Research Journal of Information Technology 5(3): 72-80, 2013
DOI:10.19026/rjit.5.5790
ISSN: 2041-3106; e-ISSN: 2041-3114
© 2013 Maxwell Scientific Publication Corp.
Submitted: March 13, 2013 Accepted: April 29, 2013 Published: September 01, 2013

Research Article

Versioning and Evolution Control of Models in Software Configuration Management System

Waqar Mehmood and Nadir Shah, Ehsan Munir
Comsats Institute of Information Technology Wah Campus, Wah Cantt, Pakistan

Abstract: In this study we present an approach to address the issues of synchronization, evolution control and version granularity in Software Configuration Management (SCM). Our approach is based on a unified model developed during software lifecycle. The unified model consists of a set of different kinds of model and the interlinks information between these models, such models includes Analysis and design model, Test models etc. These models may possibly be created using different development tools in a heterogeneous environment. Our approach is based on identifying interlinks dependencies between different model elements. By using these interlinks information we develop our evolution control policy and perform synchronization of models elements.

Keywords: Evolution control, modeling, software configuration management, versioning

INTRODUCTION

Software Configuration Management (SCM) is a discipline for controlling the evolution of software systems. SCM serves two different needs (Conradi, 1998):

- **As a management support discipline:** By identifying product components and their baselines, controlling changes (establishing a process for change) and auditing the product (quality assurance)
- **As a development support discipline:** By accurately recording the composition of versioned software products evolving into many revisions and variants, maintaining consistency between inter-dependent components and reconstructing. In this study the presented work and the discussed approaches falls into the category of development support discipline.

In development support discipline versioning is the key activity. The two main types of artifacts in software development are graphical models and textual files such as code. Fundamentally, the main differences between code and model versioning occur because of their different structures. Code versioning assumes an implicit tree structure with nodes being text files and with no relations. In contrast, models are based on graphs, with nodes being complex entities and arcs (relations) containing a large part of the model semantics. These dissimilarities clearly indicate that code and model versioning cannot be handled in the same way.

MDE dream is to perform SE activities only on models. In reality Models and files co-exist and will have to be managed together consistently. As identified in Jacky *et al*. (2009, 2010) this situation requires the definition of new evolution paradigms for software projects that are made of a mixture of models and files. Defining such an evolution paradigm requires on one hand to take into account the different nature of models and on other hand it must rely on the tools and systems available in traditional SE. Moreover the issue of synchronization, definition of new evolution policy and maintaining the consistency and completeness of composite objects need to be addressed.

In this study we present an approach to address all the issues described above using a unified model of the software lifecycle. Our unified model consists of a set of different kinds of model and the interlinks information between these models, such models includes Analysis and design model, Test models and System implementation model etc. These models may possibly be created using different development tools in a heterogeneous environment. Our approach is based on identifying interlinks dependencies between different model elements. By using these interlinks information we develop our new evolution control policy and perform synchronization of models elements.

LITERATURE REVIEW

In this section we first describe some basic terminologies used in SCM systems (Conradi 1998; Marcello *et al*., 2012). A version V represents a state of an evolving item I. V is characterized by a pair V = (ps, vs), where ps and vs. denote a state in the product space

Corresponding Author: Waqar Mehmood, Comsats Institute of Information Technology Wah Campus, Wah Cantt, Pakistan

and a point in the version space, respectively. The term item covers anything which may be put under version control. A version model defines the object to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions. A version model is expressed considering both the product space, which represents the objects to be versioned and the version space, which represents the way versions are organized. Revision is often used to talk about the state of the same element, at different points in time. The last revision is usually better than the previous one. A workspace is a place where revisions can be copied from the repository and modified. A snapshot is a repository element that is the image of a set of objects as they were in a workspace when the snapshot was created.

Graphs are well suited to represent the organization of a versioned object base, even if the corresponding system is not graph-based. E.g Subversion, SCCS, RCS are file-based but the version space of a text file may be represented naturally as a version graph. Version granularity refers to the size of a version. In SCM literature version granularity are at three levels i.e., Component versioning, Total versioning and Product versioning. Component versioning means that only atomic objects (not the composite one) are put under version control. Each object has its own version space, modeled for example by a version graph. Total versioning applies to all levels of the composition hierarchy (if there are more than one composite object, each one has its own version space). Product versioning differs from total versioning be arranging versions of all objects in a uniform, global space (one version for whole models/files). The difference between two models is known as delta. Delta granularity refers to the size of those units in terms of which deltas are recorded. Two ways to calculate delta between two versions of a model are state based delta and operation based delta (Maximilian *et al*., 2010).

