Research Journal of Applied Sciences, Engineering and Technology 11(3): 235-244, 2015
DOI: 10.19026/rjaset.11.1712
ISSN: 2040-7459; e-ISSN: 2040-7467
© 2015 Maxwell Scientific Publication Corp.
Submitted: May 31, 2014 Accepted: August 19, 2014 Published: September 25, 2015

Research Article

A Fine-granular Data Model for UML-compliant Models in a Model-based Software Configuration Management Systems

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

Abstract: Software Configuration Management (SCM) aims to provide a controlling mechanism for software artifacts created during the software development lifecycle. Traditional SCM systems are file-centric and consider software systems as a set of text files. Today software development is model-centric. New challenges such as model diff, merge and evolution control arise while using models as central artifact, traditional systems are unable to resolve these challenges adequately. In its essence these challenges are mainly due to the inappropriate representation of models at fine-granular level by traditional systems. In this study we present a generic data model to represent model at fine-grain level. We use graph structures to represent models at fine-granular level, which is an intermediate representation based on graph theory. By transforming models into the graph structures we get several ad-vantages. Firstly, we avoid several problems associated with textual representation of models. Secondly, we can handle different types of UML diagrams. Thirdly, it can be used to develop a generic model-based SCM framework, which provides model configuration management services for any UML model.

Keywords: Data model, file-based SCM system, fine-granular representation, model-based SCM system, model transformation, versioning

INTRODUCTION

Developing large software systems involving more than one persons essentially need the efficient management of software artifacts created during software development lifecycle. In the absence of controlled management, the software products that industry has turned out have typically been de-livered much later than scheduled, cost more than anticipated and have been poorly designed and documented (Bersoff *et al*., 1978). Software Configuration Management (SCM) aims to provide a controlling mechanism to such problems. It deals with controlling the evolution of software systems during development and maintenance. This requires many activities to perform, such as, construction and creation of versions, identification of differences between versions, conflict detection and merging (Conradi and Westfechtel, 1998; Koegel *et al*., 2010; Marcello *et al*., 2012; Xing, 2010). Traditional SCM Systems, such as Subversion (Pilato, 2004) and CVS (Cvs Project, 2012), are file-based, i.e., they consider software sys- tem as a set of text files mainly in the form of source code however, today software development is model-centric. Model-Driven Engineering (MDE) is a technique which aims to reduce the complexity of software development by assigning models a central role in the development process. Traditional SCM systems work for 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 are not well suited for performing con-figuration management tasks on models. Below we describe several reasons why existing systems are not adequate for performing SCM activities, such as, Model Diff, Merge and Evolution Control activities.

Fundamentally, the main reason of the inadequacy of existing systems is due to the fact that these systems are file-based and consider software artifacts as a set of text files having no relations (Ohst *et al*., 2003; Ohst *et al*., 2004). In contrast, models are graphs with nodes being complex entities and arcs (relations) containing a large part of model semantics. File-based SCM tools use textual or structured data (such as XMI) to represent models at fine-grained level. Representing models with textual or structured files is an inadequate solution since it requires operation on models at a low level of abstraction. For instance, a class diagram might be represented by a few lines of text in a 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

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

arbitrary order. Furthermore, the position where a class symbol appears in the diagram is explicitly stored in layout data. However, these textual representation are sensitive to changes of the order in which lines appear in a text file and they are also sensitive to changes in the layout. To a large ex- tent, 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 will hardly produce meaningful results. Thus, one of the main motivation of our work is to use a suitable representation for models at fine-grained level to perform Model Diff and Merge activities. Another problem of this order and layout sensitivity is that even small changes in the diagram can lead to a complete reshuffling of the file contents resulting in a large number of significant textual differences. It is also possible that different file contents actually represent the same diagram. Traditional SCM systems are unable to handle this situation adequately.

In its essence these challenges are mainly due to the inappropriate representation of models at fine-granular level. In this study we present a generic data model, i.e., a Domain Specific Language (DSL) for graph structures to represent UML models at fine-grain level based on which one can avoid the problems of model diff, merge and conflict detection activities. Apart from other advantages, such as avoiding the sensitivity problem of textual representation of models, one important benefit of graph structure DSL is that it is generic and can be used to represent different types of UML models at fine-grained level.

MATERIALS AND METHODS

We classify SCM systems into two main categories:

- File-based SCM Systems
- Model-based SCM Systems

File-based SCM systems: File-based SCM systems are traditional text-based systems, which consider software artifacts as a set of text files. These systems, such as, Subversion and CVS, assume set of files as implicit tree structure with nodes being text files having no relations. They 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 they take in case of modification of a document is that one or few adjacent lines of the text are inserted, deleted or modified.

