# Version Control in AToMPM

*Simon Van Mierlo*

## 1. Introduction

This document will attempt to devise a framework for version control in AToMPM. This framework will build on techniques used in the popular SVN version control system for version controlling projects written in a textual programming language. The main challenge consists of adapting these techniques to be used for model versioning, which is quite different from plain text file versioning.

The main idea of version control is to have a repository where all versions of the project's artifacts are stored. This makes it easy for the developer(s) to manage these artifacts and, for instance, find out where a bug was introduced, take a look at what changes were made by possible other developers, etc. Popular platforms include SVN, CVS and GIT. However, no platform exists that provides native support for managing versions of a project employing a model-driven engineering approach.

The document is structured as follows. In Section 2, current approaches to version control will be explained. In Section 3, the SVN platform will be examined more closely. Section 4 will explain how the SVN platform is used without modifications to accommodate model-driven engineering projects. Lastly, in Section 5, the techniques used in SVN will be projected onto AToMPM, the platform which was chosen to implement version control for model-based projects.

## 2. Current Approaches to Version Control

Version control is a useful technique, both in single- and multideveloper environments. Firstly, it allows a developer to query the history of the project's artifacts. Secondly, version control systems such as SVN implement functionality to aid collaboration when multiple developers work simultaneously on the project. In the next two subsections, these two facets of version control systems will be discussed.

### 2.1. History of a Project

During the lifetime of a project, artifacts are edited. It may be interesting to keep a *change history* of these artifacts, to be able to revert to previous versions when the need arises, examine the history and compute metrics, etc. In version control systems, all *versions* of all artifacts are stored. In a naïve implementation, when a new version of an artifact is stored in the *repository* of the project, it is saved as a full-text file. This would, however, require a large amount of disk space after a while, even for small projects. That is why version control systems keep track of the *changes* (or *deltas*) between different versions of artifacts. These deltas can either be computed by an algorithm which compares the two versions of the file, or be kept as a side-effect of the editing process: the deltas are then exactly those actions the developer performed to edit the artifact.

There are two methods (in SVN) to store these differences: either forward, or backward[1]. In the next two sections, these methods will be discussed in detail.

### 2.1.1. Fast Secure File System

In a Fast Secure File System (FSFS) repository, each revision of a file is represented as a delta or set of changes against an older version of the file[2]. The first revision is represented as a delta against the empty stream. To reconstruct a revision of a file, the file system code determines the chain of deltas leading back to revision 0, composes them all together using the delta combiner, and applies the resulting super-delta to the empty stream in order to reconstruct the file contents.

The history of an artifact may be represented as in the following diagram:

$v_0$ --$\Delta_1$--> $v_1$ --$\Delta_2$ --> $v_2$ --$\Delta_3$ --> $v_3$ --$\Delta_4$ --> $v_4$

Where each $\Delta_i$ is the set of changes which, when applied to $v_{i-1}$, results in $v_i$.

### 2.1.2. Berkeley DB

In the BDB back end, each revision of a file is represented as a delta against a newer revision of the file - the opposite of FSFS. The newest version of a file is stored in plain text. Whereas in FSFS, we add a new version of a file simply by picking a delta base and writing out a delta, in BDB the process is more complicated: we write out the new version of the file in plain text; then, after the commit is confirmed, we go back and "redeltify" one or more older versions of the file against the new one.

The history of an artifact may be represented as in the following diagram, which is the reverse of the FSFS one:

$v_0$ <--$\Delta_1$-- $v_1$ <--$\Delta_2$-- $v_2$ <--$\Delta_3$-- $v_3$ --$\Delta_4$ <-- $v_4$

Where each $\Delta_i$ is the set of changes which, when applied to $v_i$, results in $v_{i-1}$.

### 2.1.3. Optimizations

In FSFS, when the latest version of a file is requested, a super-delta has to combine all deltas starting from version 0. After a while, this may lower the performance significantly. That is why in FSFS, the delta base for a revision is not necessarily the previous revision. To choose the delta base for revision N, we write out N in binary and flip the rightmost bit whose value is 1. For instance, if we are storing 54, we write it out in binary as 110110, flip the last '1' bit to get 110100, and thus pick revision 52 of the file as the delta base.

