Fine-Grained Revision Control for
Collaborative Software Development

Boris Magnusson
Ulf Asklund
Sten Minör

LU-CS-TR:93-112

Also published in: Proceedings of ACM SIGSOFT'93 - Symposium on the Foundations of Software Engineering, Los Angeles, California 7-10 December 1993

Department of Computer Science
Lund Institute of Technology
Lund University

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

Fine-Grained Revision Control for Collaborative Software Development

Boris Magnusson
Ulf Asklund
Sten Minör

Department of Computer Science, Lund University
P.O.Box 118, S-221 00 Lund, Sweden
Email: {Boris.Magnusson,Sten.Minor,Ulf.Asklund}@dna.lth.se

ABSTRACT

This paper presents a framework for controlling the evolution of complex software systems concurrently developed by teams of software engineers. A general technique for fine-grained revision control of hierarchically structured information, such as programs and documents, is described and evaluated. All levels in the hierarchy are revision controlled, leaves as well as branch nodes. The technique supports sharing of unchanged nodes among revisions, automatic change propagation, and change-oriented representation of differences. Its use in a software development environment is presented, facilitating optimistic check-out of revisions and alternatives, check-in with incremental merge support, visualization of change propagation, and an integrated flexible diff-ing technique providing group awareness for team members.

KEYWORDS

Software development, version and configuration control, incremental merge, teamware, CSCW, group awareness

1 INTRODUCTION

Despite the fact that software systems are developed, documented and maintained by teams, most development environments give poor support for people working together. The availability of world-wide networks adds a dimension of geographical distribution to the picture and makes the problem even more urgent. Networks are not only used for distribution of notes and news, but are often used as an essential component in the infrastructure. It is for example very common for co-authors of papers to depend on a working e-mail connection. When people work together in existing environments, different patterns of coordination develop. One is *turn-taking*, where one person at the time does his changes. Another is *split-combine* where the shared document is partitioned and each person does changes in his part. When all are done the updated pieces are combined again. Yet another method is *copy-merge* where each person is given a full copy of the document, does his changes, and in the end all changes are merged together. Although these cooperation methods might serve in restricted situations where few persons are involved, during a short time period and developing single documents, there are some severe drawbacks with each strategy:

- Turn-taking means that only one person can work at the same time.
- Split-Combine means that the partitioning has to be fixed over some period of time.
- Copy-Merge gives a merging situation that can develop into a nightmare in case there is no strategy for who changes what and no support for merging.

For these reasons none of the techniques can be directly carried over to developing software in large teams, with a large number of inter-dependent software documents. The copy-merge approach does, however, have the advantage of providing maximal flexibility, allowing all authors to change what they want, whenever they want. To make copy-merge viable the support for merging has to be improved.

1.1 Interaction Modes and Requirements

The work in development and maintenance of software systems typically alternates between group tasks involving many persons and individual assignments. Group tasks involves activities such as initial design and specification, work structuring, integration of components, and also testing and debugging during integration of components where several groups and individuals are involved. Individual assignments involve detailed design, implementation of specific components, documentation, and component testing and debugging. Much of the implementation work is thus performed asynchronously by individuals or small groups, more or less independently. In this situation it is rarely interesting to see what somebody else is doing right now, but what has been changed over a period in time, say during the vacation, since the last release, etc. Now and then there is need for coordination and planning of the work. Simultaneous interaction with others in the team is then needed. The environment thus needs to support development in *both synchronous and asynchronous modes* of interaction.

In performing a task, like a bug fix or some new implementation work, it is often the case that several different parts of the system need to be altered. It can be difficult from the outset to determine exactly which parts to alter. The environment must thus allow the developer to access and change *any* part of the system within a reasonable time frame. It is, furthermore, very hard to determine how long it will take to complete a task. Locking files for a particular task is thus not acceptable. A policy allowing several

users to check-out a copy of the same version is needed, accepting that merging will be required at check-in. There might be other reasons preventing a developer from changing a specific component, but that should be access policies, not a result of the architecture of the environment.

Software components are often organized according to their logical relations into modules, abstract data types, or classes. One consequence of such an organization is that a change of a small part of such a component results in the whole component being considered as changed. This is of particular importance in object-oriented languages, since the update patterns differ from modular procedural languages. In an object-oriented language, adding functionality typically means adding methods in several classes, all now considered changed. In a procedural language the same scenario leads to only one new procedure in a new file. The environment must thus provide mechanisms to *limit the effect of a change*. Furthermore, a class can be seen as a configuration of operations, each having its own revision history, which brings configuration aspects into the picture already at this stage.

Since a task often involves changes to several different software components, the environment must support the creation of *alternative configurations*, where new revisions of software systems can be built and the modifications tested. Changes to different modules originating from the same task depend on each other. The environment must give support in keeping track of dependent changes.