Ohst *et al*. (2003) and Ohst *et al*. (2004) identify several reasons under which these sys- tem failed to work for document management in the early phases of software development. 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 or XML formats. For instance, 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. Furthermore, the position where a class symbol appears in the diagram is explicitly stored in layout data. However, these textual representation are sensitive to changes of the order in which lines appear in a text file and they are also sensitive to changes in the layout. To a large extent, the order of text lines and their layout is immaterial for diff and merge operations on models. Consequently, even small changes in the diagram can lead to a complete reshuffling of the file contents resulting in a large number of significant textual differences. Its also possible that different file contents actually represent the same diagram. Therefore, applying diff and merge operations at the level of plain text will hardly produce meaningful results.

Furthermore in MDE, analysis, design, implementation and testing are considered parallel and coordinated activities. As a result, even simple modifications can affect several files, or part of files, belonging to different development phases. Traditional file-based SCM systems are unable to correctly represent such changes due to the fact that they are not aware of the logical structure and interlink dependencies between these documents. Further-more, traditional SCM systems manage changes on a line-oriented level. In contrast, many software engineering artifacts are not managed on a line- oriented level. For instance, adding an association between two classes in a UML class diagram is neither line-oriented nor can the change be managed in a line-oriented way. A single structural change in a diagram is usually managed as multiple line changes by traditional SCM systems.

These dissimilarities clearly indicate that file-based and model-based SCM cannot be handled in the same way.

Model-based SCM systems: As discussed in previous section traditional file-based SCM systems consider software artifacts as a set of text files. However, with the advent of MDE, models become first class artifacts which evolve over the period of time due to unavoidable changes. For instance, requirements change when developers improve their understanding of the application domain, design changes with the identification of new technologies and design goals or with the identification of new solution options, similarly implementation changes for correctness and enhancement purposes (Kogel, 2008). These changes can affect every work product, from software design to source code. Furthermore, models are no longer

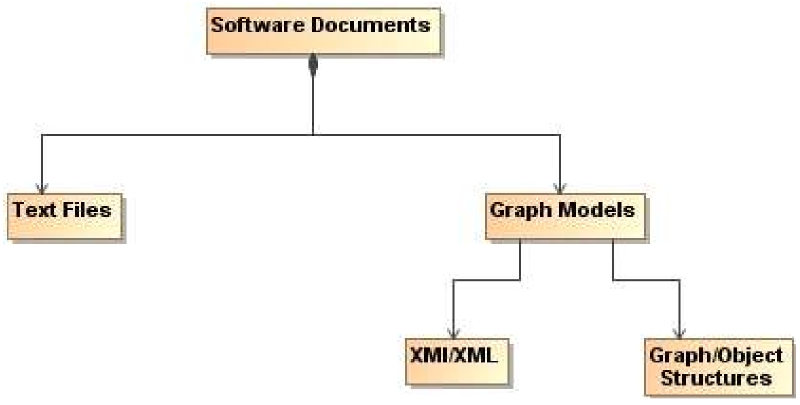

Fig. 1: Software document representations

just for documentation, they are now considered as the basis for generating executable software and many other activities such as, versioning, testing, etc. Therefore, a software configuration system for models is crucial for an effective, collaborative software development process, which considers models as central artifacts and performs SCM activities on models. In other words we need a paradigm shift from traditional file-based SCM systems to model-based SCM systems. Before we elaborate our framework, first we define our proposed target DSL, since we transform the instances of any source DSL into graph structure for further processing like model diff and merge.

LITERATURE REVIEW

In software development lifecycle two main types of software documents are text files and graphical models (Fig. 1). Text files may contain source code, documentation etc, whereas graphical models are for instance UML models. A model can be represented in three different ways (Ohst *et al*., 2004):

- The graphical representation i.e., the diagram itself
- The persistence representation e.g., XMI
- Intermediate representation e.g., syntax tree or graph structure. The graphical representation is the coarse-grained while the other two are fine-grained representations. Normally a model can be stored at fine-grained level by structure data like XMI or XML (Girschick and Darmstadt, 2006; Wang *et al*., 2003), but this kind of representation is not well suited for model differentiation purposes as pointed out in Ohst *et al*. (2003) and Selonen and Kettunen (2007) and elaborated in above section.

