2], have been used in the later phases of software development life cycle (e.g. during the implementation phase where the main artifact is the source code that is in the form of text files). However, such systems are not well suited for performing configuration management tasks on the models due to several reasons.

For instance, in MDE, software documents are not only text files, but also consist of diagrams such as different types of UML diagrams. These diagrams are often stored as XMI formats, such as a class diagram might be represented by a few lines of text in the file. 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. To a large extent, the order of text lines and their layout information is immaterial for model diff and merge operations on models [5-7]. Model diff deals with comparing the two versions of a model to detect the mappings and differences between them, while model merge deals with combining two or more versions of a model into a single one. Since traditional SCM systems are text-based systems and are not designed to operate adequately on models. Therefore, in this paper we first proposed a graph structure data model that handles the model structures adequately. Afterwards, we proposed a model diff solution that is built on top of the proposed graph structure data model and address the problem of computing the mappings and differences between the models. We implement our approach with an open source EMF [3] framework using Java as the source language. To benchmark our approach, we perform various tests and compared our approach with an open source tool EMF Compare [4]. We consider different quality parameters to compare our proposed approach with existing approaches. The experimental results prove the effectiveness of our approach.

The remainder of this paper is organized as follows. Section 2 explains the problem statement and background work. Section 3 focuses on the proposed model diff solution that deals with comparing the two versions of a model to identify mappings and differences between them. Section 4 presents our experiment design, results, and performance evaluation. Section 5 presents conclusion and future work.

2.0 PROBLEM STATEMENT AND BACKGROUND WORK

We classify existing SCM systems into two main categories a) file-based SCM systems, and b) model-based SCM systems.

2.1 File-based SCM Systems

Traditional SCM such as Subversion [1] and CVS [2] are file-based SCM systems. These systems consider software artifacts as a set of text files and have been designed to manage changes in textual artifacts, such as, source code in a file system. Consequently, they operate on the abstraction of file system and represent change in a line-oriented way. The underlying assumption **of** these systems in case of modification of a document is that one or few adjacent lines of the text are inserted, deleted or modified. Dirk Ohst et al. [5-7] identify several reasons under which these systems failed to work for document management in the early phases of software development life cycle. For instance, in MDE, software documents are not only text files, but also consist of diagrams such as, different types of UML diagrams. These diagrams are often stored as XMI formats, such as a class diagram might be represented by a few lines of text in the file. 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. Moreover, the position where a class symbol appears in the diagram is explicitly stored in layout data. To a large extent, the order of text lines and their layout is immaterial for diff and merge operations on models. Therefore, applying diff and merge operations at the level of plain text would hardly produce meaningful results. Another limitation of existing solutions is the lack in identification of shift operation in diagram modification. For instance, the shifting of a method from one class to another class corresponds to the shifting of a block of text between a file. These systems interpret this shift as a deletion of the method or block of text from one location and insertion at the second location. Visualization of detected differences between models is another problem which the traditional SCM systems are unable to handle. These dissimilarities clearly indicate that file-based and model-based SCM cannot be handled in the same way.

2.2 RELATED WORK ON MODEL-BASED SCM SYSTEMS

Many solutions to model-based SCM exist in literature. In this section we will describe the existing solutions. Alanen and Porres in [10] 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 for understanding. The approach also does not detect shifting of elements between models and detect shift operation as delete-add operation. Ohst et al. [6] address 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

which is used for model comparison. For showing the differences between two documents the unified document is used which contains the common and specific parts of both base documents; the specific parts are highlighted. EMF Compare [4] is an open source tool used EMF technology project 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 the difference 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. As compared to our approach the diff and match model produced by EMFCompare cannot be converted to graphical representation as done in our approach. Furthermore, EMF Compare also suffers from the sensitivity issue of layout or order changes discussed in Section 2.1. A detailed empirical comparison of our approach with EMF Compare is already given in Section 4 which shows the performance efficiency of our approach. Xing et al. [11] presented an automated UML-aware structural-differencing algorithm, UMLDiff . UMLDiff is an algorithm 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, reverse engineered from two corresponding code versions. The approach uses a language-based matching criterion and identifies corresponding entities based on their name and structure similarity. If two objects have 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. [12] presented a generic algorithm SiDiff which uses an internal data model comparable with a 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 the elements they consists of, the difference algorithm starts with a bottom-up traversal at the leaves of the composition tree. The approach uses a signature-based matching criterion. The Pounamu approaches presented in [13] 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 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 ID (uid)-based matching criteria. Also the approach lacks detection of the shifts operation.

