Identifying Conflicts During Structural Merge

Ulf Asklund

LU-CS-TR:94-130

Also presented at: Nordic Workshop on Programming Environment Research, Lund, Sweden, June 1-3 1994

Department of Computer Science
Lund Institute of Technology
Lund University

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

Identifying Conflicts During Structural Merge

Ulf Asklund

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

Abstract. This paper presents a model for controlling the evolution of documents concurrently developed by teams of authors. Optimistic check-out of revisions and alternatives, and hierarchic merge making use of default rules is presented. In particular the different situations occurring during a merge of parallel development lines and the benefit of storing the full evolution history is discussed.

1 Introduction

To cooperate have always been hard. Now when the cooperating persons may be spread all over the world using world-wide networks, support for people working together is even more urgent. Different systems using different approaches have been constructed to reduce some of the problems, and the distinction between the four cases same/different place an/or same/different time have been made [EGR91]. We have in our system concentrated upon the two cases with different place, independent of time. Specifically we have tried to make it possible to, in an intuitive way, provide a smooth transition from different time to same time [MM93].
Revision handling is fundamental for systems with many users, storing the history of a document's evolution. Using a single evolution line involves file-locking to prevent many persons to modify the same file at the same time. To avoid persons being blocked out by file-locking we use an optimistic check-out approach, i.e. never locking a file for creating a new revision of it, resulting in several alternative revisions as a consequence. During parallel development each person involved in the cooperation makes changes in his own revision and can thus never be disturbed by changes made by some one else. The drawback of this approach is the fact that changes made by several persons have to be merged. To reduce this problem we use a fine-grained revision control approach, making the fragments changed as small as possible. We also support a technique of keeping the persons cooperating aware of each others changes. In this way we expect unintentionally made conflicting changes to be minimized, which make the future merge simpler.

The system referred to in the paper has been developed as a part of an ongoing project on collaborative software development environments. It is based on previous work in the Mjølner project, a project on object-oriented software development [KLMM93]. The environment developed in Lund, Mjølner Orm [MMH+90], supports collaborative software development to a limited extent

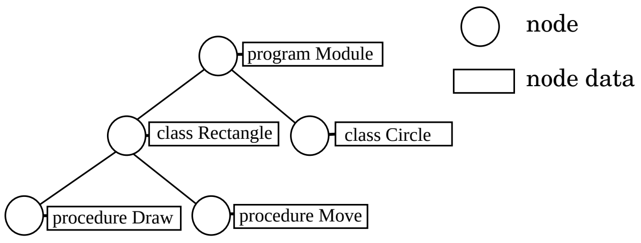


Figure 1 *A hierarchically structured program*

through its configuration management [Gus90]. The aim of the current project is to support teams of programmers working on the same system.

This paper will focus on the merge situation, what information the system can provide for the user, and how it is retrieved from the internal structure of our system. In section 2 a general description of our model is given. Creation of different kinds of revisions, and then especially the merged revision, are discussed in section 3. Section 4 contains a more in-depth description of our internal representation, supporting the requirements requested in previous sections.

2 Our Model

Our model is based on hierarchically structured documents. Typically a program is structured in blocks, classes, procedures, and functions. A paper, for instance, in chapters, sections, and paragraphs. In figure 1 an example of such a structure is shown. The *nodes*, in this example program fragments, are structured hierarchically with a module containing classes, and a class containing procedures. Connected to every node is the *node data*, containing the information stored in the node. In the example of program fragments it is program code but it can be anything stored as a string of bytes. Thus it is in the node data the contents of the document is stored and the nodes are building the structure of the document.

When a new revision of the hierarchical document is made the document is copied. Edited node data is stored as full data in the new copy while backward deltas are stored in the original revision node data as depicted in figure 2. In our system we have an integrated editor recording operation-based differences as deltas [Ols94]. These change-oriented deltas seems, as described by Lippe and Oestrom in [LvO92] and by Lie et al. in [ARMEA89], to have a much larger potential for supporting merging of variants than state-based deltas calculated from edited files, like in RCS [Tic85] and SCCS [Roe75].