A data model defines the elements, attributes and relationships between the elements at fine-grained level (Fortsch and Westfechtel, 2007). 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 allows for simple and efficient diff and merge algorithms. Different approaches use different data models to perform diff and merge activities. For example, in Alanen and Porres (2003) data model are based on MOF and are thus applicable to MOF instances. Kelter *et al*. (2005) and Xing and Zhenchang (2005) data models are trees with typed elements, that can be decorated with attributes. Ohst *et al*. (2003) and Ohst (2002) use a fine-grained data model for UML class diagram which resembles a syntax tree. All elements of a UML diagram are modelled as separate objects, e.g., all classes, all operations and all attributes.

MODEL-BASED SCM FRAMEWORK

As identified in above section existing file-based SCM systems are not adequate for performing SCM activities on models. Keeping the issues of file-based SCM systems this paper proposes a generic model-based SCM framework, which aims to overcome the challenges faced by existing systems when dealing with models as central artifact and is able to handle MOF-Compliant DSLs.

A use case view of the approach is given in Fig. 2. First a developer develops the source models conforming to any MOF-compliant source DSL. As a source DSL we are using UML (Object Management Group, 2003). A source model conforming to source DSL is transformed into target model conforming to target DSL in model transformation step. In the approach the source DSL is not fixed while the target DSL is fixed.

The source models will be transformed into target models conforming to our proposed target DSL by applying the concept of Model-to-Model transformation (Czarnecki and Helsen, 2003). The transformation from source to target models is based

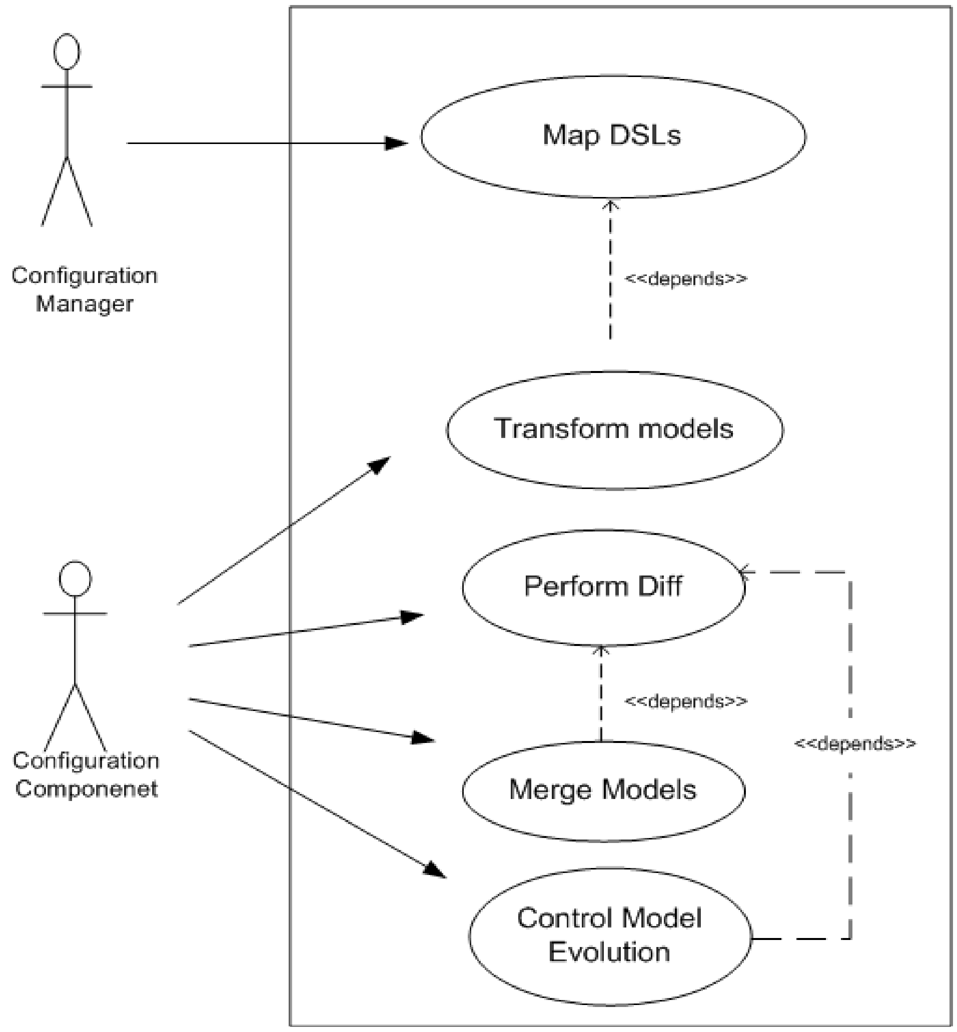

Fig. 2: Configuration management-use case view

on the mappings defined by Configuration Manager (CM) between the source and target DSLs. Further details about model transformation will be given in below section. After model transformation rest of model-based configuration management activities such as model diff, merge and evolution control will be performed on the transformed models, however these activities are not in the scope of this study.