In state-based approach only the state representations of different versions are stored, possibly using compression or sharing of common parts (Koegel *et al*., 2010). Deltas are reconstructed using a differencing algorithm that compares the different state representations. In contrast, in operation-based approach, changes are described by using the original sequence of editor operations that caused the changes. It records a sequence of change operations $op_1, \ldots, op_n$ which, when applied to one version v_1, yields another version v_2.

MATERIALS AND METHODS

Domain specific Meta models: Figure 1 shows the basic structure of the Telling Test Stories artifacts (Breu *et al*., 2007). The artifacts are categorized along two orthogonal classifications: Model and Implementation on the one side and System Artifact and Test Artifact on the other side. The System Model describes the system requirements at a business oriented level. An important assumption in TTS framework is that System Model and System Implementation are traceable. The System Implementation is the executable system under test. We assume the implementation to be structured based on the notion of (software) components. The Test Model specifies test cases developed in a step by step process. This process includes the specification of scenarios (sequences of system service calls, where we name these service calls in the sequel actions), the specification of assertions (conditions to be checked during the test) and the description of data pools and system configurations. For describing test cases we use sequence diagrams and tabular representations of data and objects. The Test Implementation is generated from the Test Model which has interlinks dependencies with System Model and System Implementation.

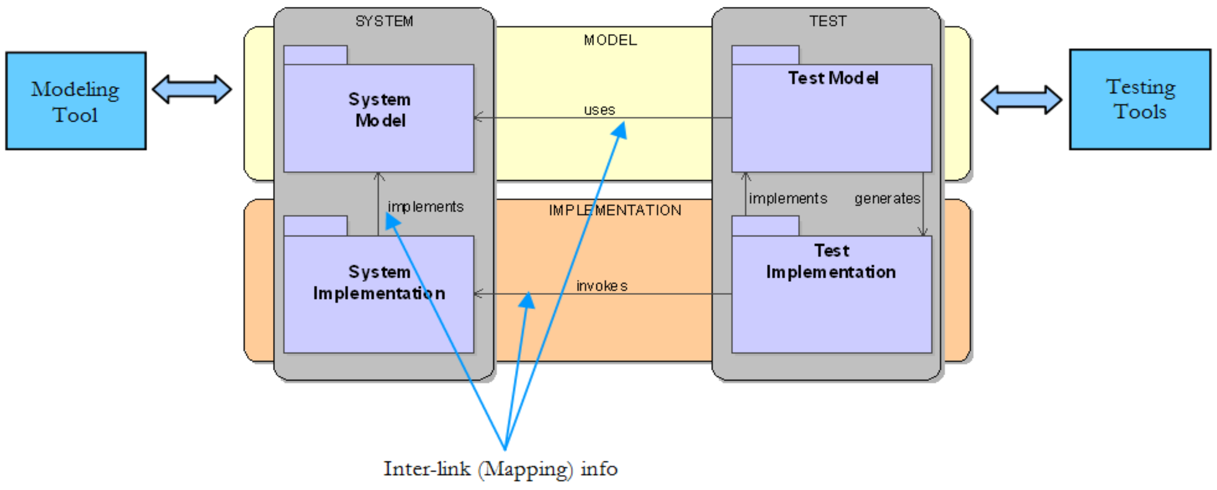

Fig. 1: Basic artifacts

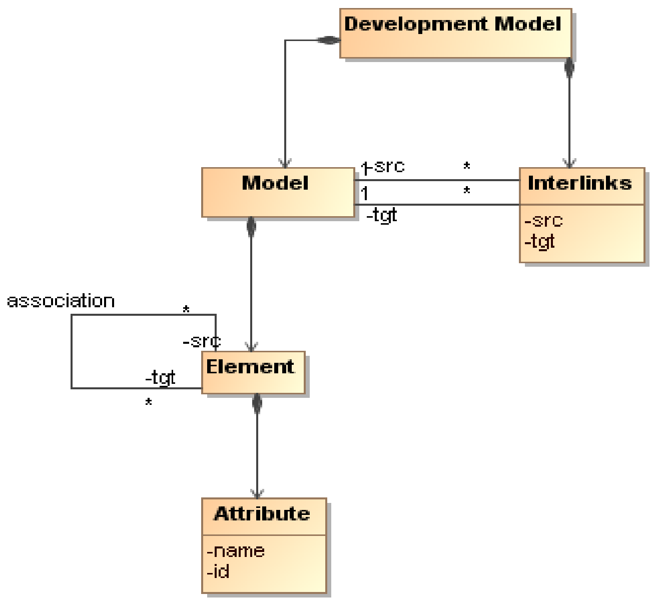

Fig. 2: Metamodel

Meta model for heterogeneous environment: Meta modeling is a common technique for conceptualizing a domain by defining the abstract syntax and static semantics of a DSML (Yuehua *et al*., 2007). It defines a set of modeling elements and their valid relationship that represent certain properties for a specific domain. The basic artifacts described above are constructed in a heterogeneous environment. For System Model and System Implementation one can use System modeling tools and for Test Model and