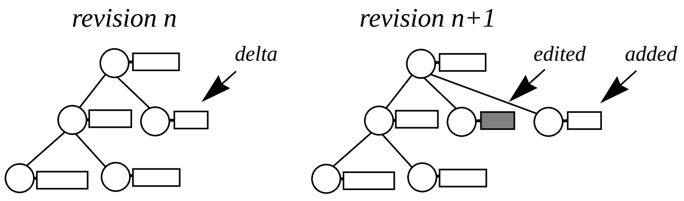


Figure 2 *Two revisions of a document*

The system is organized as a client-server architecture. A database server stores the documents and synchronizes the access from the client applications run by the users. It does not do any interpretation of the node data, all data is handled as a string of bytes. The system is integrated where the client application must understand the revision control mechanisms and the server knows about the hierarchical structure. An editor, for example, must be aware of which revision of the document is edited and must supply deltas as the document is edited. The editor must also have functionality to re-construct an old revision from these deltas. The server, on the other hand, has operations to add/delete nodes and is able to "follow" the history of a node, and manage to store and retrieve deltas corresponding to that node.

As mentioned in the introduction we want to avoid locking. The experience from people working with optimistic check-out technique in traditional environments is that conflicts are rare. In 90% of the cases the same file has not even been touched by more than one person [Kan93]. Our approach is that the system must not, at any time, prevent the user from creating a new revision. If a programmer wants to fix a bug or add a new function he creates a new revision where he makes his changes. No locking of the document is done. If another programmer also wants to add something to the same revision of the document, he creates a new revision as well, now leading to an alternative. Both users make their changes in their respective alternatives, and do consequently not lock each others work. After all the changes are made to the revision it is frozen to make sure that no more changes are made to that revision

Changes made during such a cycle (creation, changing, freezing) is stored in the server database as deltas. Between two revisions two levels of deltas can occur as a result of changes to:

- node data
- structure

A delta due to changes in the node data is created and presented by the client application (e.g. an editor). Such delta is stored in the server database as a sequence of bytes. Thus, the server do not understand the changes in the node data, only the fact *that* the node data has been changed.

Structural deltas are deltas due to changes in the structure, i.e. a node has been added or deleted. These structural deltas are stored in the server through its internal representation and can be retrieved by the client.

In the following of this paper we will only discuss the structural deltas. For a more detailed description of the deltas created and presented by the integrated editor see [Ols94].

3 Creating Revisions

This section is a discussion of the consequences when new revisions are created. The focus is on the work done in the server and, thus, only structural changes to the document is considered. We will discuss what changes can be made to the structure between two revisions and, when we discuss merge, how changes in two alternatives can be identified and compared. Only one level in the hierarchical structure, i.e. one node and its sons, is presented below, but the presented approach can be applied recursively and will in that way work for the entire document.

3.1 Creation of revisions

In figure 3 two revisions of a node and its sons is depicted. We can see how one son (X) is unchanged, and one son (Y) is added between the two revisions, resulting in the structural delta “Y added”. A son can also be deleted or *changed*. A node is *changed* because of three reasons:

- The node data corresponding to the node (not shown in the figure) has been changed
- A son has been added or deleted
- One of the sons is *changed*

The last item in combination with the hierarchic structure will make a *change* in a node to propagate up through the hierarchy, right up to the root.

One reflection is that when a new revision is created it is always identical with its predecessor, i.e. the delta is empty. Then, during editing, the delta grows.

3.2 Creation of alternatives

If the original revision already has a succeeding revision an alternative is created and parallel development is enabled. In this case, as in the case above, the new revision is equal to its predecessor in the beginning and thus the delta is empty.

Because the original revision now has two succeeding revisions there is two deltas stored in this revision. These deltas are, however, changed independently of each other according to the changes made in the two alternatives respectively. There is no difference in the way the deltas are created and the new revisions frozen compared with a single evolution line.

3.3 Creation of merged revisions

The creation of a merged revision is different from what was mentioned above. Unless exactly the same changes are made in both alternatives the merged revision can not be identical to both predecessors. Consequently, some decisions

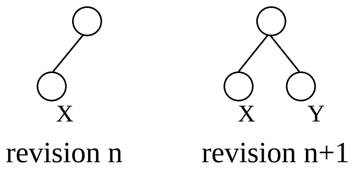


Figure 3 *Structural change between two revisions*

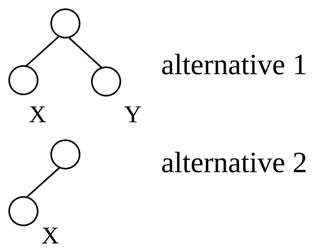


Figure 4 *Two revisions to be merged*

have to be made selecting the changes to include in the merged revision. Furthermore there are two deltas, one for each predecessor, that have to be updated during the merge. All sons (and recursively the entire document) have to be traversed to detect the differences (conflicts) between the alternatives. In addition to the included changes further edit operations can be made and eventually the revision can be frozen.