Graph DSL: A DSL for graph structures: In our approach, at a fine-grained level we represent models in an intermediate representation, i.e., graph structures. The meta model of the graph structure is given in Fig. 3. It represents graph with typed elements, that can be decorated with attributes. The main concepts of the meta model are Node, Edge, Link, Operation, Attribute, Parameter and Data Type. Apart from other advantages, one important benefit of this meta model is that it is generic and can be used to represent different types of UML and MOF-compliant DSL diagrams at fine-grained level, since in essence most of the UML diagrams except sequence diagram represent a graph (Ohst *et al*., 2003). Below we give the description of these concepts.

Node: A 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: An Attribute represents data which represent features of node. They are identified by name and have a data type.

Data type: A Data type model simple types such as Int, String, Boolean etc. They are identified by name and are most commonly used as attribute types.

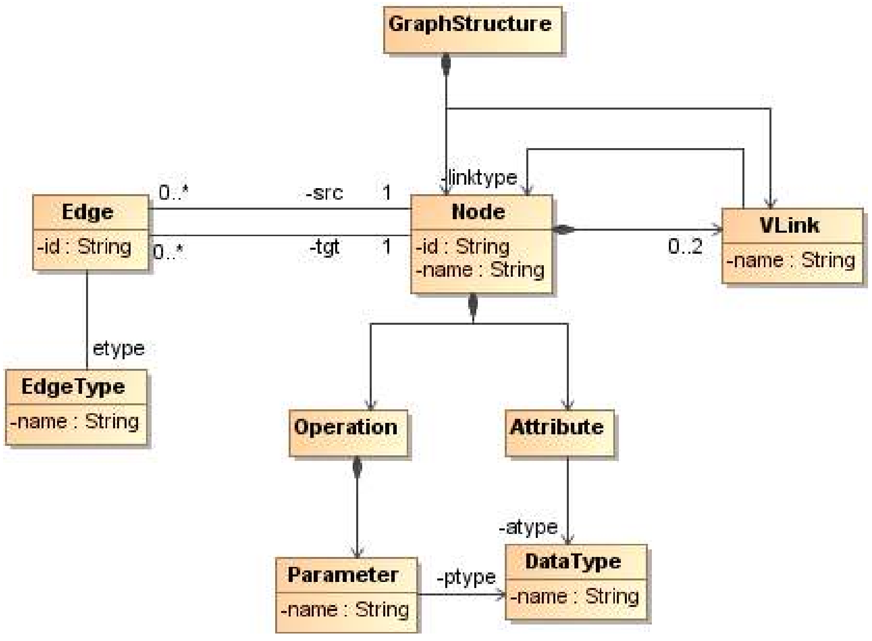

Fig. 3: Abstract syntax of graph structure

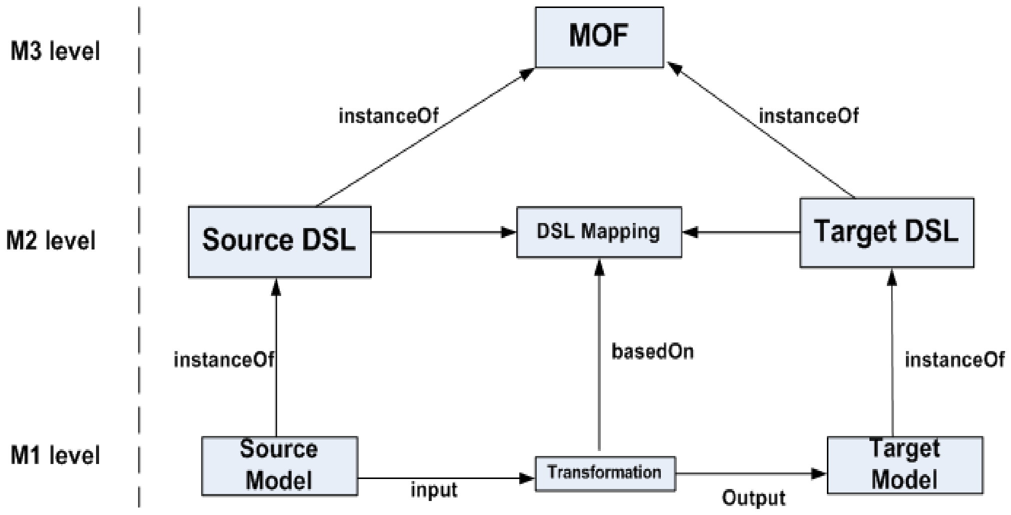

Fig. 4: Model transformation hierarchy

Edge: An 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 Edge type of the edge, which includes association, inheritance, containment etc.

