Towards Software Configuration Management for Unified Models

Maximilian Kögel
Technische Universität München, Department of Computer Science
Chair for Applied Software Engineering
Boltzmannstrasse 3, D-85748 Garching, Germany
koegel@in.tum.de

ABSTRACT

Change occurs throughout the software lifecycle. Software Configuration Management tools and techniques provide the foundation to effectively control change. With a growing number of approaches combining models from different domains into one unified, integrated model ([15], [12]), there is also an emerging demand for SCM techniques and methods that are able to support these unified models. Traditional SCM systems operating on the abstraction of a filesystem and managing change at the granularity of textual lines are not adequate for these requirements. We propose a novel approach to SCM for unified models combining product versioning, operation-based deltas and change packages. To demonstrate feasibility we have implemented our approach in Sysiphus a suite of tools for collaborating over Software Engineering artifacts represented in a unified model.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Model Comparison and Versioning, SCM*

General Terms

MANAGEMENT

Keywords

SCM, configuration management, versioning, operation-based, unified model

1. MOTIVATION

Change pervades the entire software life cycle. Requirements change when developers improve their understanding of the application domain, the system design changes with new technologies and design goals, the detailed design changes with the identification of new solution objects and the implementation changes as faults are discovered and repaired. These changes can affect every work product, from system models to source code and documentation. It is widely recognized that software configuration management (SCM) is crucial for maintaining consistency among while minimizing the risk and cost of changes to *all* of these artifacts [14]. We claim that Software Engineering models are essentially graphs. For many artefacts such as UML class or use case diagrams this is quite obvious, however even other artifacts such as release plans or even design decisions are graph-based. Wolf proposed such a model with graph-based meta model called Rational-based Unified Software Engineering model (RUSE) in [15]. In an integrated (unified) model we distinguish two different types of 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.

Support for managing change in a unified model with intra-model and inter-model links links essentially requires support for managing change in models with graph structure. The traditional SCM systems are geared towards supporting textual artifacts such as source code. Therefore changes are managed on a line-oriented level. In contrast, many software engineering artifacts are not managed on a line-oriented level and therefore a line-oriented change management is not adequate. For example adding an association between two classes in a UML class diagram is not line-oriented nor can the change be managed in a line-oriented way. A single structural change in the diagram will be managed as multiple line changes by traditional SCM systems. When using a traditional SCM system there are two approaches, the models are either versioned as a single configuration item or they can be split up into several files. The former approach inhibits any meaningful way of managing change. The most severe problem with this approach is that frequent merging will be required. In the latter approach the SCM system cannot manage valid configurations of model elements and their links, because the semantics of the model links are not represented in the version model [5]. Nguyen et al. describe this problem as an *impedance mismatch* between the flat textual data models of traditional SCM systems and graph-structured software engineering models. [8]. We conclude that the traditional SCM systems are inadequate for managing artifacts with graph structure and for supporting traceability.

This paper addresses the problem by proposing a novel approach for a software configuration management system

CVSM'08, May 17, 2008, Leipzig, Germany.

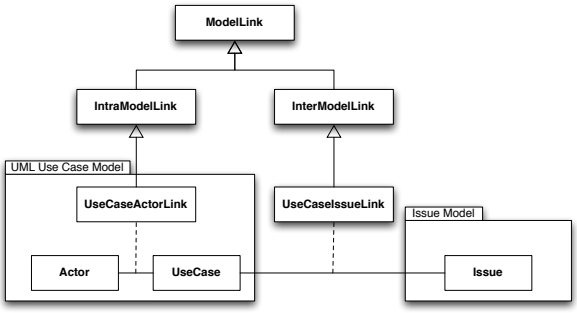

Figure 1: Intra- and Inter-Model Link taxonomy (UML class diagram)

for artifacts with graph structure featuring intra-model and inter-model links. It is based on the RUSE model, a unified model integrating software engineering artifacts from all development activities [15]. Furthermore our approach builds on operation-based deltas, change packages and product versioning. To demonstrate feasibility, we implemented the proposed software configuration management system in *Sysiphus* [1], a tool for collaborating over software engineering artifacts following the RUSE model.