Conflict?
In traditional merge systems, like Unix's diff, the two revisions to be merged is the only information available when detecting conflicts. Comparing the two alternatives, three different situations can occur: A son exists in both alternatives, A son exists only in alternative 1, or A son exists only in alternative 2. In figure 4, depicting the two revisions to be merged, node X exists in both alternatives, while node Y exists only in alternative 1. There is, however, no possibility to check whether node Y is added in alternative 1 or deleted in alternative 2. Most systems define both these cases as a conflict, and the user has to decide what to do without a more detailed description of the problem.

We benefit from storing the whole history of deltas. By following the deltas from both the alternative revisions back to the last common revision (fork revision) we can identify the sons and follow their evolution. The same example is depicted in figure 5, now with the information from the fork revision included. Y exists in the fork revision and there is thus no problem to identify son Y as deleted in alternative 2.

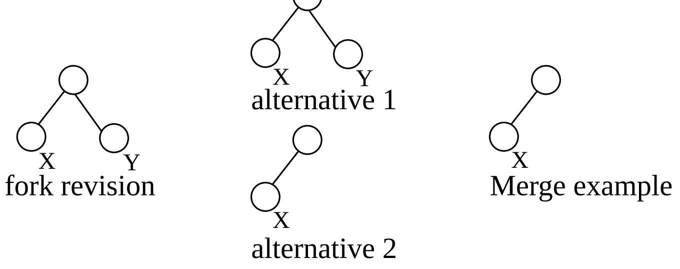


Figure 5 *Merge when using the fork revision*

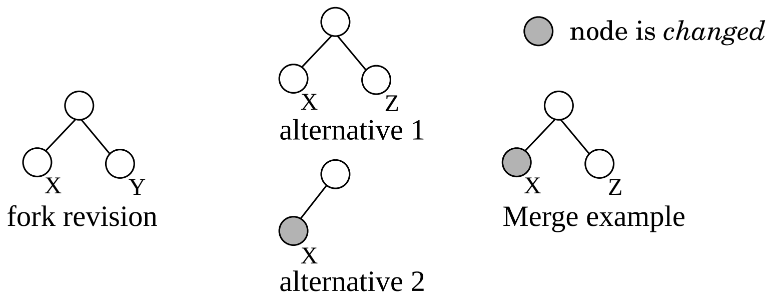


Figure 6 *A more complex merge situation*

A little more complex situation is depicted in figure 6. In this example Y is deleted in both alternatives while a new node Z is added in alternative 1. Son X is unchanged in alternative 1 but changed in alternative 2. Of course, as a result of a large number of changes, there can be several revisions between the fork revision and the revisions that will be merged. However, this will not affect the possible merge situations for a specific son.

Systems like Unix's diff3 and Emacs' (version 19) Emerge also supports a "3-way" merge. These systems are, however, line based (does not support structured documents) and they compare three revisions of full documents. In the example depicted in figure 6 a we can think of the nodes as lines in a textual document. A comparison of the fork revision and alternative 1 would result in the delta "Y changed to Z". Retrieving the deltas and following the operations made to the structure, however, clearly points out that Y is deleted and a new node Z is added.

Default Rules

The examples in figure 5 and 6 show some of the possible merge situations for a specific son. A son can be in one of four states: *not changed*, *changed*, *added*, or *deleted*. Two alternatives and four states leads to 16 permutations. Five of these, however, can never occur. A son added in one alternative can, for example, not be deleted in the other. The remaining 11 cases can, if retrieved by the application and presented to the user, be a better basis for the decisions that have to be made, or they can even be a basis for making some of the decisions automatically. In our system we have defined default rules to suggest a merged result. The user can then modify this temporary result to create the desired merge before freezing the revision. In table 1 all the possible cases for a son is listed and the temporary result from our default rules is shown in the column "Default".