Operation: An Operation represent 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: A 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.

Model transformation in Graph DSL: We will transform UML-compliant models into the instances of above defined target DSL and perform the diff, merge, evolution control activities on it. For transformation we use the mappings at DSLs level, i.e., between source DSL and target DSL.

The model transformation hierarchy is given in Fig. 4. In the hierarchy at M3 level we have MOF metametamodel, at M2 level we have MOF-

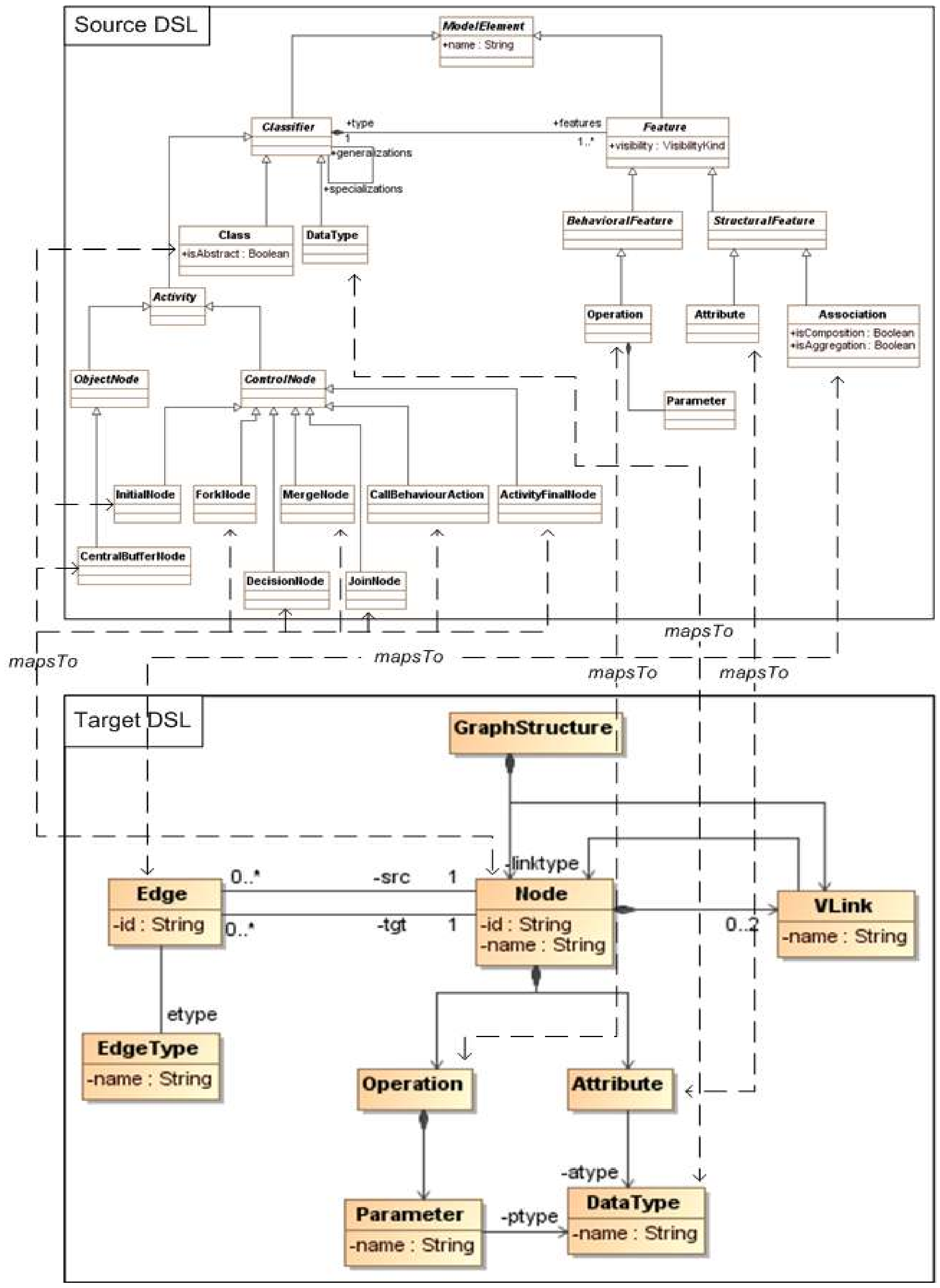

Fig. 5: Mappings between UML and GraphDSL

compliant source and target DSLs and the mappings between them and at M1 level we have source models and target models. At M1 level we also have the transformation specifications to transform the source models into target models.