Approaches	Calculation Method		Calculation Criteria			Indpendency		Shift Detection
	State-based	Operation-based	Uid-based	Signature-based	Language-based	Tool	Diagram	
Alanen et al.	√	×	√	×	×	√	√	×
DSMDiff	√	×	×	√	×	√	√	×
D.Ohst et al.	√	×	√	×	×	×	×	√
SiDiff	√	×	×	√	×	√	√	√
UMLDiff	√	×	×	×	√	√	×	×
Pounamu	×	√	√	×	×	×	√	×
Girschick	×	√	√	×	×	×	×	×
Our Approach	√	×	√	√	×	√	√	√

Legends: √ Supported × Not supported

Fig. 1: Comparison with existing Approaches

Lin et al. [18] presents an approach which is based on domain specific modeling. The approach presents a meta-model-independent algorithm for model differentiation in the context of domain specific modeling (DSM). The approach addresses the problem of computing the differences between domain-specific models by exploring the issues of: what are the essential characteristics of domain-specific models, and how are they defined, what information within domain-specific models needs to be compared, and what information is needed to support meta-model-independent model comparison, etc. Lamine et al. [19] discusses the issue of uncertain version control open collaborative editing of tree-structured documents. They uses a probabilistic XML model as a basic component of

our version control framework. Each version of a shared document is represented by an XML tree and the whole document, together with its different versions, is modeled as a probabilistic XML document. They showed that standard version control operations can be implemented directly as operations on the probabilistic XML model; efficiency with respect to deterministic version control systems is demonstrated on real-world datasets. In contrast to text-based versioning systems, the need for model-based versioning systems as graph structures is also realized by Taentzer, et al. [20]. They present an approach that takes model structures and their changes over time into account. Considering model structures as graphs, we define a fundamental approach where model revisions are considered as graph modifications consisting of delete and insert actions. The fundamental concepts were illustrated by versioning scenarios for simplified state charts. Furthermore, they showed an implementation of this fundamental approach to model versioning based on the Eclipse Modeling Framework as technical space. In [8] Girschick presented an approach for difference detection and visualization of UML class diagrams. The modification applied to a class diagram can be described using basic transformation operations such as add, delete, rename etc. In contrast to our approach, the stated approach is also an operation-based approach for model comparison thus dependent on the editor tool. The approach uses a uid-based matching criteria while in our approach we use a hybrid criteria. The approach lacks detection of the shifts operation as well. Based on the above discussion a comparison chart for our approach and rest of the approaches is given in Fig. 1.

3.0 PROPOSED MODEL DIFF SOLUTION

In this section, we address the issues pertaining to model diff. Model diff deals with comparing two versions of a model to compute the differences and mappings between two versions. It is an important and challenging task in the MDE and is required in many model management activities, such as model-based version control, model consistency, model merging, and transformation testing [14]. We address the problem of computing the mappings and differences between the models by exploring the issues of:

i. how to represent models at a fine-grained level,
ii. how to compute deltas, namely the state-based or operation-based approaches, and
iii. designing algorithms that can be used to discover the mappings and differences between the models.

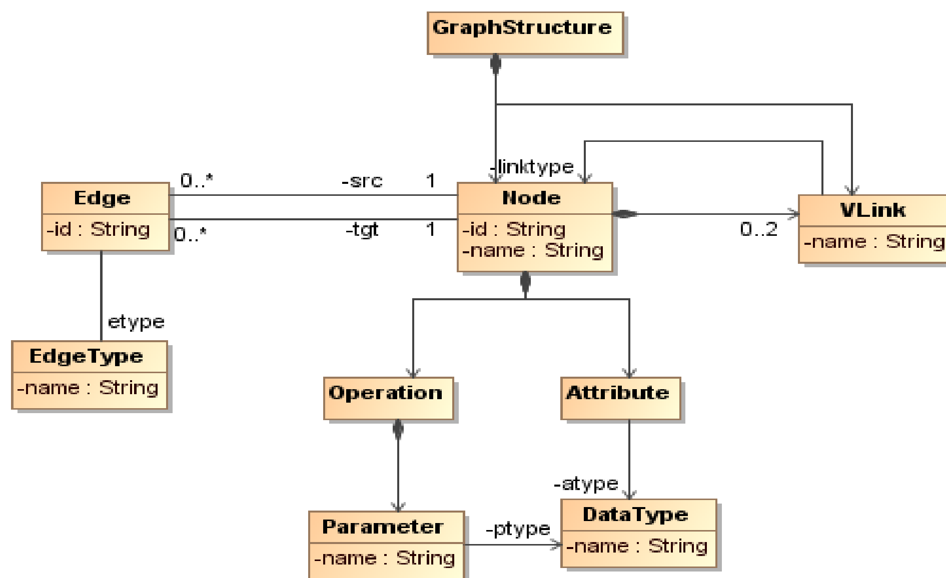

Fig. 2. Graph structure data model

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. 2). The proposed structure represents graph with typed elements that can be decorated with attributes. The basic elements of the meta model are: Nodes, Edges, Links, Operations, Attributes, Parameters, and DataTypes. Besides other advantages, one more important benefit of the meta model 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].

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. Regarding delta computation, 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. Deltas are reconstructed using a "differencing" algorithm that compares the different state representations. 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. When these operations are applied to one version v1, it yields another version v2. 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). Due to this reason we too adapt a state-based procedure in our proposed approach, as it provides generality and independency of the tools.