A result of an optimistic check-out policy, as requested above, is the need for frequent merging of alternatives. The environment must thus give strong *support for performing merging and detecting conflicts*. The environment should also provide mechanisms to enable developers to avoid causing merge situations by mistake.

1.2 Existing Environments

As examples of the existing revision control systems and the functionality they provide we take the popular systems used in Unix environments, RCS [Tic85] and SCCS [Roe75]. These systems work on complete files which often is a much larger piece than is affected by a single change, and often is the target for many independent changes. They also use a pessimistic, lock on check-out, protocol. This can be problematic when used by a local team, even creating dead-lock situations, but the problem can often be sorted out by small meetings.

In a large distributed team, such spontaneous meetings are not practical, and the problems grow with the number of developers involved. Furthermore, these systems are using state-based differences between revisions, calculated from the edited files. Change-oriented differences, e.g. recording operation-based differences while editing, seems to have a much larger potential for supporting merging of variants as described by Lippe and Oosterom in [LO92] and Lie *et al*. in [LCD+89]. The conventional revision control systems do not give much support for teams at the level of ambition we are heading for. Some systems provide more advanced support from other points of view (DSEE [LCM85], NSE [NSE]), but share the common problematic characteristics outlined here.

For teams at a single site one can rely on network file systems like NFS [San85] and Andrew [MSC+86] for the file access. These systems do not, however, solve the team related problems we are discussing since revision control aspects are not addressed. For distributed teams the files are often manually distributed (e.g. copied) by means of systems like email or ftp. The result is that revision control has to be done at all sites more or less manually. Automated support for distribution of the files, maintaining the revision control information, and merging is needed. The recent product Teamware [Teamware] from Sun Microsystems Inc. is an attempt to attack some of these problems. It does use an optimistic check-out policy with merge support, but uses a line-based textual representation and a state-based differences approach. This development is encouraging since it supports our fundamental decision of regarding the copy-merge strategy as viable.

The work on groupware and synchronous editors [EGR91, KP90, MO92] are other interesting efforts to support the group related activities discussed above, but does not solve the problems during asynchronously performed tasks. These editors also do not provide any mechanisms for revision and variant control since the metaphor is that there is only one shared document.

1.3 Our Approach

Our conclusion is that an ambitious team-supporting system must fulfill all of the requirements mentioned in section 1.1. The demand for flexibility and avoidance of inter-locking between different developers means that the systems must use an optimistic check-out approach and no locking. As a consequence, it must reduce the inconveniences that result from parallel development.

By using a fine-grained revision control model the effect of a change and thus parts to merge are kept smaller. By using a change-oriented representation the merge support can be more sophisticated since it can explore the information about the nature of the changes. By using a hierarchical representation with integrated revision control mechanisms it directly supports keeping track of related changes in the same storage unit. As an example of where merging might be trivial using these techniques, consider merging of two variants, one with a change to a local procedure of a class, and one variant with an addition of a new procedure.

A shared revision history graph provides overview and will make team members aware of each others' changes at a coarse level. The active diff facility enables team members to also see each others' changes at the fine grained level as they are made. This facility comes very close to the synchronous editing mode facilitated in group editors.

In summary we have adopted the copy-merge metaphor and provide mechanisms to reduce the merge nightmare. By exploring hierarchical organization we get the benefits of the split-combine metaphor without its drawbacks.

The system presented here 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 [MMH90], supported collaborative software development to a limited extent through its configuration management [Gus90]. The aim of the current project is to support teams of programmers working on the same system.

In section 2 we give an overview of the system architecture we are working with and in section 3 we present the data structure designed to support fine-grained revision control of hierarchical documents. This is the main contribution of this paper.

