Integrated Version Control in the
COOP/Orm Version Server

Ulf Asklund

LU-CS-TR:96-175

Also published in: Proceedings of NWPER'96, Nordic Workshop on
Programming Environment Research, L. Bendix, K. Nørmark,
and K. Østerbye (Eds), Aalborg, Denmark, May 1996.
Aalborg University, Technical Report R-96-2019

Department of Computer Science
Lund Institute of Technology
Lund University

P.O. Box 118, S-221 00 Lund
Sweden

Integrated Version Control in the COOP/Orm Version Server

Ulf Asklund

Dept. of Computer Science, Lund Institute of Technology
Box 118, S-221 00 Lund, Sweden
e-mail: Ulf.Asklund @ dna.lth.se

Position paper: This position paper describes some of the design decisions made during the development of the prototype COOP/Orm. The prototype is build to illustrate an editing model supporting distributed groups of developers working together. The model is based on hierarchical structured documents and fine grained version control. Editors as well as the database, storing the documents, understand (and maintain) versions. This position paper describes the client-server architecture of COOP/Orm, how the knowledge of versions is integrated into the server and how this affect the client-server protocol. Especially the need for a *version tube* is discussed.

1 Introduction

In our work we design a document model supporting distributed teams of programmers working together on the same system. The model we present avoids the problems that occur when documents are shared between team members, and solves the problems of how team members can be aware of each others work. We have also build a prototype which uses the model. During the development of this prototype, called COOP/Orm, we have come across many interesting design issues. Many of these are purely technical, but some of them are tightly connected to the problem we try to solve. The design decisions made in order to integrate versions into the system are particular interesting.

This position paper is a short description of the architecture of COOP/Orm and of the decisions mentioned above. Especially the interaction between the client and the server using version numbers as parameters and the role of a data structure called 'version tube', are described. The next section gives a short overview of the COOP/Orm environment, the user interface and the main functionality. Section 3 describes the architecture and our design decisions. It also introduces the term version tube and motivate its existence. A short summary concludes the paper.

2 The COOP/Orm environment

COOP/Orm is an environment supporting collaborative writing of hierarchically structured documents. It is based on previous work in the Mjølner-project [KLMM93], concerning object-oriented software development. The environment developed in Lund, Mjølner Orm [MMH+90] supports collaborative software development to a limited extent through its configuration management [Gus90]. The aim of the current project is to focus on supporting teams working on the same system, and work has been done to both integrate version and configuration control into the environment [MA96].

Version control is part of our fundamental storage model, which means that all processors, including editors, must understand (and maintain) versions and deltas. No locking is used to synchronize parallel work, instead an optimistic check-out approach resulting in alternatives of the document is used. A merge operation supports both textual and structural changes and automatically creates a suggestion using a set of default rules [Ask94]. Potential conflicts are detected and clearly marked, making it simple for the user to find and edit them in the merged version. The hierarchical structure helps in reducing the number of conflicts on the textual level. If the document is a book, for example, it is divided into chapters, sections and so forth. This makes each individual text small, and changes developers making in the same text less likely.