3.3 MODEL DIFF COMPARISON ALGORITHMS

Algorithm 1.0 shows the proposed diff algorithm for comparing models for MapSet and ChangeSet. 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 consist of node IDs, attributes, and operations; whereas, for the structural properties of the nodes, the algorithm compares the edge-signature of the nodes. The comparison algorithm works as follows: First, the header nodes of both versions of the graph structures are accessed (lines 1 − 2). Thereafter, all of the flag variables used in the algorithm are set to false (line 3). Afterwards, the nodes of the graph structures are compared. For this, each node in the vlink of the first version of the graph structure V1_GS, the nodes in the vlink of second version graph structure V2_GS are traversed to find similar nodes in V2_GS (line 4 − 46). The node IDs in both of the versions are compared in line 6. If a match of similar ID for node 'n' of V1_GS and node 'm' of V2_GS exists, then the nodes attributes is compared. To compare a node's attributes traverse all of the attributes of node 'n' and node 'm' and compare with each other. If an attribute of 'n' does not match any of the attribute of 'm', then the corresponding unmatched attribute of 'n' will be marked as the deleted attribute and set the flagA1 value to true (lines 7 − 13). Similarly, compare node m's attributes with node n's attributes to check new attributes. For this, traverse all of the attributes of node 'm' of V2_GS and node 'n' of V1_GS, and compare the attributes of 'n' and 'm'. If an attribute of 'm' does not match to any attribute of 'n', then the corresponding unmatched attribute of ' m' will be marked as an added attribute and set the flagA2 value to true (lines 14 − 20). Similarly, the algorithm compares the operations of the nodes of both versions (lines 21-34).

Algorithm 1.0 compareNodes()

Require: Graphstructures of Model Version 1 (*V1_ GS*) and Model version 2 (*V2_ GS*)
1: input: *V1_GS* header node;
2: input: *V2_GS* header node;
3: *flagA1=flagA2=flagO1=flagO2=flagE=flagD* ← false
4: ∀ node '*n*' in *V1_GS* vlink traverse V*1_GS* do

```
5:     ∀ node ‘ m’ in V2_GS vlink traverse_ V2_GS do
6:        if n.id ≡ m.id then
7:           ∀ n.attributes do
8:              compare n.attributes with m.attributes
9:              if any attribute of ‘n’ and ‘ m’ do not match then
10:                mark n.attribute as deleted attribute
11:                flagA1←true
12:             end if
13:          end for
14:          ∀ m.attributes do
15:             compare m.attributes with n.attributes
16:             if any attribute of ‘m’ and ‘ n’ do not match then
17:                mark m.attribute as added attribute
18:                flagA2←true
19:             end if
20:          end for
21:          ∀ n.operations do
22:             compare n.operations with m.operations
23:             if any operation of ‘n’ and ‘ m’ do not match then
24:                mark n.operation as deleted operation
25:                flagO1←true
26:             end if
27:          end for
28:          ∀ m.operations do
29:             compare m.operations with n.operations
30:             if any operation of ‘m’ and ‘ n’ do not match then
31:                mark m.operation as added operation
32:                flagO2←true
33:             end if
34:          end for
35:          ∀ m.edges do
36:             compare m.edge with n.edge
37:             if m.edgeType ≡ n.edgeType and m.edge = n.edge then
38:                set edge changed to m.edge from n.edge
39:             end if
40:             if any edge of ‘m’ and ‘ n’ do not match then
41:                set m.edge as added edge
42:                flagE←true
43:             end if
44:          end for
45:          if m.eSuperType= n.eSuperType then
46:             if n.eSuperType≡ null then
47:                set m.eSuperType as added super type
48:             end if
49:             if m.eSuperType≡ null then
50:                set n.eSuperType as deleted super type
51:             end if
52:             if m.eSuperType changed then
53:              set m.eSuperType modIfied from n.eSuperType
54:             end if
55:          end if
56:          if flagA1 ≡ true or flagA2 ≡ true or flagO1 ≡ true or flagO2 ≡ true or flagE ≡ true
             then
57:             add ‘ m’ to ChangeSet{}
58:          end if
59:       if flagA1 ≡ true and flagA2 ≡ true and flagO1 ≡ true and flagO2 ≡ true and flagE ≡ true then
60:             add ‘ m’ to MapSet{}
61:          end if
62:          flagD←true
```

```
63:            end if
64:        end for
65:        call checkDeleteNode&Edges()
66:    end for
67:    call checkNewNode&Edges()
68:    call checkShiftNodes()
```

Algorithm 1.1 checkDeleteNodes&Edges()

```
1:  if flagD ≡ false then
2:        add 'n' to DeleteSet{}
3:        ∀ n.edges do
4:              set n.edge as delete edge
5:        end for
6:        ∀ V1_GS nodes do
7:             traverse V1_GS's nodes whose edgeType equal to node n
8:              set corresponding edge as deleted edge
9:        end for
10:       if n.eSuperType = null then
11:             set n.eSuperType as deleted super type
12:       end if
13: end if
```

Thereafter, the algorithm compares the edge-signature of the nodes. The Edge- signature consists of the edge ID and connected nodes of the edge. To compare the node's edge-signature, we traverse all of the edges of node 'm' of V2_GS and node 'n' of V1_GS, and compare the edges of 'n' and 'm'. Thereafter, to check the modified edge the edgeType of 'm' and 'n' will be compared. If the edgeType is the same but the edge IDs are different, then the edge will be identified as the modified edge. However, if any edge of 'm' does not match to any of the edge of 'n', then we mark the corresponding edge of 'm' as the new edge, and set flagE to true (lines35 − 44).