The paper is organized as follows: In Section 2 we present a requirements analysis for key dimensions of a software configuration management system. In Section 3 we present our approach. Finally we evaluate the implementation of our approach in Section 4. A comparison with other research efforts was not manageable within the page limit of this paper, but is available on request from the author.

2. REQUIREMENTS ANALYSIS

The discussion and evaluation of various SCM techniques throughout this section largely follows Conradi and Westfechtel's *uniform version model* presented in [2, 3]. This framework provides a common terminology and classification the available alternative approaches for SCM system design can be uniformly expressed and compared with. We extended the categorization of the original framework by adding the aspect of delta representation since it is important for change management of graph-structured artifacts.

2.1 Delta Representation

Deltas can be represented using one of two basic approaches, *state based deltas* or *operation based deltas*. The differences between the two approaches can be very subtle in many SCM systems but are highly relevant. In the *state based approach* only the state representations of different versions are stored, possibly using compression or sharing of common parts. Deltas are reconstructed using a differencing algorithm that compares the different state representations. In the *operation based approach*, changes are described by using the original sequence of editor operations that caused the changes to represent the deltas.

With state based deltas, the semantic context of the original operations that caused the change has to be recalculated with the deltas which is expensive, time complexity is dependent on the project size and in some cases does not work at all [5]. For example, it can be impossible to unambiguously recalculate the original sequence of change operations when the changes of one operation are partially or completely masked by those of a later operation. This problem is of particular importance for systems where artifacts are internally represented using a meta model and a single change operation actually results in a sequence of meta-model transformations.

For such multi-layer data models with graph structure, the artifacts are often represented in a structured way, e.g. using XML, to preserve more contextual information to assist in the reconstruction of the original semantic context of the deltas. This approach is used in systems described in [10], [7], [11] and [5]. However, even this improved approach can not resolve all ambiguities and remains complicated. [5]

Storing the original editor operations automatically captures the original semantic context of the changes. Using the operation based approach, deltas can be recorded on the model layer. Several other research efforts have successfully employed the operation based approach in similar environments [13, 8, 9].

A drawback of the operations based approach is that the operations depend on the editors used, resulting in coupling the editor tools with the SCM engine. However, this can be resolved by defining a standardized language to express changes in the means of operations.

The editors have to support the recording of operations, which is usually not provided in systems with a simple interface to the SCM system, or in systems that have to support arbitrary editors. This is probably the main reason why operation based systems are not in widespread use today.

For the reasons mentioned above we suggest an operation based delta representation. It is important to note that operations need to fulfill two requirements in order to be usable in operation based deltas [6]: The operations have to be deterministically replayable in order to be used in forward deltas and furthermore the operations have to be reversible in order to be used in backward deltas.

2.2 Delta Granularity

Another important question regarding an SCM system is the granularity with which changes are described. This is called the *delta granularity*. In the RUSE model, changes occur on three different semantic layers of granularity as shown in Figure 2:

Logical Layer These changes are sets of logically coherent work as seen by the user, e.g. "I updated the use cases according to today's client review".

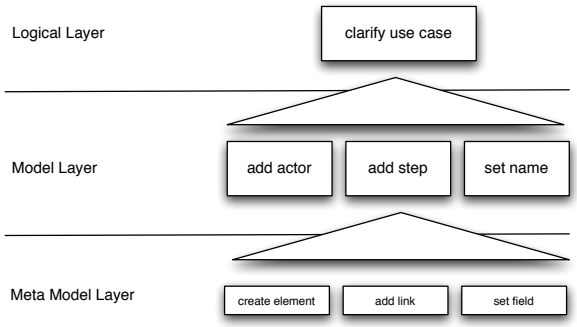

Figure 2: Three layers of change

Model Layer These changes are atomic changes as far as a specific model is concerned, e.g. "set a new initiating actor for a use case". They correspond to model specific operations on the model elements and are usually comprised of several changes on the meta model layer.