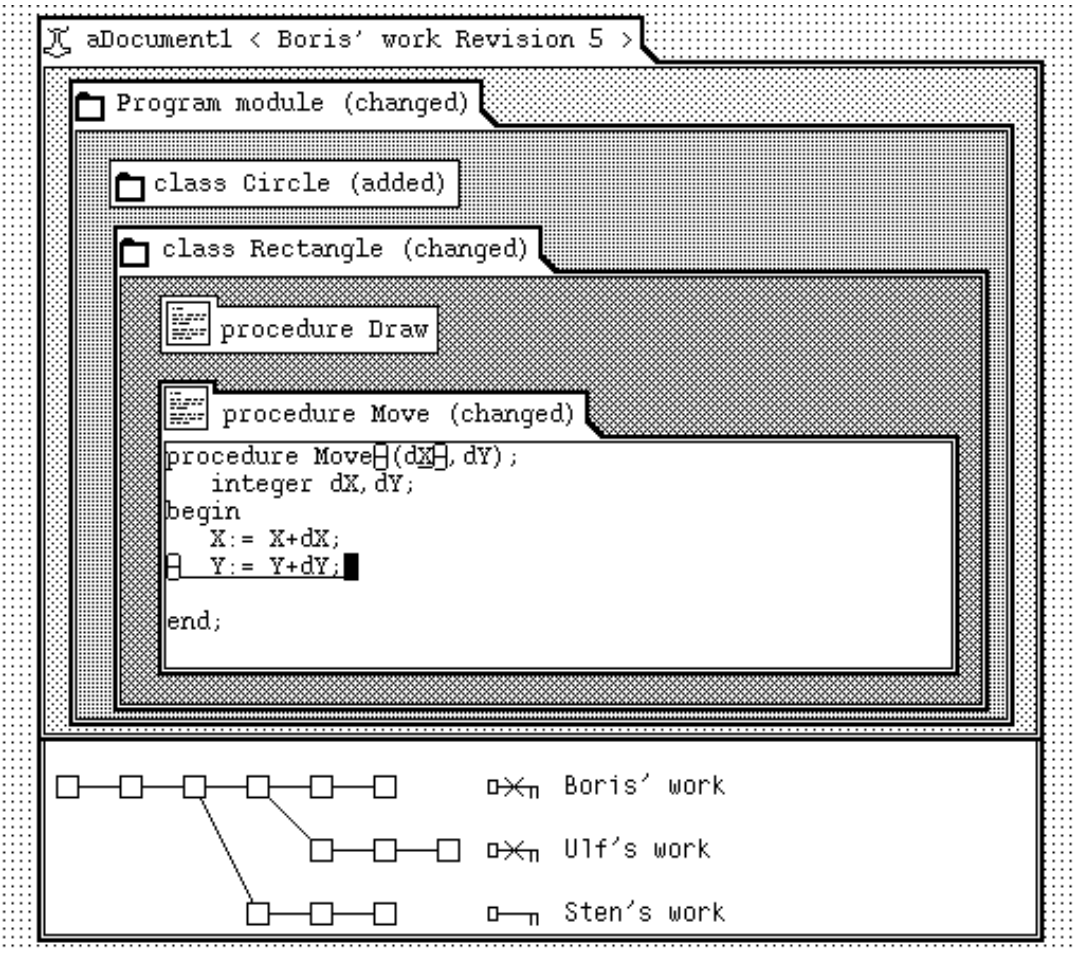


Figure 1. Hierarchical program representation

2 AN OVERVIEW OF THE SYSTEM

The system provides a flexible model for editing which covers both conventional asynchronous editing and synchronous editing. A synchronous editor allows multiple users to access and edit shared material simultaneously employing the WYSIWIS (What You See Is What I See) metaphor. This has been an active research area within the CSCW (Computer Supported Cooperative Work) community during the last years [EGR91]. As discussed in the previous section, this model is inappropriate for program development since different activities require different strategies, both synchronous and asynchronous, and a smooth transition between them [MM93].

First in this section we show the hierarchical representation of programs and the hierarchical browsing. We then give a short introduction to the fine-grained revision control functionality, which automatically keeps track of program modifications and supports merging and simultaneous editing by different users. Finally, we present the notion of active diffs, which are used for making users aware of modifications done by other users.

2.1 Hierarchical Representation and Browsing

A program[1] is organized as a hierarchical structure of fragments. The fragments typically correspond to abstractions in a programming language, such as blocks, classes, procedures, and functions. This hierarchy is fundamental in our architecture. A program is displayed using this hierarchy, it is edited in terms of the hierarchy, all elements in the hierarchy are revision controlled, and the database server stores programs in terms of the hierarchy.

Figure 1 shows an example of the user's view of a program. In the figure the program hierarchy is at the top and an evolution graph below. The program contains several class and procedure fragments, which are shown and manipulated by a hierarchical browser allowing parts at one level to overlap but not to be moved outside its parent window. Subparts are added and removed in the browser by selecting entries from a menu, e.g. "new class". The parts in the figure contain text which is edited by a text editor.

The subdivision into particular parts is not built into the system, but is described in a grammar specifying the hierarchy. By supplying different grammars, the browser may be tailored to different programming languages. It may also be used for other application areas, such as authoring where a hierarchical document consists of chapters, sections, and paragraphs. The

1. We use the term "program" here in a generic fashion. It may be a program fragment, typically a module.

approach for hierarchical browsing was initially developed in the Mjølner Orm environment [HM88] where it turned out to be a useful and "intuitive" way of viewing and editing program structures. Furthermore, in a collaborative environment it may be used as a means to organize the work.

2.2 Fine-Grained Revision Control

All fragments of the hierarchy are revision controlled, which will be explained in further detail in section 3 of this paper. Here we will only give a brief overview. The lower part of figure 1 shows an evolution graph of the program. An evolution graph contains results from three different actions: creating revisions, creating alternatives, and merging alternatives as depicted in figure 2.

A revision is one step in the evolution of a document. It may contain arbitrary changes compared with its predecessor. Typically it may contain new classes and procedures, parts may have been removed, or it may contain minor modifications such as error corrections. A revision is never changed once it has been frozen. The only way of modifying a document is to create a new revision or alternative with the desired modifications.