Since we transform the instances of any source metamodel into the instances of graph structure target DSL for further processing like model diff and merge, the configuration manager first performs mappings at the DSL level. The mappings are done between the elements of source and target DSL. Below we describe the mappings between UML and GraphDSL.

Mapping between UML and graph DSL: Figure 5 a simplified view of UML metamodel supporting class and activity diagram and their mappings with GraphDSL element is given. Fol-lowing are the mappings between source DSL, i.e., UML and target DSL, i.e. Graph structure DSL.

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

Activity-to-node mapping: Similar to the classifier Class, classifier. Activity also define a set of model entities. The corresponding concept in Graph Structure DSL is defined by Node. Since Activity is a super type of classes Initial Node, Fork Node, Merge Node, Join Node, Decision Node, Call-Behaviour Action, Activity Final Node and Central Buffer Node therefore, we map all the subtypes of Activity onto Node.

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

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

Parameter-to-Parameter Mapping: Parameters are defined in the UML as Parameter. The corresponding concept in Graph Structure DSL is defined by Parameter. Therefore, we map Parameter onto Parameter.

Data type-to-data type mapping: Data types are defined in the UML as Data Type. The corresponding concept in Graph Structure DSL is defined by Data Type. Therefore, we map Data Type onto Data Type.

Table 1: Mapping between UML and GraphDSL

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

Association-to-edge mapping: A relationship between two entities is described by Association in UML. The corresponding concept in Graph- Structure DSL is defined by Edge. Therefore, we map Association onto Edge. The type of Association corresponds to the type of Edge, i.e., Ed-geType (Table 1).

RESULTS AND DISCUSSION

As part of the work, we did a prototype implementations for model trans-formation component. The implementation is done using the open source (EMF, 2012) framework using Java as a source language.

Figure 6 and 7 shows the reference architecture of our frame-work. The Model Editor will be used by the software developer for develop-ing source models. The software developer is free in the choice of selecting a model editor provided that the model editor has the functionality of seri- alizing models in XMI representation. For instance, MagicDraw and EMF Editor both provide this facility to store and retrieve models in XMI. EMF uses XMI for default serialization, whereas MagicDraw extends its support of MDA tools by adding capability to export of MagicDraw UML model to EMF based on UML 2 XMI. The Model Loader component loads the models in EMF for further processing.

Figure 7 shows the architecture of model transformation module. It consists of Model Loader and Model Transformer component. It uses Model Loader component for taking the XMI input of the models. After loading the model the Model Transformer component is used to transform the model into graph structure. The Model Transformer component traverse the model elements.

The Model Transformation component loads the inputs model conforming to source DSL and transform it into graph structures conforming to target DSL according to the MDA model transformation hierarchy. The Model Transformation component also takes DSL mappings as input.