Meta Model Layer These are the changes as seen by the meta model layer. They change attribute values of single model elements. Users of the system are usually not aware of and do not understand this layer of change.

The SCM approach needs to be able to describe and track changes on all three layers of granularity. Change tracking can easily be achieved on the meta model layer since changes can be described here with the finest granularity of changes to single attributes of single model elements. Meta model change description has the additional benefit of being independent of the model layer. Therefore, changes should be described and tracked on the meta model layer.

Unfortunately, this alone is not sufficient since it does not capture enough context. A meta model change on its own will not be meaningful to a user of the system since he will be working on the semantic level of the model layer and generally not be aware of the mechanics of the meta model layer. Reconstructing the original model layer changes from a series of meta model changes is a difficult task [5]. Furthermore, operations on the model layer often do not have an injective mapping to the meta model, making an unambiguous reconstruction impossible even in theory. As an obvious example, the removal of an element in one part of the graph and the addition of a similar element in another part of the graph could be the result of a move operation as well as the result of a delete and add operation. Therefore, additional information needs to be provided by operations on the model layer to preserve their full semantic context.

An SCM system can automatically track and describe changes on the meta model and model layer, but not on the logical layer. Therefore, the SCM approach must provide a mechanism for manually grouping and describing logical changes. Our approach provides change packages with log messages to achieve this on the logical layer of granularity.

2.3 Version Granularity

Three possible approaches for *version granularity* are described in [2]: Component Versioning, Total Versioning and Product Versioning

Component Versioning lacks intrinsic support for managing consistent configurations, as every configuration item is in its own version space. However the inter-model and infra-model links in the RUSE model introduce dependencies among the model elements and therefore a configuration as a set of versions of model elements needs to be managed.

Total Versioning is not a big improvement in this respect since it still requires explicit management of consistent configurations. The model elements managed by the SCM system should completely and unambiguously describe exactly one system under development. Therefore, there is usually only one valid configuration at any point in time. Thus, the flexibility of being able to explicitly manage configurations is not needed and actually is a disadvantage by unnecessarily increasing the complexity of the system. Offloading this task of managing the configurations to the user would make using the system very difficult and error prone. Many problems of this approach are described in [8]. Furthermore, this would imply that the SCM engine needs to be able to handle the structure and semantic integrity constraints of configuration items and configurations. However the SCM engine should be kept as independent of the internal structure of the data model as possible.

Product Versioning lacks any modularity of the version space since it only has one uniform global version space for all configuration items. This non-modularity has its advantages and disadvantages. The main advantage is that version spaces of different configuration items are naturally related, alleviating the need to find combinations that produce valid configurations. Product versioning thus automatically provides consistent configurations without the need for explicit management of configurations or the SCM engine having to know about the exact nature of the data model. Furthermore, product versioning is a natural match with change packages since both approaches handle changes spanning multiple configuration items as a coherent entity.

One disadvantage of product versioning is that variants need to be global, too [3]. In our case this is not an important concern since varying single data model elements is seldom required as there is usually only one valid configuration at any point in time due to the many interdependencies among model elements. In cases where this is required, such variants can easily be provided by branching. Another drawback of product versioning is that no unique version or history exists per configuration item. For the former,

the last product version that changed the item can be used. For the latter, filters can easily be used that select only the changes for the specified elements when extracting history information.

Summarizing the analysis in this section, product versioning provides the version granularity most suitable for configuration management on graph-structured artifacts. We therefore use product versioning in our approach.

3. OUR APPROACH

In this section we present important aspects of our solution. We will first present our version object model and then go into detail about various other aspects of our approach.

3.1 Version Object Model

The version model resulting from the previous analysis is shown in a UML class diagram in Figure 3. The version model classes are shown with their dependencies on the data model.