Alternatives serve two main purposes. One is to split up the evolution into two (similar) programs with different purposes. A program may, for instance, be modified into two alternatives for different target machines but with a core of common contents. The other purpose is to support simultaneous development among multiple users. If one user wants to edit a revision that is edited by another user, an alternative is created instead of a revision. The users edit their own alternatives and merge them into one revision when ready.

This hierarchical fine-grained revision control functionality, where all fragments (in figure 1, for instance) are revision controlled, is implemented as basic functionality of the database server for hierarchical documents. It is also supported by the other parts of the system. One reason is to support group awareness, which we will expand on in the next section.

2.3 Group Awareness

In addition to the advantages revision control offers to teams of programmers, our system supports collaboration through group awareness. It is based on the hierarchical organization and the revision control functionality. Group awareness is available in two ways in the system: by the evolution graph which is shared among all users and by presentation of *active diffs*.

When a user enters the system, the evolution graph indicates the status of the program. The user can see who is editing it at the moment and what has happened since last time. In figure 1, for instance, one can see that there are three alternatives of the document and that two alternatives are edited at the moment since the editing keys of these alternatives are crossed over. By checking out a revision not in use for editing, that revision may be edited asynchronously. By checking out a revision in use by some other user, an alternative is created, which allows simultaneous editing of the same revision and later merging. In order to make users aware of what other users actually are doing in the document, e.g. in two alternatives simultaneously edited by two users, the system provides active diffs.

An active diff shows the difference between revisions or alternatives of the document. The diffs are based on the actual edit operations performed, which are stored by the revision/database handler as deltas between revisions. In this way fine-grained and accurate differences can be presented which reflect what modifications actually have been performed by the user. An example is given in figure 1. The differences between the current and the previous revisions are shown in the text window used for editing. All insertions appear as underlined text and all deletions as "-" markers. The "-" markers may be expanded interactively in order to see the actual deletions. In the same way the underlined insertions may be collapsed to "+" markers. The presentation is quite similar to the diffs used in Prep [NCK+92], but in our systems these markers change interactively as result of the ongoing editing. Differences at the structure level are shown in a similar way with the icons in the window titles marked as additions, deletions, and changes within a part (or one of its subparts).

Active diffs between arbitrary revisions and alternatives can be shown. This is done by selecting revisions for diff presentation in the evolution graph. In this way it is possible to see what changes were made after a certain revision was created or in what way alternatives differ. Active diffs can also be shown for different alternatives currently being edited. If, for instance, a user A sets up a diff between the alternative he is currently editing and an alternative simultaneously edited by B, the differences between the alternatives will be shown continuously, i.e. all modifications performed by B will appear as markers in A's window (hence the name "active diffs"). Notice that these diffs are shown on A's demand and A is free to turn off the diff presentation whenever he desires. The diff markers cannot be edited by A, A can only edit his own alternative. The example with two users can be generalized to several users, e.g. using color coding of the markers to visualize who has changed what. Active alternative diffs may in this way be used for giving a "synchronous editing" view of the document, presenting a "what-if-I-merged" situation. The resolving of conflicts is, however, delayed until merging at check-in. The awareness provided by the active diffs during editing contribute to avoiding unintentional conflicting changes and thus unnecessary resolving of merge conflicts.

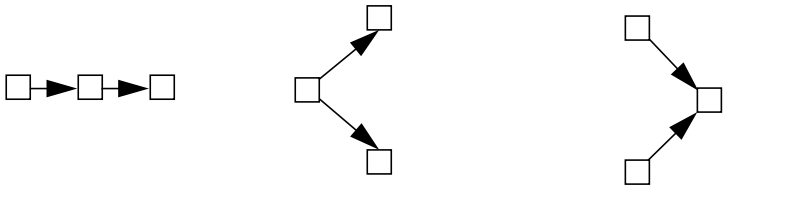


Figure 2. Different kinds of revisions

2.4 System Architecture

The system is organized as a client-server architecture. The database server stores the documents and synchronizes the access from the client applications run by the users. A major difference from other client-server systems is that the functionality in the client-application must understand the revision control mechanisms. For example an editor must be aware of which revision of the document it is editing and must supply change oriented diffs as the document is edited. It must also contain the functionality to re-construct an old revision from these diffs.

3 FINE-GRAINED HIERARCHICAL REVISION CONTROL

In this section we will describe the *RevisionTree* server, an engineering database for storing hierarchical information. First, in section 3.1 and 3.2, we will consider the representation of the structure of a RevisionTree to represent revisions and alternatives. In section 3.3 we will discuss revision control of data in the structure as well, and support for change-oriented editing. A two-level merging approach of the structure and its data is presented in section 3.4 followed by an evaluation of the techniques in section 3.5.