Test Implementation, Testing tools can be used. Moreover, there are interlinks between different types of models, such as between System Model and Test Model etc. Such a heterogeneous modeling environment is depicted in Fig. 2. In Fig. 2, a Development Model consists of different types of model developed during system life cycle and the interlinks between those Models. Each Model itself consists of set of elements (entities) which has some attributes and association (intralinks) between them.

We can formally define the concepts as:

Let E be the universe of Model Entities. A Model is a tuple M = (E_M, REL_M), where,

- E = { e_1, e_2, ..., e_n} is a finite set of entities and $E_M \subseteq E$,
- $REL_M \subseteq E_M \times E_M$ is the intra-model relation between Model entities.

A Development Model is a tuple DM = (M, CON), where,

- M = { M_1, M_2, ..., M_n } is a finite set of Models and $\forall$ M_i, $M_j \in$ M, i $\neq$ j
- CON $\subseteq \cup E_i \times \cup\ E_i$is the inter-relationship between different modelelements. i.e., $e_i \in M_i$, $e_j \in$ M_j i $\neq$ j

Mapping Set: A mapping set MS can be defined as MS = {M,S,T} where,

- S $\in$ M represent the source model
- T $\in$ M represent the target model
- M: S $\rightarrow$ T is a partial function from S to T, such that given s $\in$ S, M return t $\in$ T, where (s,t) $\in$ CON

To include the configuration management information we further extend the Meta model in Fig. 2 and added Configuration Component as part of the Development Model in fix 3.0. The Configuration Component consists of four kind of information i.e., version granularity, interlink dependencies, evolution policy and consistency and completeness information. Further detail about these terms is given in next sections (Fig. 3).

Version model: We specify our version model in Fig. 4. It constitutes product space and version space. A version space defines the items to be versioned, the common properties shared by all versions of an item and the deltas, i.e., the differences between them. Furthermore, it determines the way version sets are organized. It defines whether a version is characterized in terms of the state it presents or in terms of some changes relative to some baseline. It selects a suitable representation for the version set (e.g., version graph) and it also provides operations for retrieving old versions and constructing new versions. Each