Table 1 *The eleven possible cases for a specific son*

No	*Alt 1 (Main)*	*Alt 2 (Added)*	*Default*
1	Not Changed	Not Changed	Not Changed
2	Not Changed	Changed	Changed
3	Not Changed	Deleted	Deleted
4	Changed	Not Changed	Changed
5	Changed	Changed	Conflict
6	Changed	Deleted	Changed (unsym)
7	Deleted	Not Changed	Deleted
8	Deleted	Changed	Deleted (unsym)
9	Deleted	Deleted	Deleted
10	Added	does not exist	Added
11	does not exist	Added	Added

One predefined default rule in our system is to incorporate all changes, additions and deletions occurring in any of the two alternative development lines. This rule will propose a solution to the cases 1,2,3,4,7,9,10, and 11 in table 1. In these cases the son have been changed in one of the alternatives but is unchanged or does not exist in the other. Going back to the example in figure 5, node X should be included (rule 1), and node Y remains deleted as in alternative 2 (rule 3). In figure 6, the default merge had been to include the change in X (according to rule 2), the added son Z (rule 10), and let Y remain deleted (rule 9).

In figure 7 the cases 5 (X) and 6 (Y) are depicted. To solve case number 6 (we can, of course, not incorporate both a change and a delete of the same son) we will make the proposed merge *unsymmetric*. When the merge is requested also one of the alternatives have to be defined as the *main alternative* and the other the *added alternative*. In table 1 alternative 1 is defined as *main* and alternative 2 as *added*. In this way we can have default rules where we prefer changes from

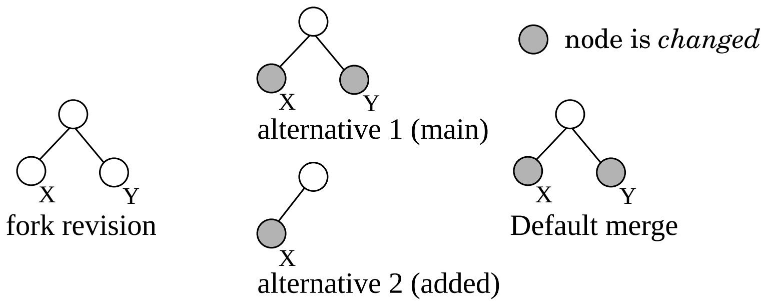


Figure 7 *Adding, deleting, and changing sons*

main rather than changes made in *added*. This is in accordance with the way to work with one main developing line and several alternatives where different approaches or new techniques can be developed and tested. When the testing is finished in the alternative, the new function is merged into the main development line. If not all changes made in the alternative is included in the merged revision a new merge can be made later. The changes not included during the first merge will also this time be presented as changes made in the alternative. We call this mechanism *incremental merge*.

Case number 5 (son changed in both alternatives) is trickier. As mentioned above a *change* can depend on both changes made in the node data of the son, or *changes* in the subtree propagated up through the hierarchy. If the change is due to propagation, the method will work recursively and maybe all changes can be merged by default. If, on the other hand, there are changes made in the node data, the client application, e.g. the integrated editor, has to solve the conflict on the node data level and give the server the two resulting deltas.

4 The Revision Tree Model

Section 3 described the different situations that can occur during a merge and how these situations can be a decision base for a default merge. In this section the *Revision Tree* server and its internal representation in the server database is described. We will focus on some of the cases occurring during a merge and how the representation allows us to identify them.

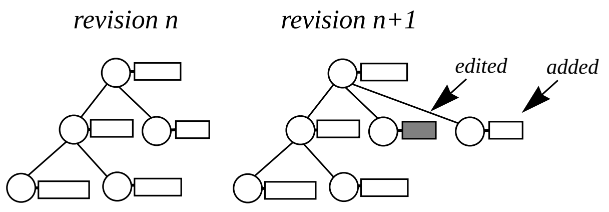
First, in section 4.2 the client-server communication during a merge is described and how the merge is done lazy. After that, in section 4.2, a short general description of the server database representation is made. This is done very briefly and a more detailed description can be found in [MAM93]. In section 4.3 an example is used to show that this representation can detect the eleven merge cases.