3.1 The RevisionTree Model

The data model in RevisionTree is a simple hierarchical structure. The nodes of the hierarchy corresponds to units which may be chosen by the application. Typically, in a software development environment as described in the previous section, the nodes will correspond to abstractions in the programming language, such as classes, procedures, and functions. In an application for authoring, nodes may correspond to chapters, sections, and paragraphs. The RevisionTree server is only responsible for storing, retrieving and revision controlling the node data as a sequence of bytes without making any semantic interpretation of it. Figure 3 gives an example of the representation of the program module shown in figure 1.

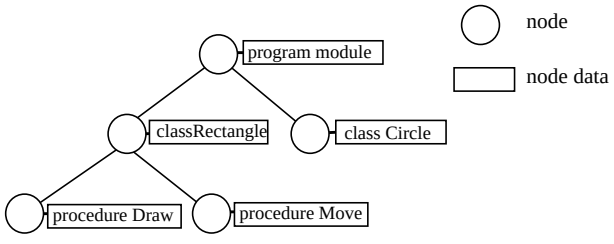


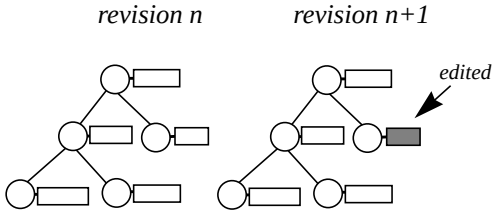
Figure 3. A hierarchically structured program

Each node in the tree is subject to revision control. If, for example, the data of a node is changed, a new revision of that node is established. Furthermore new revisions of its ancestor nodes up to the top level are also created by a change propagation mechanism. Conceptually this may be viewed as establishing a new revision of the entire tree. In practice this seems to be reasonable since a new revision of a part, say a class in a program, actually results in a new revision of the program as a whole.

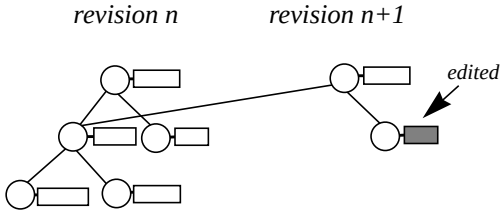
Somewhat more formally we define:

> A node is *changed* if its node data is modified or any of its son nodes are *changed*, added, or deleted.

A change is thus propagated up the tree and in particular the root node is always considered *changed* if there is any modification in the tree. In order to avoid replication of data, nodes that are not *changed* are shared. Figure 4 shows an example.



(a) Conceptual view



(b) Sharing node revisions

Figure 4. Sharing of node revisions

In (a) the program hierarchy is shown in two revisions. In the second revision the node data has been altered, conceptually resulting in a new revision of the program including the new contents. In (b) a new revision of the changed node and the node data is created. A new revision of the root node is also created as a consequence of the change propagation mechanism. The left part of the tree, unaffected by the change, is, however, shared between the two revisions. The overhead for the change propagation is hence reasonably low. In addition, the modified node data is represented as backward deltas, upon which we will comment further later in the paper. One reason for sharing revisions in the tree is, of course, space efficiency, but an equally important property is to support change-oriented editing and merging, which is facilitated by the ability to identify what actually has been modified in the tree. We will expand on this in section 3.3 and 3.4.

Naturally the application must see the same tree structure, independent of whether the storage is using sharing or not. In the simple case considered in figure 4 it is obvious that the representation is correct in this sense.

3.2 Growing a RevisionTree

We will here show the evolution of a hierarchical revision tree in an example, starting from the structure in figure 3. First a single evolution line of revisions is presented, followed by an example showing alternative revisions.

Growing Revisions

Creating a new revision is initiated by an operation to open the new revision (based on an existing revision), followed by operations that change the structure of the document or the content of nodes, and finalized by an operation that freezes the revision. The change operations are the result of user actions. We do not here discuss exactly how they are triggered; after each edit operation, now and then by an “auto save” function, or explicitly on user demand.

Figure 5 shows an example of growing revisions. To maintain a tree structure when new revisions are created, we add a *main root* with the revision roots as sons. This node is used to store the evolution graph (depicted in the rectangle above the revision tree), similar to the one in the user interface of figure 1. In figure 5a we see the result when the creation of a new revision just has been initiated. In this situation different edit actions can take place. The user might edit the content of an existing node, node 5, as shown in figure 5b. The system may have to build nodes also for ancestor nodes, due to change propagation, if these nodes do not already exist in the new revision. This is illustrated by node 2' in our example. In figure 5c we show the creation of yet another revision (building on the one we just created) and here we have added a new node, node 6. Deletion of a node is as simple. We also like to point out that there are often several changes from one revision to the other although we have chosen here to show cases where there is only one change in each step in order not to make the drawings too complex.

Again we think it is fairly easy to conclude that the sharing mechanism is correct in the sense above also in these cases.

Growing Alternatives

In the description above only a single evolution line has been discussed. There is, however, often a need for parallel development and thus several alternatives. Figure 6 shows how an alternative is created (revision r4). Node 5 is edited to 5*. Revision r1 now has two succeeding revisions (r2 and r4). It is, of course, possible to create several alternatives from one fork node.