Algorithm 1.2 checkNewNodes&Edges()

```
1:  flagN←false
2:  ∀ node 'm' in V2_GS vlink traverse V2_GS do
3:        ∀ node 'n' in V1_GS vlink traverse V1_GS do
4:              if m.id ≡ n.id then
5:                    flagN←true
6:              end if
7:        end for
8:        if flagN ≡ false then
9:              add 'm' to NewSet{}
10:               ∀ m.edges do
11:                     mark m.edge as new edge
12:               end for
13:         end if
14:         if m.eSuperType = null then
15:               set m.eSuperType as added super type
16:         end if
17: end for
```

To check the inheritance relationship between the nodes, we compare the nodes eSuperType property. If node m.eSuperType is not equal to node n.eSuperType, then we check for three cases, first if node n.eSuperType is null then its means that a super type is added to node m, second if node m.eSuperType is null then its means that a super type is deleted to node m, and in third case a super type is modified from n.eSuperType to m.eSuperType (lines 45 − 55). Afterwards the values of the flag variables flagA1, flagA2, flagO1, flagO2, and flagE will be checked, if any of the flag variable value is true then node 'm' will be added to ChangeSet{}, and if all the flag variable values are true then node 'm' will be added to MapSet{} (lines 56−61). Algorithm 1.1 checkDeleteNodes&Edges() checks deleted nodes and their edge-signature. The algorithm first check if a node is deleted in the second version and then process the deleted edges.

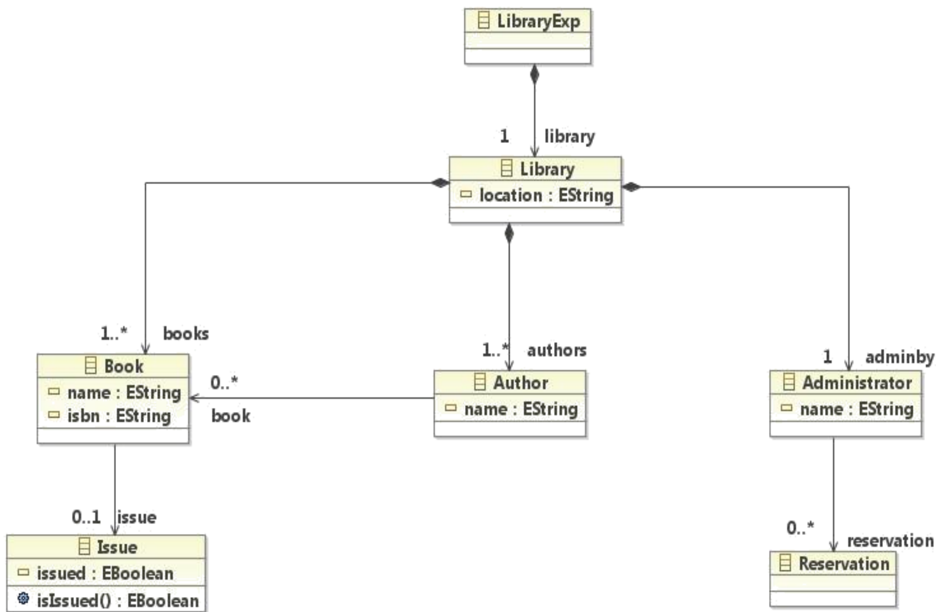

Fig. 3. Library system model version 1 (V1)