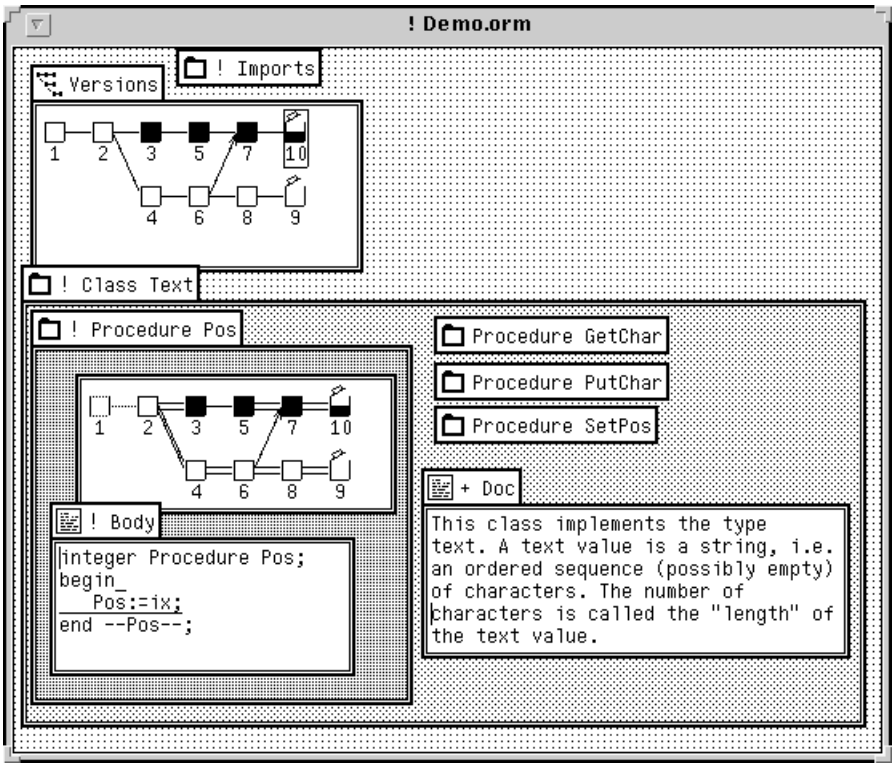


Figure 1 *Snapshot of the COOP/Orm client.*

Figure 1 depicts the user interface of the COOP/Orm client. We have in this example opened the document Demo.orm. In the lower part the document contents is shown, containing hierarchically structured text (here a class containing its procedures and documentation). The hierarchy, which has an unlimited number of levels, is presented using nested windows and a structure editor allows the user to create new (or delete) folders and leaf nodes. The text in each leaf node is edited using an integrated text editor.

Above, in the window called 'Versions', the version graph of the document is visualized. Each numbered square represents a version, and the lines between them the revision-of relation. In this example we can see that there has been parallel development, started from version 2, that were later merged into version 7. We can also see that the parallel work is continued in both alternatives after the merge.

The version graph is also used for quick and simple interaction, such as changing which version of the document to view. By clicking directly in the graph versions are selected and the view of the document is instantly changed to reflect the selected version. By selecting a range of versions the differences between two versions are highlighted (henceforth we call these two versions Viewed and Compare). In Figure 1 version 10 is viewed and deltas between version 3 and 10 are highlighted in the document contents.

At the structure level the differences are marked with the signs +, -, and !, meaning added, deleted, and changed respectively. A node is considered changed when either the contents of the node is changed, or if the node is a folder, any son is changed. This works recursively down the hierarchy and results in change-marks propagated up the hierarchy when a leaf node is changed, making it very easy to find the changed places. In the leaf nodes changes are marked as underlined or overstroken, marking added and deleted text respectively.

3 The COOP/Orm architecture

COOP/Orm has a client-server architecture in order to cope with the primary requirements: distribution and synchronization between clients. We have taken the distribution requirement one step further so we also have server-server communication to handle distribution over wide area networks. The server-server protocol is, however, not further elaborated in this paper. One decision necessary to make when designing a client-server architecture is how general (or the other way around, how specific) the server and the client-server protocol should be. The hierarchical structure and the integrated version control are in our model fundamental, and are therefore known by all parts of the system, including the server. This also means that version numbers and addresses of specific nodes in the document hierarchy are included in the client-server protocol. Figure 2 depicts the client-server architecture. The tree in the client reflects the hierarchical structure of a specific version of a document. This hierarchy is visualized by nested windows in the user interface. Attached to each node is the