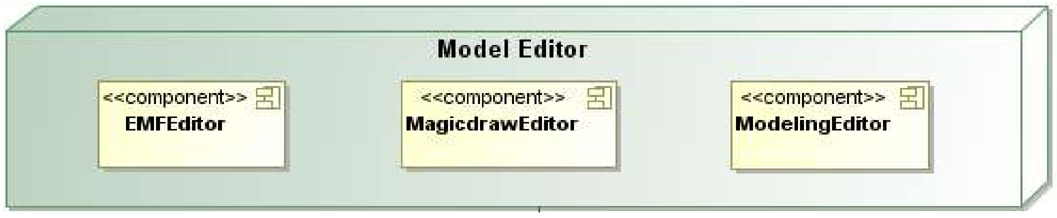

Fig. 6: Model editor architecture

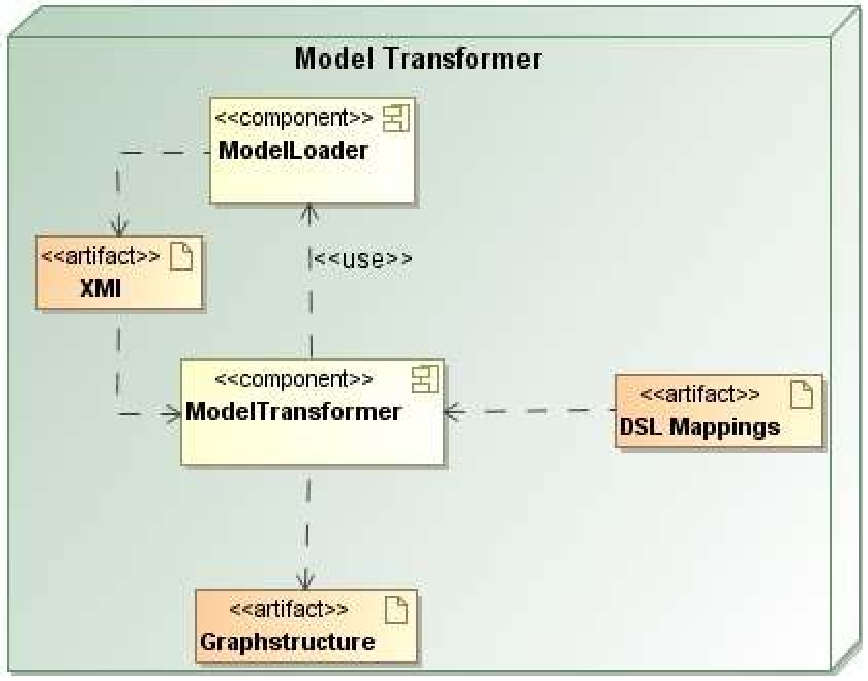

Fig. 7: Transformation architecture

The output of the Model Transformer are graph structures of models.

Algorithm 1 shows the pseudo code for transforming DSL models into Graph DSL models based on the mappings for the given DSL. The trans-formation algorithm works as follows. In order to transform DSL models first models will be loaded from memory then model's root element will be accessed. The root element is a container for all the elements of the models. Afterwards the root element will be traversed for its contents, i.e., traverse root. eContents (line 1-31). Then for all the elements in node mappings the traversed element is checked for node mapping if the element is mapped to node then first it will be typecast to appropriate type (line 2–4), for instance, if the element is E-Class then first typecast the element into E-Class. Then a new Node will be created in the target Graph DSL model by calling the method create new Node() given in algorithm 2 (line 5). Afterwards the element will be traversed for its contents, i.e., eM.e Contents (line 6-28). Then for all elements in the attribute mappings the traversed element will be checked for attribute mapping if the traversed element is mapped to At-tribute then first typecast the element into eA of appropriate type (line 7-13), for instance, if the element is of type EAttribute then first typecast the element into EAttribute. Then set new Node attribute as eA.name and its type as eA.type. Then for all elements in the edge mappings the traversed element will be checked for edge mapping if the traversed element is mapped to Edge then first typecast the element into eR of appropriate type (line 14-20). Then set new Node edge as eR.name and its type as eR.type. Then for all elements in the operation mappings the traversed element will be checked for operation mapping if the traversed element is mapped to Operation then first typecast the element into eOp of appropriate type (line 21-27), for instance, if the element is of type EOperation then first typecast the element into EOperation. Then set new Node operation as eOp.name and its type as eOp. type. The whole process will be repeated until all the elements of root.eContents are traversed.

Algorithm 2 shows the pseudo code for creating nodes in Graph DSL. First a new Node of type Node will be initialized (line 1). The id and type of new Node will be id and type of the element which will be represented by the new Node (line 2-3). Initially header Node and last Node are set to null. Upon creation of first new Node the header Node and last Node will point to new Node (line 4-8). Upon creation of second new Node the header Node will point to the first new Node and last Node will point the second new Node and so on. Finally, the created new Node will be returned to the calling procedure (line 9).

CONCLUSION

In this study we presented a generic data model for UML-based models in model-based SCM systems. The main contribution of this paper is defining an adequate representation for UML models to be presented at fine-grained level. This is what called by Frtsch and Westfechtel (1998) as determination of document model. The document or data model defines the elements, relationships and attributes to be considered and has a strong impact on the capabilities of the diff and merge tool. Traditional SCM systems uses simple file-based data model to represent software artifacts, i.e., as a set of text files having no relations and some metadata information about the files. Due to this file-based data modeltraditional SCM tools uses textual or structured data to represent models at fine-grained level, which causes many problems in performing SCM activities on models. For instance, textual representation are sensitive to both changes in the order of text and changes in the layout. To a large extent, the order of text lines and their layout is immaterial for diff operation of models. Therefore, applying diff operation at the level of plain text will hardly produces meaningful results. We presented a generic graph structure representation for models based on which we developed model transformations algorithm. By our generic graph structure representation we are able to represent any UML-based models. Thus our approach is not limited to any specific model. Furthermore, we are able to avoid the problems of textual representation of models, such as, lay- out change, reshuffling issue, etc. Finally, our approach is tool-independent. Our approach allows the developers to be flexible in selecting model editor tool for developing models. As a future direction we will work on the rest of the component of our model-based software configuration management framework. First the core activity of model-based SCM, i.e., model diff, will be performed. Model diff deals with comparing two versions in order to detect differences and matches between them. Afterward the model merge which deals with merging two or more models will be performed (Algorithm 1 and 2):