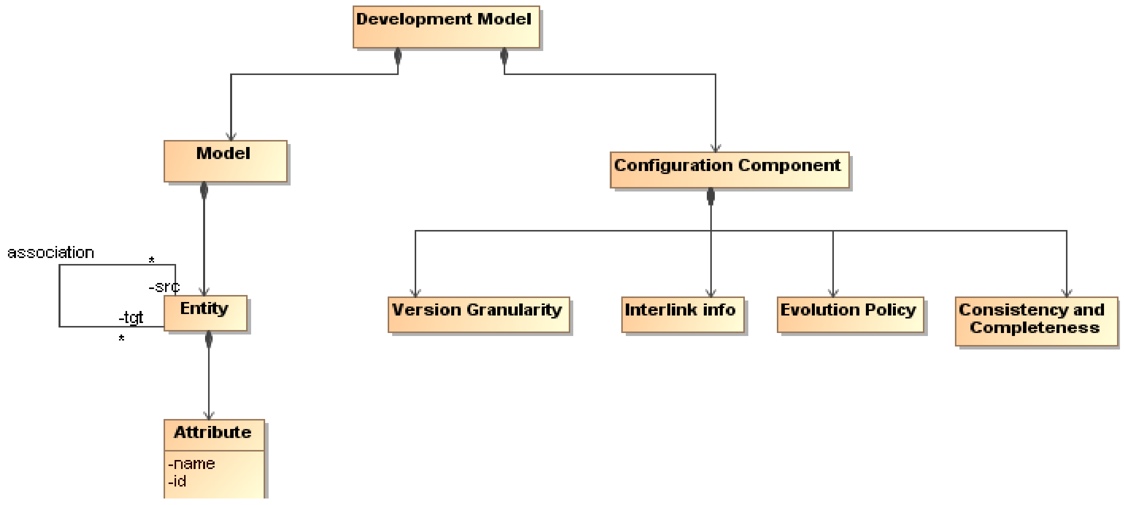

Fig. 3: Meta model extended

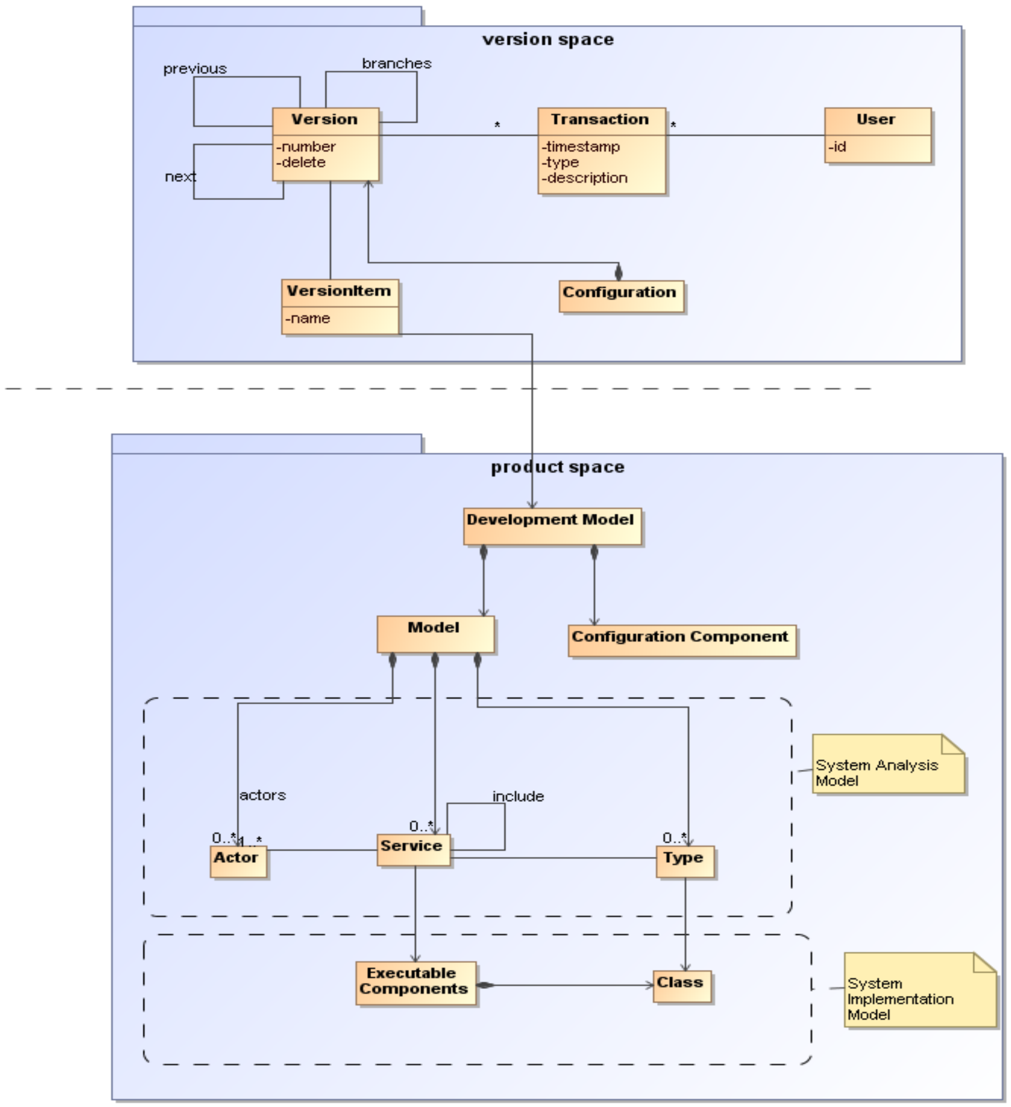

Fig. 4: Version model

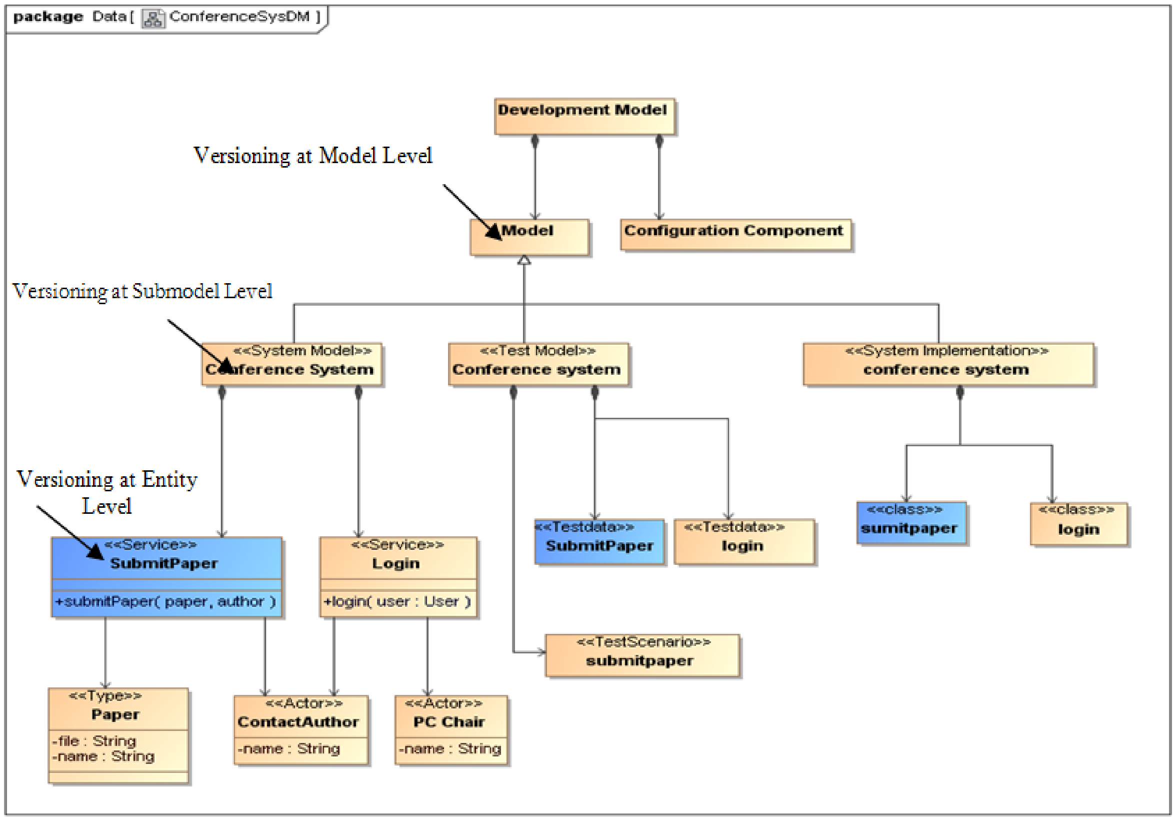

Fig. 5: Instance model

Configuration item is composed by versions. A specific attribute differentiate versions that were deleted by the user. Versions are queried or created by transactions. These includes both read-only and read-write transactions, such as history, checkout, check in, update etc. Versions have relationships to model elements that cross version model border. This relationship connects versioning model with product model, which is a UML, E Core or Domain Specific Meta Model (DSMM). In our case it is DSMM.

