TIME - Tracking Intra- and Inter-Model Evolution

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: Modern software development approaches, especially the model-driven approaches, heavily rely on the use of models during the whole development process. With the increasingly integrated tools these models converge into one unified model for various artifacts from various development activities. These unified models evolve over extended periods of time, which creates an emerging demand for versioning, comparing and merging them. Current approaches for Software Configuration Management (SCM) systems are not adequate for the special requirements of these integrated models. Most of them have been designed to manage changes in textual artifacts such as source code in a file system. Consequently they operate on the abstraction of a file system and represent change in a line-oriented way. Some of them have been developed as a solution for a single tool and can therefore not provide a seamless solution for an integrated model.

We propose TIME (Tracking Inter- and Intra-Model Evolution), a novel software configuration management approach for software engineering artifacts that is able to manage change in graph-structured artifacts and supports traceability. Our approach is based on operation-based deltas, change packages and product versioning. To demonstrate feasibility we implemented a TIME prototype in Sysiphus, a tool for collaborating over software engineering artifacts.

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 [Vie05]. We claim that Software Engineering models are essentially graphs. For many artefacts such as UML class or use case diagrams this seems to be very 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 [Wol07]. 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.

Integrating different tools, also implies integrating their models. Intra-model links are already part of the models, while inter-model links are added during integration. Only by adding inter-model links models from different tools can be set into relation and additional value is generated compared with having separate models. Inter-model links may be added for numerous reasons but a major motivation for adding them is traceability. Only by adding inter-model links elements from the previously isolated models can be traced to model elements from different models. As intra-model links are defined in the context of the respective tool they can be supported by the tools’ SCM capabilities if there are any. However this will create a media break when viewing models from different tools. In contrast the inter-model links are unknown to the tools, subject to integration, and are not supported by their SCM capabilities. So in summary for supporting the management of change in an integrated environment we need an integrated approach for SCM that is not limited to a single tool and intra-model links. Support for managing change in an integrated model such as RUSE 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. 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. [NMBT05]. 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 for integrated models such as RUSE featuring support for intra-model and inter-model links. Our approach builds on operation-based deltas, change packages and product versioning.

## 2 TIME

The discussion and evaluation of various SCM techniques throughout this section largely follows Conradi and Westfechtel’s uniform version model presented in [CW97, CW98]. 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. In the first three subsections we discuss three dimensions from the uniform version model. We present the different design alternatives and explain our choice. Finally we present the resulting version model based on the decisions from the previous subsections.
2.1 Delta Representation

Delta Representation deals with the representation of differences between two models at different points of time or at different variants. There are two basic approaches for Delta Representation: 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 contrast in the operation based approach, changes are described by using the original sequence of editor operations that caused the changes.

With state based deltas, the semantic context of the original operations that caused the change has to be recalculated with the deltas. This recalculation is expensive, time complexity is dependent on the project size and in some cases it does not work at all [Let05]. For example, it can be impossible if the changes of one operation are partially or completely masked by those of a later operation.

Storing the original editor operations automatically captures the original semantic context of the changes. Several other research efforts have successfully employed the operation based approach in similar environments [RW98, NMBT05, OS06]. A drawback of the operations based approach is that the operations have to be recorded while the models are manipulated. This will either result in coupling the editor tools with the SCM engine or in coupling a model layer underneath the editor with the SCM engine. However, this can be resolved by defining a standardized language to express changes in the means of operations.

For the reasons mentioned above we use an operation based delta representation in our approach.

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 an unified model, such as the RUSE model, changes occur on three different semantic layers of granularity.

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

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, basically
a graph. These changes represent simple modifications in the graph structure or node or edge attributes. 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 [Let05]. Furthermore it is nearly impossible in an integrated model like RUSE since it would require the SCM engine to have detailed knowledge of every model on the model layer. Additionally operations on the model layer often do not have an injective mapping to the meta model, making an unambiguous reconstruction impossible even in theory.

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. Changes on the model and meta model layer are automatically captured.

2.3 Version Granularity

Three possible approaches for version granularity are described in [CW97]: 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 intra-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. Many problems of this approach are described in [NMBT05].

Product Versioning lacks any modularity of the version space since it only has one uniform global version space for all configuration items. The main advantage of this approach 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. The most important disadvantage of product versioning is that variants need to be global, too [CW98]. In cases where this is required, such variants can easily be provided by branching.

In summary product versioning provides the version granularity most suitable for configuration management on graph-structured artifacts.

2.4 Version Object Model

The version model resulting from the previous analysis is shown in a UML class diagram in Figure 1. The version model classes are shown with their dependencies on the data model.

![Version Model (UML class diagram)](image)

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

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 ProjectState or implicitly by its position in the version graph and the appropriate deltas.

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 \textit{VariantLink}, in which case it has to be the initial revision of a new branch.

2.5 Change representation

In section 2.4 we did not show how the three layers of change introduced in section 2.1 are reflected in the version model.

![Three layers of change (UML class diagram)](image)

Figure 2: Three layers of change (UML class diagram)

Figure 2 shows the four classes for representing change on their appropriate layers. The \textit{ChangePackage} class represents change on the logical layer. It is an ordered aggregation of all instances of \textit{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 \textit{ModelOperation} are automatically captured by the implementations of the various model elements and their methods (e.g. \texttt{setInitiatingActor(Actor actor)}). \textit{ModelOperation} is part of an ordered composite pattern with \textit{AbstractOperation} and \textit{MetaModelOperation}. Thereby we obtain an ordered tree structure of instances of \textit{ModelOperation} as inner nodes and with instances of \textit{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 \textit{MetaModelOperation} class represents changes on the meta model layer. An instance of it always reflects \textit{exactly one change} affecting \textit{exactly one model element}.

3 Future work

To demonstrate feasibility, we implemented the proposed SCM system in \textit{Syspithus} [BDW06a], a tool for collaborating over software engineering artifacts following the RUSE model. 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.
References