Algorithm 1: Transform DSL

```
Require: Graphstructure gs and EObject root
1:  for all element eM in model's root.eContents do
2:      for all NodeMappings do
3:          if eM mapsTo Node then
4:              typecast an element eN to type eM
5:              newNode = createNode(eN.id,eN.type)
6:              for all element e in eM.eContents do
7:                  for all AttributeMappings do
8:                      if e mapsTo Attribute then
9:                          typecast an element eA to type e
10:                         set newNode.attribute←eA.name
11:                         set newNode.attributetype←eA.type
12:                     end if
13:                 end for
14:                 for all EdgeMappings do
15:                     if e mapsTo Edge then
16:                         typecast an element eR to type e
17:                         set newNode.edge←eR.name
18:                         set newNode.edge←eR.type
19:                     end if
20:                 end for
21:                 for all OperatopmMappings do
22:                     if e mapsTo Operation then
23:                         typecast an element eOp to type e
24:                         set newNode.operation←eOp.name
25:                         set newNode.operationtype←eOp.type
26:                     end if
27:                 end for
28:             end for
29:         end if
30:     end for
31: end for
```

Algorithm 2 creatNode

```
Require: entityId and entity Typ
1:      instantiate new Node of type Node
2:      set newNode.id←entity Id;
3:      set new Node. type←entity Type;
4:      if header Node ≡ null then
5:          header Node←new Node;
6:      else if last Node = null then
7           lastNode.v node←newNode;
8:      end if
9:      return new Node
```

REFERENCES

Alanen, M. and I. Porres, 2003. Difference and union of models. Proceeding of the UML Conference. Springer-Verlag, LNCS 2863, San Francisco, California, pp: 2-17.

Bersoff, E., V.D. Henderson and S.G. Siegel, 1978. Software configuration management. Proceeding of the Software Quality Assurance Workshop on Functional and Performance Issues, pp: 9-17.

Conradi, R. and B. Westfechtel, 1998. Version models for software configuration management. ACM Comput. Surv., 30(2).

Cvs Project, 2012. Retrieved form: URL http://www.nongnu.org/cvs.

Czarnecki, K. and S. Helsen, 2003. Classification of model transformation approaches. Proceeding of 2nd OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

EMF (Eclipse Modeling Framework), 2012. Retrieved form: URL http://www.eclipse.org/modeling/emf/.

Fortsch, S. and B. Westfechtel, 2007. Differencing and merging of software diagrams-state of the art and challenges. Proceeding of International Conference on Software and Data Technologies. Barcelona, pp: 90-99.

Girschick, M. and T. Darmstadt, 2006. Difference detection and visualization in UML class diagrams. Technical Report TUD-CS-2006-5, 2006.

Kelter, U., J. Wehren and J. Niere, 2005. A generic difference algorithm for UML models. In: Liggesmeyer, P., K. Pohl and M. Goedicke (Eds.), Software Engineering. Lecture Notes of Informatics, Gesellschaft für Informatik, Essen, Germany, P-64: 105-116.

Koegel, M., M. Herrmannsdoerfer, Y. Li, J. Helming and J. David, 2010. Comparing state- and operation-based change tracking on models. Proceeding of the 14th IEEE International Enterprise Distributed Object Computing Conference (EDOC '10), pp: 163-172.

Kogel, M., 2008. Time-tracking intra- and inter-model evolution. Proceeding of Software Engineering Conference-Workshop, 2008.

Marcello, L.R., D. Marlon, U. Reina and M.D. Remco, 2012. Business process model merging: An approach to business process consolidation. ACM T. Softw. Eng. Meth., 22(2).

Object Management Group (OMG), 2003. Unified modeling language 2.0 infrastructure specification. September 2003.

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

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. Internal Report, University of Siegen.

Pilato, M., 2004. Version Control with Subversion. O'Reilly and Associates, Inc., Sebastopol, CA, USA, 2004, ISBN: 0596004486.

Selonen, P. and M. Kettunen, 2007. Met model-based inference of inter-model correspondence. Proceeding of 11th European Conference on Software Maintenance and Reengineering (CSMR, 2007), pp: 71-80.

Wang, Y., D.J. DeWitt and J.Y. Cai, 2003. X-diff: An effective change detection algorithm for XML documents. Proceeding 19th International Conference on Data Engineering, 54: 519-530.

Xing, E. and S. Zhenchang, 2005. Umldiff: An algorithm for object-oriented design differencing. Proceeding of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp: 54-65.

Xing, Z., 2010. Model comparison with genericdiff. Proceeding of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10), pp: 135-138.