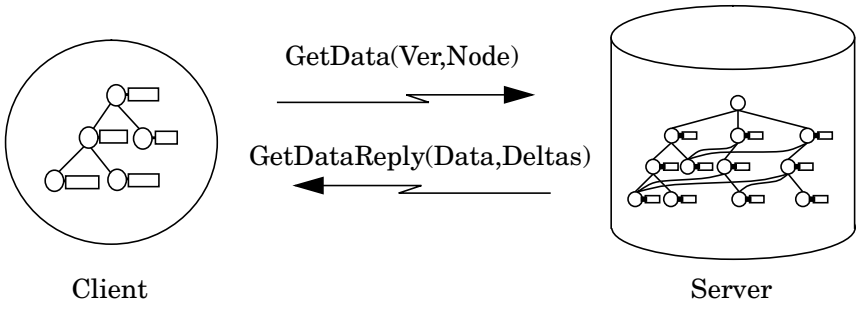


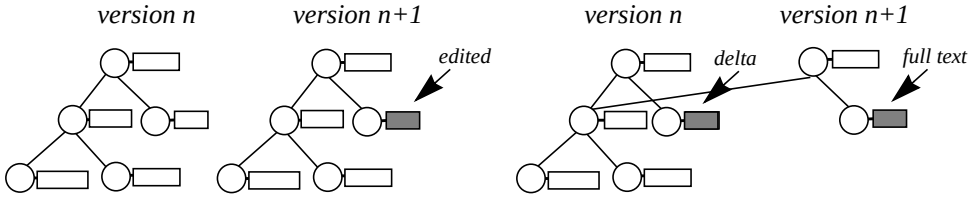
Figure 2 *Client-Server architecture*

text presented in the windows, handled by the integrated editor. The more complicated structure in the server stores all the versions of the document. Or more precisely all the changes made to the document, making it possible to retrieve any version of it. Therefore, a client's request for node data, must both include the node address and the version of the document required. The example in the figure shows such a command requesting data.

Server storage

One question that arises when all versions of a document are stored is the size of the document file. In COOP/Orm, unchanged parts of the document are shared between versions rather than copied. Figure 3a depicts the conceptual model of two versions of a document, here with one leaf node changed in the younger version, n+1. In our model, all nodes not changed are shared as depicted in Figure 3b. In changed nodes, the contents is stored using delta-technique. The (new) full text in the changed node is stored in the edited version (n+1). The backward delta (i.e. the delta required together with the new full text to rebuild the old text) is stored in the preceding version (n), replacing the previously stored full (old) text. If yet another version (n+2) is created and the same node is changed, the same procedure results in that the new text is stored (full) in version n+2, and the new backward delta is stored in version n+1. I.e. to rebuild version n of this node, the full text and both deltas now are required.

An editor in our model must understand and support versions and deltas. When a node is changed both the new full text and the delta are constructed by the editor and sent to the server. This in contrast to systems where only the new full text is sent, and then the delta is calculated in the server, using diffing algorithms to find out what was really changed. In fact we do not use the delta tech-



(a) Conceptual view (b) Sharing node versions

Figure 3 *Sharing of node versions*

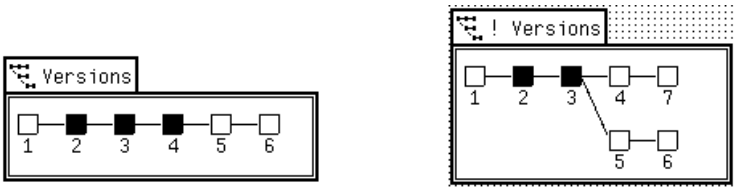
nique primarily to save disk space, but to be able to present accurate deltas between versions to the user.

More details about the server database representation and the integrated editor can be found in [MAM93] and in [Ols96].

Server reply
In the example in Figure 2 the client requested node data from the server. As response to that request the server returns the full text stored in the latest changed version, and all the deltas needed to rebuild the requested version of the node. Note that the server, by itself, finds out in which version the full text is stored and the deltas that are needed. This is an example of how the server understands both versions and the delta technique used. The contents of the data and the deltas themselves are, however, only understood by the editor.