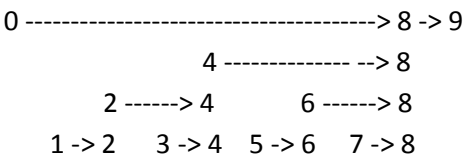
A file with ten versions (numbered 0-9) would have those versions represented as follows:

0 <- 1  2 <- 3  4 <- 5  6 <- 7
0 <------ 2  4 <------ 6
0 <---------- 4
0 <---------- 8 <- 9

[1] http://svnbook.red-bean.com/en/1.1/ch05.html
[2] http://svn.apache.org/repos/asf/subversion/trunk/notes/skip-deltas

This requires the computation of the deltas which don't have their predecessor as delta base. One way of doing this would be to compare the two versions of the file and compute a delta from the differences. Another consists of combining all deltas leading back to the predecessor we're looking for (for instance, for rev. 4, we combine deltas 4->3, 3->2 and 2->0). Note that this only leads to a performance increase if the resulting delta consists of fewer edit steps than the sum of the steps in the deltas leading up to the current version.

In BDB, the goal of the redeltification process is to produce the reverse of the FSFS diagram:




To accomplish this, we write out the version number in binary, count the number of trailing zeros, and redeltify that number of ancestor revisions plus 1. For instance, when we see revision 8, we write it out as "1000", note that there are three trailing 0s, and resolve to redeltify four ancestor revisions: the revisions one back, two back, four back, and eight back.

An optimization which could be implemented for both FSFS and BDB is *checkpointing*. This technique consists of saving the complete version of a file at regular intervals, resulting in increased disk usage, but decreased retrieval of versions.

## 2.2. Multi-User Environment

We now know how versions of an artifact can be efficiently stored. But until now, we have not considered the presence of multiple developers, possibly editing the same artifacts at the same time. A version control system not only has to offer functionality to store and query versions of a project and its artifacts, but also offer functionality to deal with concurrent editing of these artifacts. If no such techniques are in place (i.e. each developer is allowed to make changes to any file, without restrictions), developers might overwrite changes someone else made without knowing it. In the next two subsections, two approaches to version control[3] are explained. Note that I will not discuss branching, which I consider future work.

[3] http://svnbook.red-bean.com/en/1.7/svn.basic.version-control-basics.html

### 2.2.1. File Locking

In the file locking approach, only one person gets to edit a file on the repository at the same time. If a developer wants to edit a file, he requests a lock on that file. Having acquired the lock, he can edit the file without anyone else interfering. The main disadvantage is that other developers might also want to work on the file simultaneously, and the changes they will make are most probably not on the same part of the file, which means that they should be allowed to do so. On top of that, if the first developer fails to remember to release the lock, the file might be locked until an administrator unlocks it. As such, file locking approaches are rarely used.