Product space describes the structure of a software product without taking versioning into account. It can be represented by a product graph whose nodes and edges correspond to software objects and their relationship. The product space in Fig. 4 contains a Development Model. Due to space limitation we just show System Analysis and Implementation Model as part of our Development Model for TTS framework. However it also consists of Test Model and Test Implementation as we seen above, an instance of all Model is given in Fig. 5. The System Model is a description of the functional system requirements at business level based on the following three core concepts. A (system) service describes the basic functionality the system provides to the outside. We use the notion of service instead of use case since many of the systems which are target of our method are modeled as service oriented architecture. Services may be hierarchically structured. Services have input parameters and an output parameter. All parameters are either basic data types or of a Class type. Finally, actors are representations of the roles that interact with the system. At implementation level we rely on the two approved notions of components and classes.

EVOLUTION CONTROL POLICY

In this section we describe the issues of Synchronization, Evolution control and Version granularity.

Synchronization: Software Engineering involves many entities of different natures, including various models and multiple files. In addition, one must consider the fact that Software Engineers typically work on these entities simultaneously, be they model or file entities. Model transformations began to be used for maintaining consistency between model and software artifacts when the application code is fully derived from a model (Jacky *et al.*, 2009). In that case, users only work on the model, while artifacts are (re) generated when needed; in other words, the model is a high-level source code. In general, models do not contain enough details to be executable. This is why developers work on model and code at the same time. Usually, code skeletons are generated from the model, but the model cannot be reconstructed from artifacts and vice versa. Hence, we fall into synchronization issues where modifications on model and artifacts must be reconciled. Few MDE tools support permanent synchronization both ways between model and artifacts.

We maintain synchronization between artifacts with the help of interlinks information. Our assumption is that development model artifacts at different levels are interconnected with interlinks. If any entity which is in the set of interlink entities is modified, all its interlinked entities need to be considered for synchronization. For instance, in Fig. 5. The entity Submit Paper in the System model is an interlink entity with the Submit Paper of Test Model and Submit Paper of System Implementation. Thus any change in Submit Paper in the System Model requires synchronization with Submit Paper of Test Model and System Implementation.

Evolution policy at entity level: Evolution control refers to the criteria by which new versions of an object are created (Jacky *et al*., 2009). In most systems, only the versioning mechanism exists, while the evolution policy remains undefined and relies on the good will of developers. Depending on the change importance, versioning may mean:

- Update the object in the repository
- Create a new revision of the object
- Create a new object

The entities in our development models can be categorized as:

- Entities which have interlink dependencies
- Entities which has interlink dependencies
- Entities which has both inter- and intra-link dependencies. Based on these dependencies the evolution properties assigned to an Entity are:

  Evolution Properties = {Mutable, Immutable}

where, a Mutable entity is of type 2 and Immutable entity is of type 1 and 3. The evolution properties are based on Inter/intra dependency information rather than attributes of the entity as given in Jacky *et al*. (2009). We can define a mapping function $\mathcal{M}$: E $\rightarrow$ P, such that:

$$\text{Mutable if } E \in REL_M$$
$$\mathcal{M}(E) = \text{Immutable if } E \in CON$$

Based on the above information we can define our Evolution Policy as follows:

If an immutable entity is changed than a new version of the entity (or model) will be created in the repository. However, if a mutable entity is changed then entity (or model) will be updated in the repository. We can query a model entity to check its property i.e.,

M.e. get EP = (Mutable | Immutable)

Version granularity: Version granularity refers to the size of a version, whereas delta granularity refers to the size of those units in terms of which deltas are recorded. E.g. in SVN, CVS (2012) and Pilato (2004) version granularity (UOV) and delta granularity (UOC) are at the level of text files and text lines, respectively. In this case, the delta granularity is much finer than the version granularity. In case of class model, if class is both UOV and UOC then a conflict will be notify if two or more user edit the same class, even if they are working in different parts of the class (Murta *et al*., 2008).

There are three possibilities of version granularity in our approach:

- Versioning at Development Model level (Product Versioning)
- Versioning at Model level (Total Versioning)
- Versioning at Entity Level (Component Versioning)

Versioning at development model level: If the UOV is the development model, which in our case a unified model as we can see in Fig. 5 then the entire model has global versioning. In global or product versioning all elements of the model has unique version identity. In such a case completeness and consistency issues doesn't exist since we have a unified model in our workspace which is complete and consistent. A new version will be created based on evolution policy defined.

Versioning at model level: The second possibility is that we have the UOV at Model level i.e., each sub model has its own versioning information. In such a case the completeness issue is resolved since working on a sub model requires that complete sub model will be in the workspace. However, consistency issue needs to be resolved. To resolve the consistency issue we need synchronize the model entities with the rest of model entities, which can be done with the help of interlink information. This is a form of Total versioning.