Scalability
One very important issue easy to forget when building a prototype is to make it scalable, i.e. making not only small examples possible but real industry projects as well. In a large document with a long history consisting of many versions, retrieving an old version could potentially be a timeconsuming operation. In the situation with the user often changing the viewed version and frequently comparing it to other versions (which is possible in our model), long delays are not acceptable. In order to scale up to handle large documents with many version it is important to reduce the communication between the client and the server. In connection to data retrieval COOP/Orm 'optimize' the communication in two dimensions, both in space and time:

- lazy data and delta retrieval. The hierarchical structure is utilized to limit the data retrieved when, for example, a new version of the document is viewed. Only data for open windows are retrieved, i.e. if a window, in the user interface, is iconized and consequently there is no need for data for the corresponding node, no data is retrieved. The request is delayed and instead made on demand when windows are opened. In this way complete subtrees can be loaded lazy, which makes the number of nodes that retrieve data proportional to the number of open windows and not to the document size.
- caching delta in the client. Attached to each delta returned from the server are two version numbers (specifying the link between these two versions) telling the client where the delta belongs. This makes it possible for the client to treat each delta individually, which in turn makes it possible to incrementally retrieve deltas for a node instead of reload the entire history every time. Figure 4a depicts a version graph for a document where version 4 is viewed and Compared is set (differences are highlighted) to version 2. If Compare in this situation is moved to version 1, only the delta between version 1 and 2 is retrieved (and only for the visible nodes). The new delta is then, in the presentation, concatenated to the already presented. If instead Compare is moved to version 3 no deltas are needed at all, but the delta between 2 and 3 can be dropped and is not shown anymore.



(a) single development line (b) branching into two alternatives

Figure 4 *Versiongraphs depicting different Viewed and Compare situations*

Version tube

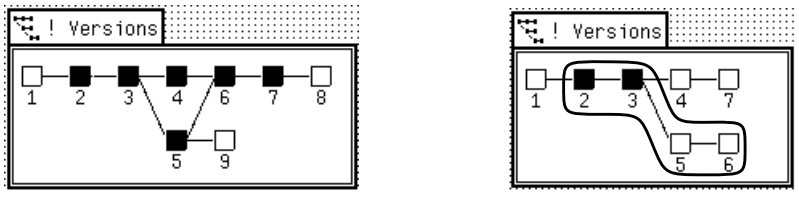
The two well motivated 'optimizations' mentioned above, together with support for parallel development and merge, made it necessary to extend the client-server protocol with more elaborate version information. It is not always enough to only give a particular version, but the path to a particular end version must be known. We call the data structure to specify such a path for a 'version tube'.

Figure 4a depicts a version graph with Viewed set to version 4 and Compare set to version 2. This means that for all open windows full data of version 6, deltas to rebuild version 4, as well as deltas between version 2 and version 4 to highlight differences, are loaded. If a window is opened and new data and deltas are needed the data request will return the required data without any problem. This example is simple to handle because in the version history without branches there is no ambiguity of which delta to retrieve.

The situation gets, however, more complicated when the development branches into parallel alternatives. Figure 4b depicts a situation where a version older than the split version is viewed. If a window is opened in this situation, the question arises: from which alternative should data and deltas be retrieved to rebuild version 3? The already loaded nodes got their data from one of the alternatives (7 or 6), but which one depends on how the user came to this situation.

A similar problem in the same example arises if Viewed is moved to, for example, version 5. If this alternative was used rebuilding version 3 no deltas are needed at all, but if the other alternative was used, both data and delta rebuilding version 3 using the new alternative instead, must be retrieved. Thus, the questions are, which alternative did we get the data from, and can we retrieve delta from the same alternative again?

An underlaying assumption is that we do not want to create a situation with data and deltas from different alternatives for different nodes, since this should increase the complexity of the caching delta-algorithms mentioned above.