The Version space is represented by a version tree graph structure consisting of branches, version nodes and edges for revision and variant relationships. The revision edges are associated with the change packages describing the changes between its two version nodes.

History The *History* class represents the history of a project. It provides operations for creating revisions, branches and tags and for accessing specific versions, differences and history information.

Branch The *Branch* class represents a branch of concurrent development in the version space and is composed of all versions on that branch.

Version *Versions* represent the nodes in the version graph. The state of the project at a specific version can either be represented explicitly by an instance of *Project Data* or implicitly by its position in the version graph and the appropriate deltas.

HistoryLink, VariantLink, RevisionLink The *HistoryLink* class and its subclasses represent the edges in the version tree graph. A revision relationship between two *Versions* is represented by the *RevisionLink* class. The changes that caused the revision are described by the associated *ChangePackage*. A variant relationship is represented by a *VariantLink*. Note that a *Version* can have at most one incoming and at most one outgoing *RevisionLink*. If it has no incoming *RevisionLink*, it is the initial version of a branch. If it has no outgoing *RevisionLink*, it is the head revision of that branch. A *Version* can have an arbitrary number of outgoing *VariantLinks* since it can have an arbitrary number of variants. However, a version can have at most one incoming *VariantLink*, in which case it has to be the initial revision of a new branch.

3.2 Change representation

In section 3.1 we did not show how the three layers of change introduced in section 2.1 are reflected in the version model.

Figure 4 shows the four classes for representing change on their appropriate layers. The *ChangePackage* class represents change on the logical layer. It is an ordered aggregation of all instances of *ModelOperation* that are generated between two commits of a workspace. As mentioned earlier grouping and describing change on the logical layer is up to the user by providing meaningful and descriptive log messages. Instances of *ModelOperation* are automatically captured by the implementations of the various model elements and their methods (e.g. setInitiatingActor(Actor actor)). These announce to the SCM engine whenever they begin and finish an operation. *ModelOperation* is part of an ordered composite pattern with *AbstractOperation* and *MetaModelOperation*. Thereby we obtain an ordered tree structure of instances of *ModelOperation* as inner nodes and with instances of *MetaModelOperation* as leaf nodes. This is useful for structuring change for visualization. Complex changes such as refactoring are more easily apprehensible in this way. The *MetaModelOperation* class represents changes on the meta model layer. An instance of it always reflects *exactly one change* affecting *exactly one model element*.

3.3 Deterministic replay and reverse of change

The requirements for delta representation discussed in section 2.1 imply that all instances of *ModelOperation* and *MetaModelOperation* have to be deterministically replayable and reversable. Instances of *ModelOperation* are replayed by replaying all their leaf nodes of *MetaModelOperation* in the order specified by the ordered tree structure as described in 3.2. They are reversed by reversing the specified order and reversing all instances of *MetaModelOperation*. Both deterministic replay and reverse for *ModelOperation* rely on deterministic replay and reverse of *MetaModelOperation*. Going into detail on the deterministic replay and reversibility of every *MetaModelOperation* subclass would fill the remainder of this paper, but the essence can be described more abstract. All subclasses of *MetaModelOperation* only change the value of exactly one model element in a deterministic way (e.g. no search and replace). Reversing instances of these subclasses of *MetaModelOperation* only requires to maintain the previous value in the instances of the *MetaModelOperation*. Swapping previous and new value will reverse the change.

3.4 Diffing and Merging

In our version model the differences between two versions can be described as the directed sequence of change operations transforming the source version into the target version. Since we use operation based deltas as described in Section 2.1, this information is readily available by traversing the version graph on a directed path from the source version to the target version. This path always exists since the version graph is a tree structure with the initial empty version at its root. The question concerning two way versus three way diffing no longer arises as the ambiguities occurring in two way diffing do not exist due to the additional information provided by the operations. When diffing with this approach a diff can contain many redundant operations that the user is not interested in, e.g. when a later change masks an earlier one. However by canonizing the sequence, we can eliminate redundant operations. For merging, we propose a diff-and-apply approach. In this approach a sequence of change operations that is derived by differencing as described above is applied to the state of the local workspace. Conflicts need to be detected and resolved as described below. We decided to use diff-and-apply merging since it is a very simple yet powerful approach that supports all types of merging use cases as described in [4].