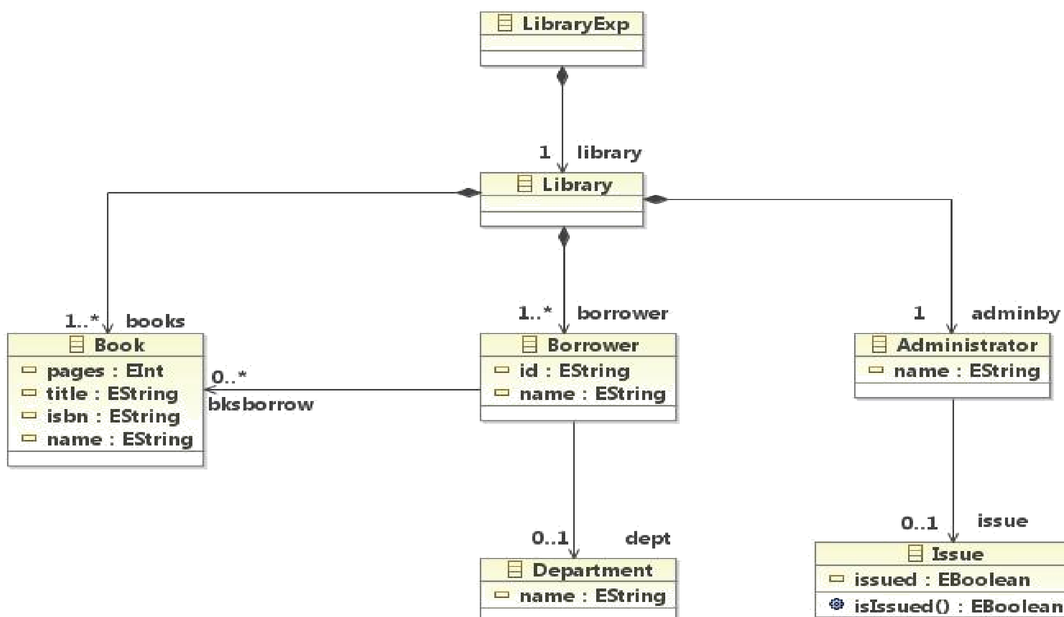

Fig. 4. Library system model version 2 (V2)

First the value of flag variable flagD is checked, if it is false then it means node n of V1_GS is deleted and will be added to DeleteSet{} (lines 1-2). Then all edges of node ‘n’ will be marked as delete edges (line 3-5). Then to identify those edges whose edgeType are the deleted nodes we traverse all nodes in V1_GS whose edgeType equal to deleted node ‘n’ and set the corresponding edge as deleted edge (lines 6-9). Then the super type of the deleted node will be checked, if the deleted node is a sub type of another node then the value n.eSuperType will be set as deleted super type (lines 10-13).

Algorithm 1.2 checkNewNodes&Edges() checks new nodes and their edge-signature in second version. First the value of variable flagN will be set to false (line 1). In order to identify new nodes in the second version of the model, each node ‘m’ in the vlink of the V2_GS, and node n in the vlink of V2_GS are traversed (lines 2-17). The nodes ids in both versions are compared, if a match of similar ids exist then we set flagN to true (lines 3-7). If no match exist, i.e., flagN is false then node m is added to NewSet{} and all its edges are marked as new edges (lines 8-13). Then the super type of the new node will be checked, if new node super type is not null then it will be set to new super type (lines 14-16). The whole process will be repeated until all nodes of the second version's graph structure will be traversed.

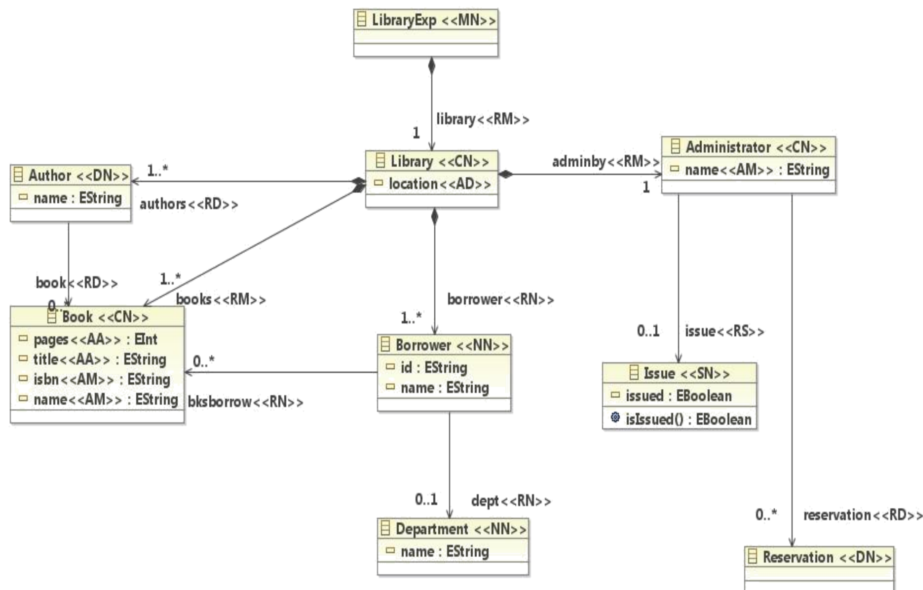

Fig. 5. Diff result of model version l (V1) and version 2 (V2)

4.0 PERFORMANCE EVALUATION AND EXPERIMENTAL RESULTS

4.1 Eclipse Modeling Framework

We implement our diff approach using the Eclipse Modeling Framework (EMF) [3, 17]. EMF is an open source project which provides modeling framework and code generation facility for building tools and other applications based on a structured data model. EMF is one of the most successful approaches to MDE. EMF provides good support for code generation, metadata querying, model serialization, and model editor. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. EMF is split into three large components: core, edit and codegen. Core contains the ECoremeta model, persistence, serialization, a model tracer (change notification and recording) and a validation framework. Edit includes a default model viewer and helps in building rich user interfaces (view, editor) for models. Codegen supplies a code generator

for ECore based models and several importers (for example from Rational Rose). In the context of this paper we are not interested in the code generation facility of EMF, while the core and edit components are used for model development, metadata querying and serialization purposes.