4.1 Client-Server communication during merge

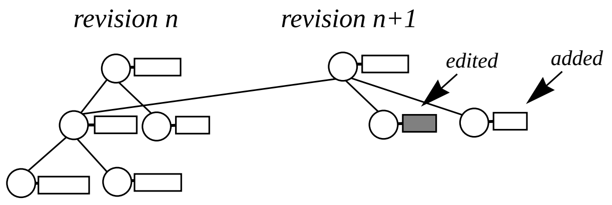
The merge is done hierarchically and lazy, and is controlled by the client application. First the application orders a merge and tells the server the revision number of the two revisions that will be merged. The server creates a new revision, finds the last common revision for the two revisions and compare the root node sons. According to the eleven cases the server then makes the preliminary default merge of these sons. It then returns a list to the client with all the sons and in to what merge case they belong. If, for example, a son is reported as *changed* in both alternatives (case 5) the client does not know of what reason the node is marked *changed*. It is now up to the client to order merge of such a son, making the server to do the same procedure for this son's sons. In this way the merge is done lazy and the application can browse down to the real conflicts.

4.2 The server database representation

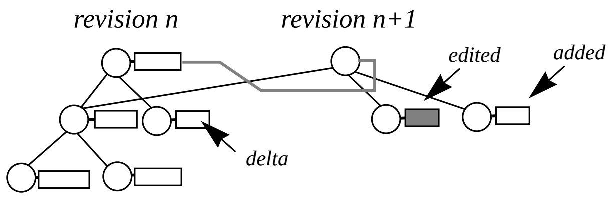
Figure 8 depicts the database representation of two succeeding revisions. Conceptually a new copy of the entire document is created when a new revision is created. As depicted in figure 8a changes, one node data edited and one added node, only affect the new revision. To avoid replication of data, nodes not changed are shared with the previous revision. In figure 8b an unchanged subtree is shared between the two revisions. The figure also depicts how a change propagates up the tree (the father node to changed node is never shared), and in particular how the root node (i.e. the document) is considered changed if there is any modification in the tree. Due to the change propagation some nodes are duplicated despite their corresponding node data is unchanged. To avoid replication of such node data, also the unchanged node data is shared. This is shown by the dashed (shadowed) line in figure 8c. The figure also shows that the node data in the edited node of revision n is replaced by the delta produced by the client application during changes made in revision n+1. With this delta and the edited node data in revision n+1 the client application can re-create the replaced node data.



(a) Conceptual view



(b) Sharing node revisions



(c) Sharing node data

Figure 8 *Sharing of nodes between revisions*

4.3 Representation

We will now, by an example, show that the representation can detect the different cases occurring during merge. Figure 9 depicts the representation after a merge. To make the example as simple as possible the shortest possible evolution, with only four revisions including the merge, is depicted. Revision r1 is the last common revision, r2 and r3 the alternatives respectively, and r4 the merged revision.

Starting at the top level, examining the sons one by one, we can see that son 1 (folder A) is changed in both r2 and r3. If a son is not shared with the last common or an older revision and the son exist (may be shared) in the last common revision, it is changed. A son that is changed in both alternatives, case 5, must be duplicated in the merged revision to enable that the merge can continue recursively down in that son. The next son (folder B) is shared in both alternatives and obviously not changed. The default rule is here to include (share) such a son (according to case 1 in table 1).

This was level one in the lazy merge. Next level starts when the client application wants to see for what reason “folder A” is marked as a conflict. Exactly the same algorithm is used, now with “folder A” as the father and its sons under examination. In our example son 1 (Doc One) is changed in both alternatives and marked as conflicting, and son 2 (folder C) is shared in both and consequently unchanged. Son 3 (Doc Two) is not shared in r2 and it does not exist in r3. Depending on the existence of Doc Two in the last common revision the son is added in r2 or deleted in r3. In our example there is no “Doc Two” in r1 and “Doc Two” is consequently marked as added in r2.

If the client now wants to go one further level and selects “Doc One” the server detects it as a conflict on the node data level. The node data is retrieved from both alternatives including all the deltas back to the last common revision. With this information the client application can make the merge on the node data level and produce the two required deltas, which will be stored in the old nodes respectively (marked *). The full merged node data is stored in r4 (marked **).

The dashed lines down from “folder B” and “folder C” depicts that there can be subtrees under these nodes. Such subtrees will not influence the presented approach.