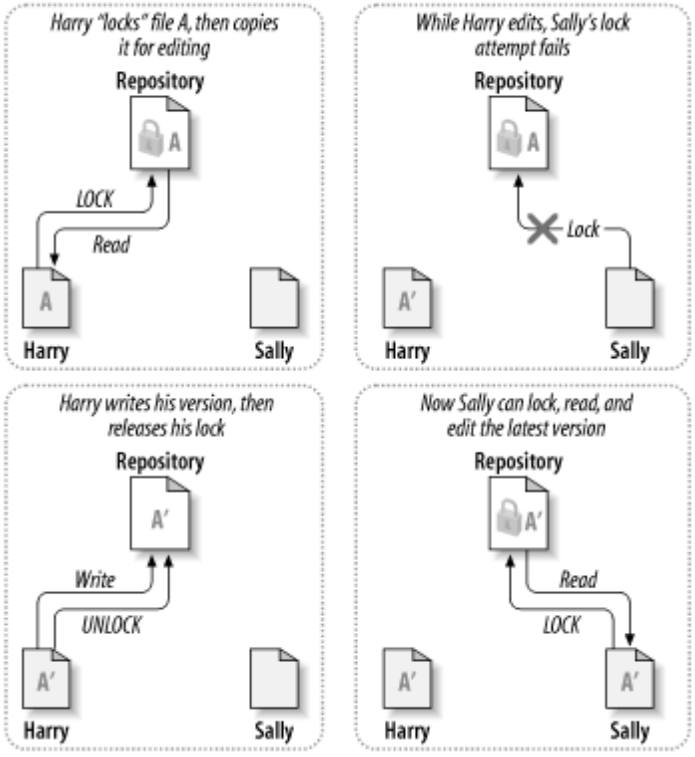



Figure 1: The Lock-Modify-Unlock Solution[3]

### 2.2.2. The Copy-Modify-Merge Solution

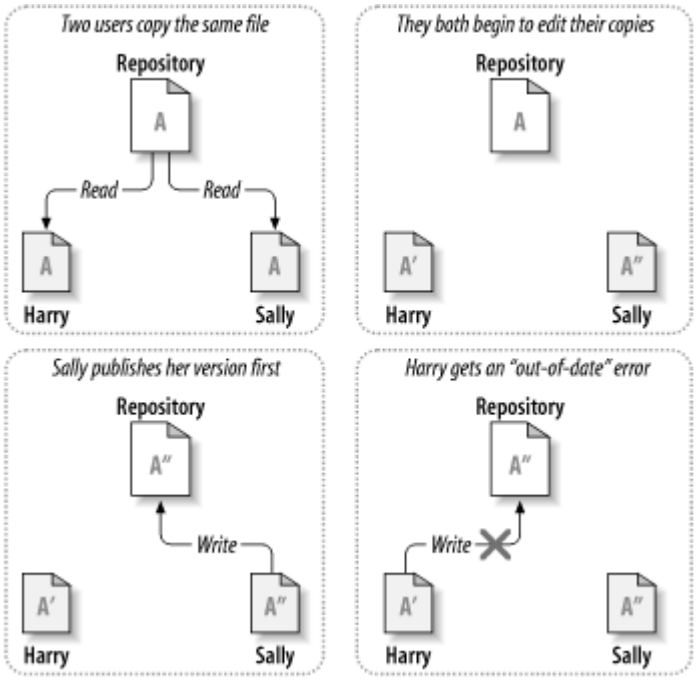



Figure 2: The Copy-Modify-Merge Solution[3]

In this solution, every developer has a local 'working' copy of the project. Each developer edits his version of the project, which are then merged together by the version control system. In SVN, when a developer tries to commit his changes (which is a transaction with ACID properties), the version control system checks whether his local copy of each file is the same as that on the server. When it is not, he gets an error which tells him to perform an update. The updated copy on the server is then merged with his local copy. This happens automatically or manually when conflicts are detected (for instance, if Sally in Figure 2 changed the return type of a method 'a' to 'int' and Harry did the same but changed it to 'string', a conflict is found). The merged copy can then be committed to the repository.

# 3. Using SVN for Model-Based Projects

Models are, such as all project artifacts, persisted to disk. Several formats are used to represent the file on disk: XML, JSON, … These files can be version-controlled by a system such as SVN. However, there are certain key differences between models, which have a graph-like data structure, and flat files written in a programming language. A file written in a programming language only has one representation: there is no choice where to put functions, class declarations, etc., because the programmer chooses each element's position in the file while editing it. However, a model has a possible infinite number of representations. The graph structure can be represented in XML or JSON, but two identical graphs may have two very different representations, which entirely depends on the algorithm to build that representation. In theory it would be possible to write out the *canonical version* of a graph, which is unique. In practice, this has proven to be very complex. As such, there is a need for *native* version control support for models. The next section will describe an approach to building a version control system in the research tool AToMPM.

# 4. From SVN to AToMPM

In this section, the techniques used in SVN will be projected on AToMPM, resulting in a version control system for model-based projects. In the next section, the Create-Read-Update-Delete (CRUD) operations which can be performed on models are listed. The 'deltas' to construct one version of a model from another will necessarily consist of these operations.