4.2 Empirical Performance Evaluation

To benchmark our approach, we performed different test, and compared our approach with the open source tool EMF Compare [4]. The EMF Compare uses the Eclipse Modeling Framework (EMF) technology project to compare models. It is realized by a package of Eclipse plugins that overwrite the Eclipse's standard comparing behavior. We select the EMF Compare for comparison with our approach because it is the only available open source tool, and secondly, our approach also uses the EMF technology.

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	437	-	608	-
2	3	4	452	15	640	32
3	7	4	474	22	723	83
4	7	10	480	6	730	7
5	9	19	486	6	767	37
6	10	24	490	4	780	13

Fig. 6. Comparison results

The main assessment criteria of our evaluation are the quality of the calculated results and the required execution time. We performed a controlled experiment on a library system class model to benchmark the diff algorithm. As a running example we use two versions of a library system model given in Fig. 3 & 4. The class model of Library system consists of several classes to model the static structure of the system by showing the system's classes, their attributes, and the relationships between the classes. In version V1 we have the following classes, Library, Book, Author, and Administrator. Library class has containment association to Book, Author and Administrator classes. The names of the associations are books, authors, and admin by, respectively. Each Book, Author, and Administrator class has one attribute name. In version V2 we have the following classes, Library, Book, Administrator, and Reservation. Library class has containment relationships to Book, and Administrator classes and has attribute location. Book class has attribute isbn. Administrator class has association relationship to Reservation class and has attribute id. Reservation class has been added and Author class has been deleted in second version. The model diff result is shown in Fig. 5. We use the following annotations to clarify the diff results. Mapped Node <<MN>> is used to represent the map nodes in both versions. Changed Node <<CN>> is used to represent the nodes which are changed in second version. New Node <<NN>> is used to represent the nodes which are added in second version. Deleted Node <<DN>> is used to represent the nodes which are deleted in second version. Shifted Node <<SN>> is used to represent the nodes which are shifted in second version. Whereas, <<RM>>, <<RN>> , <<RC>> , <<RD>> , <<RS>> are used to represent the mapped, new, changed, deleted and shifted references in both versions.

We took six test scenarios. The test scenarios differed in the size of the elements of the models. The execution time reported is the average of hundred test runs. The tests were performed on a standard PC Intel Core Duo CPU P9400 with 4 GB memory. There were three main differences that we observed between our approach and the EMF Compare as follows. First, the EMF Compare computes the reordering in the layout of the models as a difference between the models. Reordering of the elements in a model is a layout change rather than the change in the model's semantics. This is same issue as we mention in Section 2.1. That is to say that the order of the sections of the text in the XMI file is irrelevant for model diff. Therefore, computing reordering as a difference is not desired. Not only this would reduce the accuracy of the approach, but also would reduce the efficiency of the approach as this would require some extra execution time. In the experiments performed, we observed that nearly in every case, the EMF Compare identified some reordering differences. For instance, the result of the reordering differences identified by the EMF Compare is given in Fig. 8 for experiment. Fig. 7 shows the comparison chart for reordering and unmarked

differences of the experiments. For all cases, except case 1, the EMF Compare identified the reordering difference that in fact is a layout change rather than a change in the model's semantics. In the experiment for the largest input, the total number of reordering differences was ten. Whereas, in our approach, we do not consider the reordering of elements as a difference. Therefore, in all of the cases we have no reordering difference computation. Consequently, we conclude that an increase in the number of differences of the order of XMI's elements reduces the accuracy and efficiency of the EMF Compare tool. However, in such cases our approach remains unaffected.

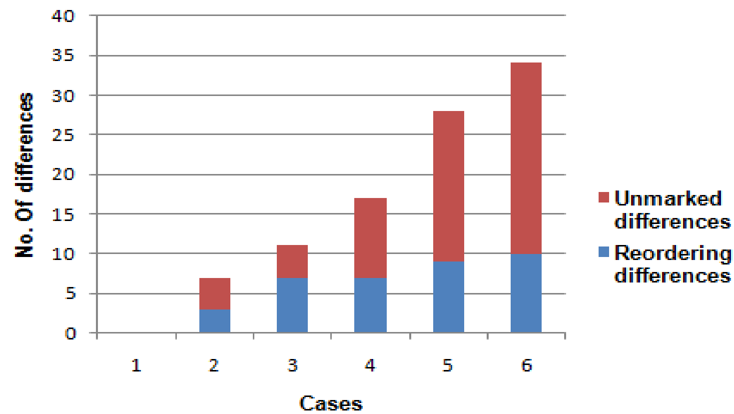

Fig. 7. EMF Compare reordering and unmarked differences

Second, by closely analyzing the output of the EMF Compare it was observed that our approach identified the differences on a more fine-grained level than the EMF Compare. This is because for the elements addition and deletion the EMF Compare mention changes only at the level of nodes. If a class/node is added or deleted in a new version then the EMF Compare only identifies the addition and deletion of the class and does not mention the changes of other features, such as references or attributes of the class at fine-grained level. For instance, for the results of unmarked differences by the EMF Compare given in Fig. 7, it was observed that for all of the cases in which there are classes added or deleted, the EMF Compare did not marked the differences at fine-grained level for added or deleted classes. Moreover, as we increase the size of the problem, the unmarked differences are also increased. For instance, for the largest input model the total number of unmarked differences are 24.