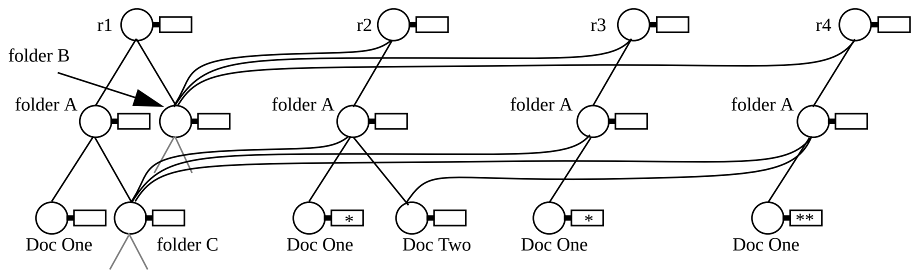


Figure 9 *Revision Tree representation*

5 Summary And Future Research

Revisions are fundamental for systems supporting collaborative work. Thus, integrating revisions into a system, making both the client application and the server database knowing about revisions, enables a advanced support. In particular merging two alternative developing lines of a hierarchical document involves many decisions based on the work made in the two alternatives. It is thus valuable to have a system that in the first place helps identify the cases of conflicts that can occur, and second give support in solving such conflicts. A server that knows the evolution graph can find the common ancestor of the two alternative revisions, all the needed deltas, and can by following predefined default rules propose a preliminary merge. The client application can then interactively agree to or modify this automatic preliminary merge.

The next step in development of the server is to support multiple servers in order to improve speed and stability for geographically distributed teams. This involves developing automatic merging of structures (after network or server failure) and protocols to keep the duplicated database consistent at all sites. Here questions about access policies and protection will also be urgent both for access to the data and revisions. We have so far ignored these questions and focused on enabling sharing and collaboration in the first place.

Although we have only mentioned supporting merge of two alternatives at the time, there is no inherent problem in merging several alternatives at the same time using the same default rules. In fact, it would be an interesting facility to hypothetically merge all existing alternatives of a system to get an overview of how the system develops, including future merge conflicts to prepare for.

In the long run we aim at using this technique also for storing structured program information in Abstract Syntax Tree (AST) form. This will involve, among other things, development of algorithms for representing, and applying deltas to trees. Some of the needed algorithms are documented in [Gus90]. In the Mjølner Orm environment we also need to store ASTs decorated with semantic information. This includes designing representations of deltas of semantic information which will avoid long re-calculations to take place when revision focus is changed. We will also investigate the use of semantic information for making more intelligent diff and merge functionality, not only taking lexical changes into account but also the semantic consequences of a change [HPR89].

Acknowledgment
I want to thank all the members of the software development research group at Dept. of Computer Science, Lund University, for stimulating discussions which have contributed substantially to the work presented in this paper. In particular, I want to thank Sten Minör for constructive comments on earlier drafts of this paper.

The work presented in this paper was supported in part by NUTEK, the Swedish National Board for Industrial and Technical Development.

References

[ARMEA89] Lie Annund, Conradi Reidar, Didriksen Tor M., and Karlsson Even-André. Change Oriented Versioning in a Software Engineering Database. In *Proceedings of the2nd International Workshop on Software Configuration Management*, volume 17, pages 56–65. ACM SIGSOFT Software Engineering Notes, October 1989.

[EGR91] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware. Some issues and experiences. *Communication of the ACM*, 34(1):38–58, January 1991.

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

[HPR89] S. Horwitz, J. Prins, and T. Reps. Integrating Noninterfering Versions of Programs. *ACM Transactions of Programming Languages and Systems*, 11(3), July 1989.

[Kan93] J. Kannegaard. The nineties are an exciting time for software development. Keynote address at TOOLS Europe'93, Versailles, March 1993.

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

[LvO92] Ernst Lippe and Norbert van Oosterom. Operation-based Merging. In H. Weber, editor, *SIGSOFT'92 Proceedings*, Tyson's Corner, Va., December 1992. ACM. SIGSOFT Software Engineering Notes, 17(5).

[MAM93] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-Grained Revision Control for Collaborative Software Development. In *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*. Kluwer Academic Publishers, 1993.

[MMH$^+$90] Boris Magnusson, Sten Minör, Görel Hedin, et al. 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.

[Ols94] Torsten Olsson. Group Awareness Using Fine-Grained Revision Control. In *Proceedings of the Nordic Workshop on Programming Environment Research*, Lund, 1-3 Juni 1994.

[Roe75] M. J. Roekind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, December 1975.

[Tic85] Walter F. Tichy. RCS - a system for revision control. *Software Practice and Experience*, 15(7):634–637, July 1985.