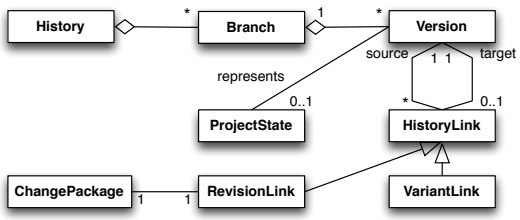

Figure 3: Version Model (UML class diagram)

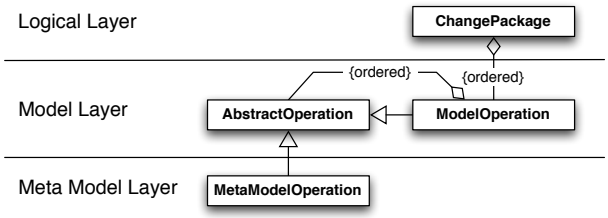

Figure 4: Three layers of change (UML class diagram)

3.5 Conflict Detection and Resolution

The checkout/modify/commit interaction model we use as workspace interaction model implies an optimistic concurrency control scheme. Therefore, conflicts can arise when synchronizing the workspace with the central repository. Another possible source of conflicts is the merging of changes into a workspace.

Closer inspection reveals two basic types of conflicts that have to be handled differently. The first type of conflicts is caused by concurrent change operations where the end result differs depending on the order of serialization of the operations. An example for this is the concurrent modification of the same element. These conflicts can be resolved by choosing a serialization or discarding one of the change operations. The second type of conflicts is caused by operations that can not be applied to the current state of the model or would violate the model's integrity when applied. An example for this type of conflict are modification operations to elements that no longer exist.

To detect conflicts we apply a very simple approach. Two change packages A and B conflict if A does contain meta model operations on model elements that are also touched by a meta model operation in B. If a conflict was detected, the system will need user assistance to resolve the conflict. Conflict resolution goes through two phases: deciding whether model layer operations should be accepted or rejected and, when all operations have been decided upon, merging the changes according to these decisions to produce a resulting list of operations.

In the first phase, whenever the user decides to include a model layer operation, the system proceeds as follows:

- Determine all model layer operations in the same list that are required for the selected operation and hold them in set A. More formally speaking, calculate the transitive closure of the *required* relationship on all model layer operations in the list of the selected operation. If a is the selected model layer operation and a *requires* b is true in the transitive closure, b is added to the set A.
- Determine all model layer operations in the other list that are conflicting with any operation in set A and save them in set B. Formally, we add a model layer operation b of the other list to B when the conflict detection strategy determines $a\ conflicts\ b$ for any a in B.
- Determine all model layer operations in the other list that are required for any operation of set B and add them to set B. Formally, we calculate the transitive closure on the *required* relationship on all model layer operations in the list not containing the selected operation. If for any operation b in set B and another operation a from the list not containing the selected operation, a *requires* b is true, then we add a to the set B.
- All model layer operations in set A will be marked as to be accepted, all model layer operations in set B as to be rejected.

In the second phase, the actual merge producing a resulting list of model layer operations takes place. All operations have been decided upon at the start of this phase. The system will merge the lists of operations as follows:

- Revert the workspace back to its base version but save all uncommitted changes in the workspace.
- Update the workspace to the version it was intended to be updated to in the beginning. Note that this will not always be the current head revision of the respective branch.
- All model layer operations that are marked as to be rejected in the non-negotiable list of changes of the repository are reversed and added to the result list in reverse order of appearance in time. This will undo all

model layer operations the user has decided to reject although they have already been committed.