Versioning at entity level: The third possibility is that the UOV is at entity level i.e., the each entity of a model has its own versioning information. If the entity has inter/intra dependencies with rest of the entities then the synchronization issue need to be resolved. The completeness and consistency issues can be resolved using a similar approach as in Jacky *et al*. (2009), (2010).

RESULTS AND DISCUSSION

Eduardo *et al*. (2009) presents an approach for SCM model developed for scientific workflows, which includes both version control and diff/merge algorithms. The focus of this work is on the development support characteristics of software configuration management. With this concern, a workflow definition can be stored in a large number of formats and can be composed via different strategies. Although there are a few Workflow Management Systems (WfMS) that only support textual definitions composed by hand, using a plain text editor, it has become a tendency in WfMS to support graphical interface. WfMS with graphical interface model a workflow as a directed graph. This graph representation simplifies the understanding of the workflow. Almost all WfMS with graphical interface store the workflow definition in a XML format.

In workflows, like in software, in order to support configuration management, it is necessary to have the following properties:

- A repository of workflows with access control, in which it is possible to store workflows and to register what are the stable and under development versions
- A mechanism to represent and store versions for the activities being used in a workflow composition
- The presence of the workspace concept to support the modeling of a workflow during both creation and maintenance phases. The workspace must also support the workflow publication in the repository

Two basic concurrency control mechanism are adopted during a check-out, change and check-in cycle. The first is the optimistic approach, in which the workflow is not locked and two or more users can modify it in parallel (Conradi, 1998). The second is the pessimist approach, in which the workflow is locked for commit and only the user that first performed check-out of the workflow can commit changes to its Conradi (1998). Nevertheless, other users can check-out the workflow, work on it in the workspace and wait until the locking user commits his work. Maximilian and Jonas (2009) and Maximilian (2008) present a SCM approach for software engineering artifacts that is able to manage change in graph-structured artifacts and supports traceability. The approach is based on operation-based deltas, change packages and product versioning. The approach is based on the claim that SE models are essentially graphs. The authors identified two different types of links in an integrated model, Intra-model links and Inter-model links. Intra-model links connect model elements within one model, such as a use case model. In a use case model a link from a use case to a participating actor is an intra-model link. Inter-model links connect model elements of different models. A link from a use case in the use case model to an open issue in the issue model is an inter-model link. Jacky *et al*. (2009, 2010) presents the solution for providing consistent support for model and code co-evolution. It is shown that it requires to:

- Define, what evolution policy is to be applied
- Closely synchronize ways, the model entities and the computer artifacts
- Enforce consistency constraints and evolution policies during the commit and check-out of both model elements and their corresponding artifacts. Traceability links can be defined between the model and the artifacts. These links translate the operations performed on the model to modifications performed on artifacts and vice-versa. It is possible, for example, to define that the concept of service defined in a Meta model should be mapped to an Eclipse java project with a specific structure and specific files (e.g., metadata information and templates). The synchronization ensures that each time a service is defined in a model the corresponding Eclipse project is created. Conversely, changes in some files (metadata) are translated into attributes and relationships in the model. To define an Evolution policy at an Entity level assigns properties to its attributes as Mutable, Immutable, Transient and Final. Compare to this approach, we in our approach address all these issues through interlinks. Dimitrios *et al*. (2009) provide an overview of the existing Model Matching approaches. Model differencing consists of three main steps i.e., identifying Match Elements, Different Elements and Visualization of the results.

  Matching approaches are categorized into:

- Static identity-based matching
- Signature-based matching
- Language-Specific Matching Algorithms

Static identity-based matching: The approach is based on Universally Unique Identifiers (UUID) assigned to model element upon its creation. Therefore, a basic approach for matching models is to identify matching model elements based on their corresponding identities. The main advantages are that these approaches are fast and require no configuration from the user perspective. The disadvantage is these approaches can’t be applied on Models constructed independently from different sources and Model representation technologies that do not support UUID.

Signature-based matching: The approaches fall in this category are based on defining a signature for model elements. A signature calculated dynamically from the values of elements features by means of a user-defined function. An advantage of these approaches is that it compares models that have been constructed independently of each other. The disadvantage is that configuration effort is required and developers need to specify a series of functions that calculate the identities of different types of model elements.