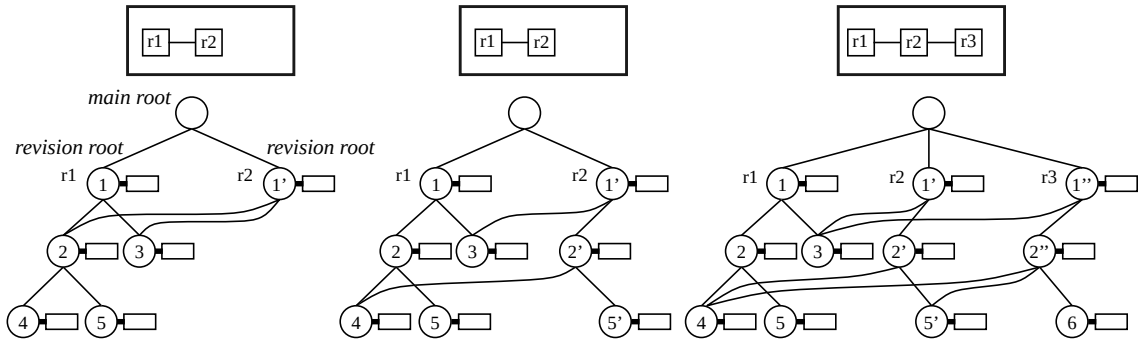
3.3 Change-Oriented Editing Support

In the discussion in the previous sections we have ignored the handling of data in the nodes, or assumed the data to be copied when node data is edited or a node is copied as a result of the change propagation mechanism. We will here describe how this is improved by using a change-oriented delta scheme and in some cases sharing of the node data, although the nodes themselves are not shared.

From the RevisionTree server's point of view, the application will provide a new revision of the node data together with a delta with each write request. It is the responsibility of the RevisionTree server to store this information and return node data and deltas in future read requests. It is thus the application that has the capability to construct and combine deltas. The RevisionTree server uses a backwards delta technique, storing the latest revision in full, since we expect these to be most likely to be retrieved frequently.

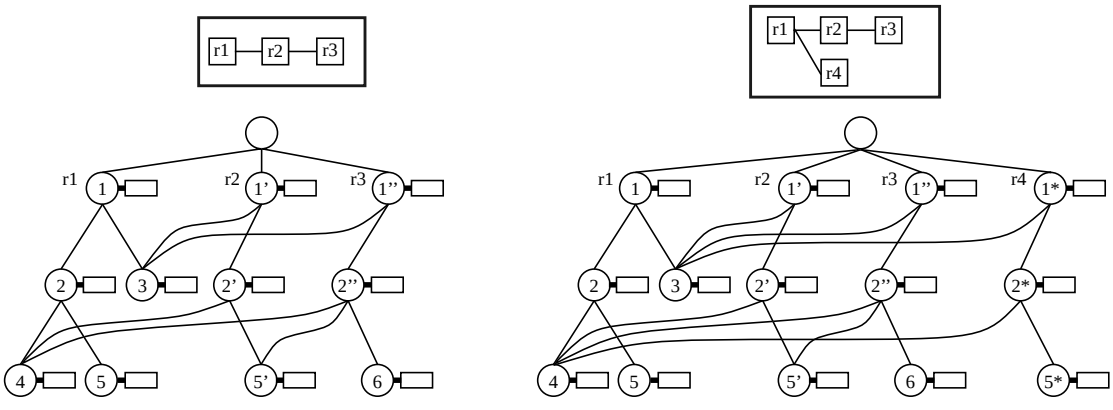
Figure 7 shows a simple example where the data of a leaf node has been edited. This is actually a more complete version of figure 4b, which was somewhat simplified for the sake of clarity in the presentation. Figure 5 and 6 should also be augmented with shared data and backwards deltas in order to give the full picture.

Figure 7 differs from figure 4b in two respects. First, the two revisions of the root node share the same data. The data has not been modified since the second revision of the root node is the result of the change propagation. In this way there is no overhead



(a) Revision 2 initiated (b) Editing node 5 in revision 2 (c) Adding node 6 in revision 3

Figure 5. Growing revisions



(a) Single evolution line (b) Alternative revision created

Figure 6. Alternatives

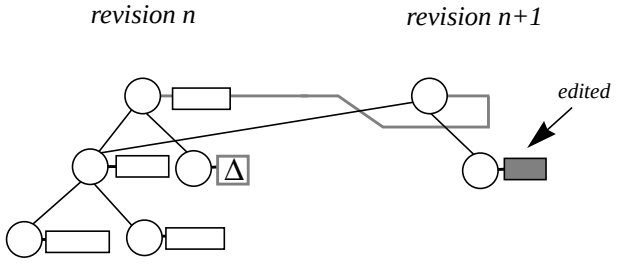


Figure 7. Deltas and sharing of data in node revisions

for replicating the data for the two revision nodes. Second, the edited data of the leaf node is not replicated. The full data is stored in the latest revision and the data of earlier revisions are represented as backwards deltas. The deltas are basically the inverses of the editing operations performed by the user in editing the data from revision n to revision n+1. In the case of alternative revisions, the full data is stored in the latest revision of each alternative. The fork revision where the alternatives spawn thus has several deltas, one for each alternative. Furthermore, on the creation of an alternative revision from a node which is not the latest, the full data has to be re-created from the latest revision by applying the deltas from the latest revision to the fork node.