- All model layer operations marked as to be accepted in the non-negotiable list of changes of the repository, can be ignored, they have already been committed earlier and they are already applied in the workspace also.
- All model layer operations marked as to be rejected in the negotiable list of uncommitted changes of the workspace, can be ignored, they will not be committed and have already been reverted in the workspace.
- All model layer operations marked as to be accepted in the negotiable list of uncommitted changes of the workspace, are applied to the workspace and added to the result list in order of appearance in time.
- The result list is defined as the list of uncommitted changes of the workspace.

Note that the result of the conflict resolution will not be committed automatically after the merging is completed. This gives the user the possibility to review the result of the merge on his model in detail before committing.

4. EVALUATION

To demonstrate the feasibility of our approach we implemented it for Sysiphus according to the presented key ideas. The time complexity for the retrieval of n changes from the repository (needed in update and view changes) is in O(n) for calculating the changes (they only have to be collected) and also O(n) for possibly applying the changes to a workspace. Canonizing the changes before sending for update operations ist still in time complexity O(n). When the changes are canonized before sending they have the theoretical minimum size when not taking the representation as Java objects into account. Please note that the time complexity for retrieving changes is independent from project size because of the operation based approach.

We have employed the SCM for Sysiphus in three student projects. The project size in number of elements was 200, 1200 and 2700. The number of changes (on the meta model layer) was 900, 7800 and 16900 respectively. The students reported increased productivity in evaluating change and awareness about change, apart from the obvious advantage of offline operation. The anecdotal evidence we collected shows that the approach is feasible and works in practice. We are currently evaluating the SCM in a student project with 40 students working on a problem in the area of airport logistics. Extensive data has already been collected and will hopefully help to improve our approach. We plan to extend the approach by improved conflict detection and visualization. Especially conflict detection will need further research since false positives heavily impact the productivity while merging.

5. REFERENCES

[1] B. Bruegge, A. H. Dutoit, and T. Wolf. Sysiphus: Enabling informal collaboration in global software development. In *Proceedings of the First International Conference on Global Software Engineering*, October 2006.

[2] R. Conradi and B. Westfechtel. Towards a uniform version model for software configuration management. In *ICSE '97: Proceedings of the SCM-7 Workshop on System Configuration Management*, pages 1–17, London, UK, 1997. Springer-Verlag.

[3] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, 1998.

[4] P. H. Feiler. Configuration management models in commercial environments. Technical report, Software Engineering Institute, Carnegie Mellon University, 1991.

[5] K. Letkeman. Comparing and merging uml models in ibm rational software architect. Technical report, Modeling Compare Support, IBM Rational, 2005.

[6] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, New York, NY, USA, 1992. ACM Press.

[7] A. Mehra, J. Grundy, and J. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 204–213, New York, NY, USA, 2005. ACM Press.

[8] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 215–224, New York, NY, USA, 2005. ACM Press.

[9] T. ODA and M. SAEKI. Meta-Modeling Based Version Control System for Software Diagrams. *IEICE Trans Inf Syst*, E89-D(4):1390–1402, 2006.

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

[11] H. Oliveira, L. Murta, and C. Werner. Odyssey-vcs: a flexible version control system for uml model elements. In *SCM '05: Proceedings of the 12th international workshop on Software configuration management*, pages 1–16, New York, NY, USA, 2005. ACM Press.

[12] I. Research. Jazz - innovation through collaboration, Jan. 2008.

[13] J. Rho and C. Wu. An efficient version model of software diagrams. In *APSEC '98: Proceedings of the Fifth Asia Pacific Software Engineering Conference*, page 236, Washington, DC, USA, 1998. IEEE Computer Society.

[14] T. View. IEEE Standard for Software Configuration Management Plans. *IEEE Std 828-2005 (Revision of IEEE Std 828-1998)*, pages 0_1–19, 2005.

[15] T. Wolf. *Rationale-based Unified Software Engineering Model*. Dissertation, Technische Universität München, July 2007.