## 4.1. Current CRUD Operations in AToMPM

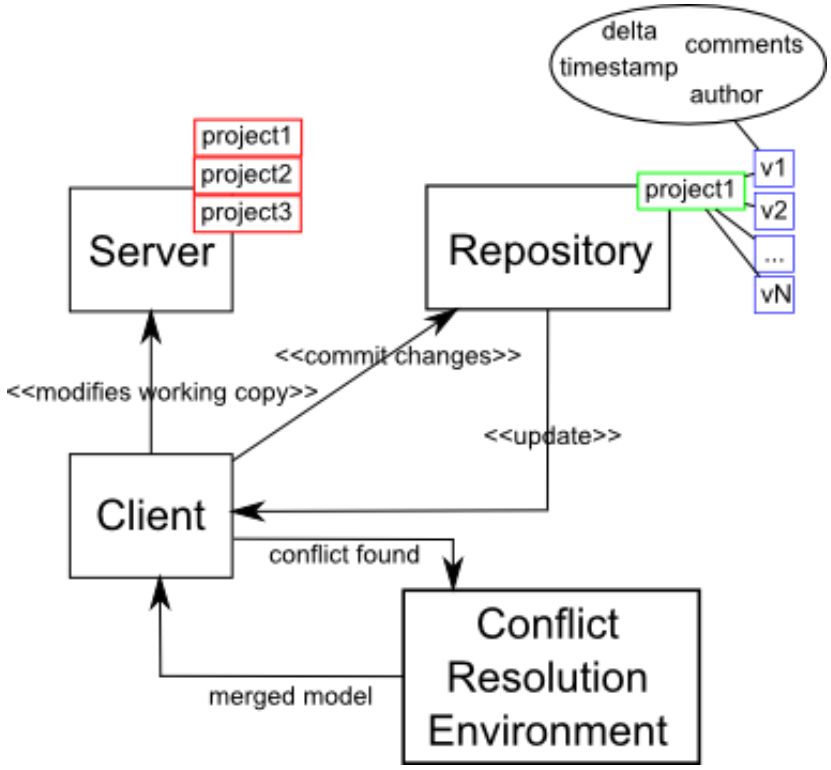
A 'delta' always exists of a number of CRUD operations, which are atomic operations. The CRUD operations present in AToMPM, together with their inverse, are listed below.

- RMNODE <-> MKNODE
- RMEDGE <-> MKEDGE
- CHATTR <-> CHATTR
- LOADMM <-> DUMPMM
- LOADASMM <-> DUMPMM

Currently, all changes performed by the modeler are logged to a journal. This journal is then used to support undoing and redoing of changes. At certain points, for instance when a batch edit is performed, checkpoints are inserted into the journal. These checkpoints can then be used to undo the effects of the batch edit. These techniques can be held in place to edit the local, working copy and reused for the purpose of version control.

## 4.2. Envisioned Approach

The envisioned approach would leave the current implementation of the client-server architecture intact. However, every time a model is saved, the set of changes performed is saved alongside it (note: when a 'string' or 'code' property of an element in the model is changed, we could employ textual differencing techniques to discover the changes to the string, instead of seeing it just as a CHATTR operation and saving the whole new string). Then, once the developer is satisfied with his work, he can commit it to a repository. This repository receives the set of changes performed on the models and does the same as SVN: it checks whether the modeler had the latest version of the model. If not, it asks the user to perform an update and, if needed, resolve merge conflicts. The update which the client receives from the repository is a delta consisting of those CRUD operations that result in the latest version of the model when applied to the local copy of the model. Then, he can perform the commit, saving the changes to the repository. The algorithm of choice is the forward strategy as used in the FSFS system, for its ease of development.




To implement version control in AToMPM, two elements will need to be added: a repository and a conflict resolution environment. On top of that, a version controlled project will have to keep track of all changes performed to the files (until the files are committed). The repository will have to make sure a commit by the client has the ACID (Atomicity, Consistency, Isolation, Durability) properties.