(a) Are deltas already retrieved between version 3 and 5? (b) A version tube

Figure 5 *Situations occuring when merge have been done*

Another situation depicted in figure 5a occurs when Compare and Viewed spans a 'split-merge'. Both loading a node (as a consequence of opening a window), or moving Compare to version 4 or 5 gives the client the same situation of not knowing which alternative to retrieve deltas from or if they already are retrieved.

In COOP/Orm the requirements addressed by the examples above are implemented using the 'version tube', which is a data structure representing a path through the version graph. A tube always starts from the Compare version, runs through Viewed, and ends in the last version of an alternative, see Figure 5b. Each client creates its own tube when the first data is requested and uses it to remember the way deltas have been retrieved. To guarantee delta to be retrieved the same way again when new windows are opened, the version tube is used as parameter to the data request operation. The server still must support the versions and the delta technique, but is now limited to the sequential version history in the tube when it searches for data and delta. The same example as depicted in Figure 4b, but now with a version tube defined, is depicted in Figure 5b. The tube drawn in the figure symbolize how the client remembers from which alternative deltas were retrieved. If Viewed in this situation is moved to version 5 the client knows that the deltas needed for the new situation already are loaded and no deltas are requested.

4 Summary

We have presented a data structure called a *version tube* as one technique meeting the new requirements raised when integrating fine-grained version control into a database server. We have shown how, well motivated, optimizations together with the possibility to merge parallel development lines leads to problems that can be solved by this data structure.

The presented technique is however not the only solution to the problem. The presentation in this paper is based on the requirement that all nodes in a document must retrieve their deltas from the same alternative, i.e. using the same version tube. If this requirement is given up, it also removes the need to force the server to follow a certain path when a node is loaded. Thus, is the 'version tube'-extension of the client-server protocol not needed. The drawback of this solution, however, is that the optimization in time (retrieving deltas incrementally) must be done for each node instead of for the document. I.e. the client must have one version tube for each document node, instead of one for the entire document.

The contribution of this paper is, however, not to find all possible trade-offs, but to show how an advanced editing model adds new requirements to the server, and how these requirements are fulfilled in the COOP/Orm prototype.

References

[Ask94] Ulf Asklund. Identifying Conflicts During Structural Merge. In Boris Magnusson, Görel Hedin, and Sten Minör, editors, *Proceedings of the Nordic Workshop on Programming Environment Research*, Lund University of Technology. LU-CS-TR:94-127, Lund, January 1-3 1994.

[Gus90] A. Gustavsson. *Software Configuration Management in an Integrated Environment*. Licentiate thesis, Lund University, Dept. of Computer Science, Lund, Sweden, 1990.

[KLMM93] J.L. Knudsen, M. Löfgren, O. L. Madsen, and B. Magnusson, editors. *Object-Oriented Environments - The Mjølner Approach*. Prentice-Hall, 1993.

[MA96] Boris Magnusson and Ulf Asklund. Fine-Grained Version Control of Configurations in COOP/Orm. *In Proceedings of the Sixth International Workshop on Software Configuration Management* (Ed. I. Sommerville), Berlin, 25-26 March 1996.

[MAM93] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-Grained Revision Control for Collaborative Software Development. In *Proceedings of ACM SIGSOFT'93 - Symposium on the Foundations of Software Engineering*, Los Angeles, California, 7-10 December 1993.

[MM93] Sten Minör and Boris Magnusson. A Model for Semi-(a)Synchronous Collaborative Editing. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work*, Milano, Italy, 1993. Kluwer Academic Publishers.

[MMH+90] Boris Magnusson, Sten Minör, Görel Hedin, Mats Bengtsson, Lars-Ove Dahlin, Göran Fries, Anders Gustavsson, Dan Oscarsson, and Magnus Taube. An Overview of the Mjølner Orm Environment. In J. Bezivin et al., editors, *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, Paris, June 1990. Angkor.

[Ols96] Torsten Olsson. A View of A Merge. *In proceedings of the Nordic Workshop on Programming Environment Research*, Aalborg, Denmark, May 29-31 1996.