Finally, the difference between the execution time shows the efficiency of our approach in terms of speed, as compared to the EMF Compare. Fig. 8 shows the comparison chart for execution time of the experiments. During experiments we observed that the efficiency of our approach in term of speed, as compared to the EMF Compare. For instance, for the largest input model the execution time taken by EMF Compare was 780ms; whereas, our approach took 490ms. These experiments show the fast execution of our approach as compared with EMF Compare and also the scalability of our approach as the time difference between Case 5 and Case 6 for the largest input model was 4ms in our approach and 13ms for EMF Compare.

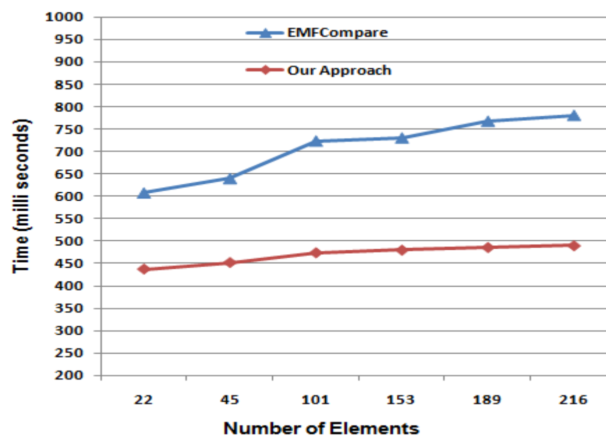

Fig. 8: Comparison chart for execution time in millisecond

4.2 OTHER EVALUATION PARAMETERS

Beside performance evaluation given in Section 4.1 here we further evaluate our approach by the parameters given mentioned by Fortsch and Westfechtel [15]. For model difference, it includes the requirements of accuracy, high conceptual level, domain independence, tool independence, history independence, efficiency, and user-friendly representation. Below we examine how well these criteria were met by our approach.

i. **Accuracy:** This requirement deals with the accuracy of the result, i.e., the diff tool (or model diff) calculate the difference between two versions v1 and v2 as precisely as possible.

   Achievement: The main problem with text-based approaches for models is the sensitivity issue related to the layout and order of text line which sternly affect the accuracy of text-based approaches. Since our approach is not text-based rather we used a graph structure the sensitivity to layout and order of text doesn't remain an issue at all. This increases the performance of the algorithm in form of accuracy of result. We also performed empirical tests to show the accuracy of our approach.

ii. **High Conceptual Level:** The diff tool calculates differences on a high level of abstraction, i.e., it has to operate on logical rather than physical level.

   Achievement: This quality is also achieved since the differences are calculated based on the type of model element. For instance, in case of class diagram, the differences between the classes, their attributes, operations and relationships are calculated.

iii. **Domain Independence:** The diff tool should be applicable to a large set of diagram types.

   Achievement: Apart from other advantages, such as overcoming sensitivity problem, accuracy etc, one important benefit of the data model used in our approach is that it provide generality, i.e., it can be used to represent different types of domain specific level diagrams at fine-grained level.

iv. **Tool Independence:** The diff tool should be independent of the tools which were used to create the diagram versions to be processed.

   Achievement: Approaches to model diff are state-based and operation-based. In operation-based approach, changes are described by using the original sequence of editor operations that caused the changes. In contrast, in a state-based approach only the state representations of different versions are compared. Differences or deltas are reconstructed using a difference algorithm that compares the different state representations. The major drawback of operation-based approach is that it is tool-dependent, while we apply a state-based approach to compare the two versions, thus our approach is tool independent.

v. **History Independence:** The result produced by the diff tool should depend only on the final states of the diagram versions, but on the history of edit operations used to create these versions.

   Achievement: History independence is based on tool independency. Since our approach is not based on tool or recording the history of edit operations rather than on the final states of diagram versions, therefore our approach is history independence.

vi. **Efficiency:** The diff tool should calculate the result as fast as possible, requiring as little space as possible.

   Achievement: For calculation of efficiency of our approach we performed empirical tests. We compare our approach with open source tool EMFCompare. The tests results showed that our approach produced both accurate and fast results. The space complexity for the diff process is 2n, since both states need to be present.

vii. **User-friendly Representation:** The tool should represent its output in a user friendly way.

   Achievement**:** The output of diff is brought back to models. Thus, the developer can see the differences between the model elements in a user friendly way.