Language-specific matching algorithms: This category involves matching algorithms tailored to a particular modeling language such as UML. The advantage is that it can incorporate the semantics of the target language in order to provide more accurate results and also drastically reduce the search space too. For instance, when comparing UML models, two classes or data types with the same name always constitute a match regardless of their location in the package structure, while the same does not hold for other types of elements (such as parameters or operations). Similarly compare two operations if the classes they belong to are already known to match. The disadvantage is that it needs to specify the complete matching algorithm manually, which can be a particularly challenging task. While for previously discussed approaches developers need to spend little (e.g., provide a configuration or write signature generators) or no effort at all (e.g., identity matching).

CONCLUSION

In this study we presented an approach for SCM. Our approach is based on defining a Meta model for heterogeneous environment. We then specify that in such an environment there exist intra and interlinks between different model elements. We address the issues of synchronization, evolution control and version granularity. Our approach is based on identifying interlinks information between different model of a unified model developed during software lifecycle. By using these interlinks information we develop our evolution control policy and perform synchronization of models elements. We have shown that there are three possibilities of version granularity in our approach first, Versioning at Development Model level, second Versioning at Model level and third Versioning at Entity Level. Each possibility has its own pros and cons. Then we give an overview of the related in this area which enables us to compare our work with the rest of the approaches.

REFERENCES

Breu, R., J. Chimiak-Opoka and C. Lenz, 2007. A novel approach to model-based acceptance testing. Proceeding of the 4th MoDeVVa Workshop. Model-Driven Engineering, Verification and Validation, Nashville, TN, USA.

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

Conradi, B.W., 1998. Version models for software configuration management. ACM Computing Surveys, 30(2): 232-282.

Dimitrios, S.K., D.R. Davide, P. Alfonso and F.P. Richard, 2009. Different models for model matching: An analysis of approaches to support model differencing. Proceeding of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, IEEE Computer Society Washington, DC, USA.

Eduardo, O., R. Pablo, M. Leonardo, W. Claudia and M. Marta, 2009. Comparison and versioning of scientific workflows. Proceeding of the 2009 ICSE Workshop on Comparison and Versioning of Software Models, IEEE Computer Society Washington, DC, pp: 25-30.

Jacky, E., L. Thomas and V. German, 2009. Evolution control in MDE projects: Controlling model and code co-evolution. Proceeding of the 3rd IPM International Conference on Fundamentals of Software Engineering, Springer-Verlag Berlin, Heidelberg, pp: 431-438.

Jacky, E., L. Thomas and V. German, 2010. Defining and supporting evolution strategies for model driven software projects LIG-IMAG, 220, rue de la ChimieBP53,38041GrenobleCedex9,2010 France {Jacky.Estublier,Thomas.Leveque}@imag .fr

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, IEEE Computer Society, Washington, DC, USA.

Murta, L., C. Corrêa, J.G. Prudêncio and C. Werner, 2008. Towards odyssey-VCS 2: Improvements over a UML-based version control system. Proceeding of the International Workshop on Comparison and Versioning of Software Models, ACM, Leipzig, Germany, pp: 25-30.

Maximilian, K., 2008. TIME - tracking intra- and inter-model evolution. Proceeding of the Software Engineering Conference - Workshop, München, Germany.

Maximilian, K. and H. Jonas, 2009. Operation-based conflict detection and resolution. Proceeding of the ICSE Workshop on Comparison and Versioning of Software Models, IEEE Computer Society Washington, DC, pp: 43-48.

Maximilian, K., H. Markus, L. Yang, H. Jonas and D. Joern, 2010. Comparing state- and operation-based change tracking on models. Proceeding of the 14th IEEE International Enterprise Distributed Object Computing Conference, Washington, DC, USA, pp: 163-172,

Marcello, L.R., D. Marlon, U. Reina and M.D. Remco, 2012. Business Process Model Merging: An Approach to Business Process Consolidation. ACM Transactions on Software Engineering and Methodology (TOSEM), Retrieved from: http://eprints.qut.edu.au/38241/.

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

Yuehua, L., G. Jeff and J. Fr´ed´eric, 2007. DSMDiff: A differentiation tool for domain-specific models. Europ. J. Inform. Syst., 16(4): 349-361.