The deltas serve several purposes. One is scalability. The deltas save space when the number of revisions grow, still allowing the data of any revision to be reconstructed. Another reason is to provide more advanced change-oriented merge support and presentation of active diffs, which both are based on the actual operations performed by the user rather than a computation of the differences.

3.4 Two-Level Merging

In most cases we expect people to organize their work as little projects which will be merged into a main development line when "ready". Potential conflicts here might be caused by other projects already merged into the main line. The experience from people working with this kind of optimistic check-out technique is that conflicts are rare. In 90% of the cases the same file has not even been touched by more than one person [Kan93]. It is thus valuable to have a system that in the first place helps identify these rare cases of conflicts, and second gives support in solving such conflicts.

Merge of alternatives thus takes place at two levels, the node structure and the node data level. On each level the system uses default rules to suggest a merged result. The user can then modify this temporary result to create the desired merged revision. In figure 8 we illustrate this technique on a small example, merging alternatives r3 and r4 in our running example to a new revision r5. The two alternatives have a common root (revision r1). In one alternative (revision r2 and r3) we have edited node 5 and added node 6 as a son of node 2. In alternative r4 we have edited node 5.

On the *node structure level* the default rule is to incorporate all changes, additions and deletions occurring in any of the two alterative development lines.

In the example this means:

- Node 3&4 are included since they are shared among revision r1, r3 and r4.
- Node 6 is included and marked as added since it was added in revision r3.
- Node 5 is included and identified as a conflicting node since it was edited in both development lines.
- Node 2 is marked as conflicting since it contains a conflicting node.

If the user is not happy with this result he can for example delete node 6 from the merged result. In case of conflicting modifications of the node structure, e.g. a node inserted in one alternative in a subtree deleted in another alternative, this has to be resolved manually by the user.

On the *node data level* the conflicting nodes are resolved by the application. Again, the application provides default rules in order to suggest an alternative. The text editor we are working with is optimistic and will suggest that all textual changes in a conflicting node should be included. Changes are presented in a similar way as shown in figure 1. Also here the user can remove some of the changes and do any modifications he wants.

After a merge the developer might want to continue to work on an alternative, creating new revisions, although some of the changes he has made have already been merged. At a future point a merge will include all recent changes and these that were ignored at the previous merge. The mechanism thus supports *incremental merging*.

3.5 Evaluation

The hierarchical structure of revisions in a RevisionTree may be considered as a kind of configuration management. However, the aim of the model is not primarily to support the representation of

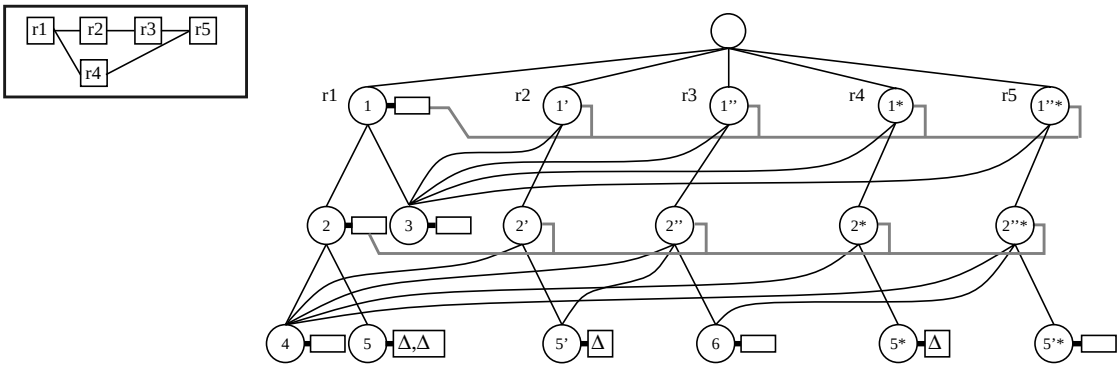


Figure 8. Revision 3 and 4 merged to revision 5

generic configurations and dynamic version selection, as described in [Tic88], but rather to support the evolution process of a configuration. One common feature of our model and the change oriented model by Lie et al. [LCD+89] is that the revisions are global. This means that a revision of the document first will be selected, and then the individual nodes in that single revision can be accessed.