5.0 CONCLUSION

In this paper we presented a generic graph structure representation for models based on which we developed our model diff algorithms. We first classified SCM systems in two broad categories, i.e., File-based SCM systems and Model-based SCM systems. File-based SCM systems are traditional text-based systems such as, Subversion, CVS, which consider software artifacts as a set of text files. File-based SCM systems gained high acceptance in software development when dealing with source code or other textual software artifacts, but failed severely in handling software artifacts with diagrammatical representation. In the era of MDE a paradigm shift in software development occur from code-centric to model-centric activities. In MDE models having diagrammatical representation, such as UML, becomes the central artifact in the software development process. All software development activities ranging

from analysis to the maintenance of software system heavily depends on graphical models. Traditional file-based SCM systems are unable to handle this paradigm shift in order to perform SCM activities at the appropriate level of abstraction. The goal of this work was to address this paradigm shift. By our generic graph structure representation we were able to avoid the problems of textual representation of models, such as, layout change, reshuffling issue, etc. Contrary to existing approaches our approach did not require changes in algorithm by changes the models. The approach used a state-based technique to compute delta thus it is tool-independent. The approach allowed the developers to be flexible in selecting model editor tool for developing models and also as our approach was history-independence we were not requiring the history of edit-operations of the tool for performing diff or merge. Finally, we showed the performance of our approach w.r.t. open source tool EMF Compare. The results of the tests we performed showed that our approach produced more accurate and fast results than EMF Compare. We set the future direction of our work as follows. In model diff activities an appropriate visualization of differences between two versions is important for understanding the differences therefore we need some technique to visualize the differences in user friendly manner. Furthermore in future we will work on rest of the version control activities such as model merge, conflict detection and version control policy.

REFERENCES

[1] Michael 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] Eclipse Modeling Framework (EMF), http://www.eclipse.org/modeling/emf/.

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

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

[6] Dirk Ohst, Michael Welle, Udo 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*, , *New York, NY, USA, 2003, ACM*, pages 227–236, ISBN 1-58113-743-5, doi:http://doi.acm.org/10.1145/940071.940102.

[7] U.Ohst, D., M.Welle, Kelter, "Merging uml documents," *Technical report, University Siegen*, 2004.

[8] M.Girschick, "Difference detection and visualization in uml class diagrams," *In Technical University of Darmstadt, Technical Report TUD- CS-2006-5*, pp 37–51, 2006.

[9] Sabrina Fortsch, Bernhard Westfechtel, "Differencing and merging of software diagrams-state of the art and challenges, " *In ICSOFT(SE)*, pp 90–99, 2007.

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

[11] Eleni 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.

[12] Udo Kelter, Jrgen Wehren, Jrg 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.

[13] Akhil Mehra, John Grundy, John 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, New York, NY, USA, 2005, ACM,* pp 204–213, ISBN 1-59593-993-4, doi:http://doi.acm.org/10.1145/1101908.1101940.

[14] 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.

[15] Sabrina Foertsch and Bernhard Westfechtel. "Differencing and merging of software diagrams - state of the art and challenges." *In ICSOFT (SE), Proceedings of the Second International Conference on Software and Data Technologies,* Volume SE, Barcelona, Spain, July 22-25, 2007, pp 90-99.

[16] W.L.Yeow, R. Mahmud, R.G. Raj, "An application of case-based reasoning with machine learning for forensic autopsy", *Expert Systems with Applications*, Vol 41, No. 7, 2014, pp. 3497-3505, ISSN 0957- 4174, http://dx.doi.org/10.1016/j.eswa.2013.10.054. (http://www.sciencedirect.com/science/article/pii/S0957 417413008713).

[17] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, David Launay, "Neo4EMF, A Scalable Persistence Layer for EMF Models," *In Modelling Foundations and Applications, Lecture Notes in Computer Science, Springer International Publishing,* 2014, Vol. 8569, pp 230-241.

[18] Westfechtel, Bernhard. "A formal approach to three-way merging of EMF models." *In Proceedings of the 1st International Workshop on Model Comparison in Practice, ACM*, 2010, pp. 31-41

[19] M. Lamine Ba, Talel Abdessalem, and Pierre Senellart. 2013. "Uncertain version control in open collaborative editing of tree-structured documents". *In Proceedings of the 2013 ACM symposium on Document engineering (DocEng '13). ACM, New York, NY, USA,* pp. 27-36. DOI=10.1145/2494266.2494277 http://doi.acm.org/10.1145/2494266.2494277

[20] Taentzer, Gabriele, Claudia Ermel, Philip Langer, and Manuel Wimmer. "A fundamental approach to model versioning based on graph modifications: from theory to implementation." *Software & Systems Modeling* February 2014, Vol. 13 No. 1, , pp 239-272

[21] Völter, Markus, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. "*Model-driven software development: technology, engineering, management*". John Wiley & Sons, 2013.

[22] Edward H. Berso_, Vilas D. Henderson, and Stan G. Siegel. Software configuration management. Software configuration management: a tutorial. *IEEE Computer*, Vol. 12 No. 1, January 1979, pp. 6-14. [23] Reidar Conradi and Bernhard Westfechtel, "Version models for software configuration management". *ACM Comput. Surv.* Vol. 30 No. 2, June 1998, pp. 232-282.

[24] Jacky Estublier, "Software configuration management: a roadmap", *Proceedings of the Conference on The Future of Software Engineering*, June 04-11, 2000, Limerick, Ireland, pp.279-289.