Our model is similar to the unified framework for version modeling in engineering databases proposed by Katz [Kat90] in its revision controlled component hierarchies and the change propagation mechanism. However, whereas Katz model is concerned with limiting the scope of the change propagation, our model always propagates changes all the way to the root. In order to do this efficiently, the sharing of node revisions as presented above and the delta techniques are crucial. A system with integrated revision control functionality using sharing and change propagation techniques is the EH editor [FM87]. The aims of EH differs from ours. EH uses AVL-DAGs for storing documents as a list of lines, each vertex containing a line, whereas we are modeling the document structure where each node contains a class, procedure, etc. Furthermore, our model is designed for supporting merging of configurations, comprising both the configuration tree and node data, which is not addressed in the above models. Yet another fundamental difference from many other systems is that our approach handles revisions as a basic property of the database server and furthermore that the applications (e.g. the editor and browser as presented in section 2) support revision control. This integrated revision control is in contrast to most other systems which “glue” revision control on top of existing files and tools.

Two major performance and scalability concerns may come up regarding our representation: storage overhead and time to produce an old revision of the information in a node. The storage overhead comes from storing deltas from old revisions and from the overhead for structure information of the tree. The structure information is fairly small, one pointer per node, which should be negligible if the content of the node has a reasonable size, like a procedure or a text paragraph, which is the intended use. Also keeping track of deltas costs one pointer to be compared with a separate file in the traditional approach costing file directory overhead and much longer access times. The deltas themselves could (in the worst case) be stored in the same format as standard diffs (as generated by the Unix utility *diff*, used by other revision control systems), so there is no reason to believe that our storage form is inherently worse than current techniques. There are in fact several aspects that indicate that the revision trees might be more memory efficient. First, the change-oriented deltas can turn out to be more compact than traditional line-based text-oriented diffs, especially if changes typically are small compared to the length of the source line. Second, RevisionTrees use a large amount of sharing of common parts. This will pay off as the number of alternatives grow, since a traditional representation must store a full document for all parts in each alternative.

Regarding the time overhead to calculate an old revision we can note that this is done on demand on fairly small pieces of information assuming the hierarchical storage is used as intended. The time for the calculations needed should be compared to communication times between the client and the server (where the full node information and necessary deltas are sent as one message) and file access times (where our model gives essentially one read per delta while traditional diffs requires also a file to be opened and closed). We are thus confident that performance for applying deltas will not be a problem in practice even for large applications.

Scalability in terms of number of simultaneous users is also a concern. The overhead for the representation of the possibly large number of alternatives caused by simultaneous users was addressed above. Likewise, we do not expect any performance problems with the active diffs involving a large number of alternatives since diffs are set up when needed and then probably only for a subset of all alternatives. Even if an active diff is set up for a large number of alternatives, the order of magnitude for the response times required still is seconds rather than fractions of a second. Without having any experimental evidence, we expect this modest requirement can be fulfilled. Finally, merging between a very large number of alternatives may be a scalability problem. However, a successive two-way merge of alternatives will probably be practically feasible. Typically, alternatives are created from a main development line (this may be the main development line of the system as a whole or just the main development of the alternative a group currently is working with) and subsequently merged with the main development line again. Furthermore, we expect merging will be a fairly straightforward process, since merge conflicts probably will be reasonably infrequent as a result of the group awareness provided by the active diffs. This expectation is supported by the experience from using the group editor GROVE as reported in [EGR91].

3.6 Implementation Status

The fine-grained revision control model has been implemented on top of an existing tree-structured storage implementation [Gus90]. A corresponding editor for structured text documents also exist as a first application and try-out case. The support for merging in the editor is at the time of writing implemented as another sub-project and there still remains development of full support for merging on the detailed level. Currently multi-server functionality is not supported.

4 FUTURE RESEARCH

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, revisions, and active diff information. 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 finding 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].

5 CONCLUSIONS

We have identified fine-grained revision control as a crucial enabling technique for supporting software evolution by possibly geographically distributed teams of software engineers. The main characteristics of our design includes support for versioning in hierarchically structured information, fine-grained revision history, fine-grained change-oriented diffs, and sharing of common parts among document revisions. The model supports storing of documents, programs, and other tree-structured information. It integrates revision control seamlessly into all levels of a system, the database as well as the applications (e.g. editors and browsers).

The novel benefit of the design is that it enables parallel development by allowing optimistic check-out and providing advanced support for merging alternatives both regarding structure and content. It supports flexible modes of interaction ranging from asynchronous to a form of synchronous work, where members of a team can be made aware of each others' actions without interference with each other.

ACKNOWLEDGMENTS

The authors wants 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, we want to thank Torsten Olsson who is working on the structured document editor and Görel Hedin 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

[EGR91] Ellis, C.A., Gibbs, S.J., Rein, G.L., Groupware: Some Issues and Experiences, *Communications of the ACM*, 34(1), January, 1991.

[FM87] Fraser, C.W., Myers, E.W., An Editor for Revision Control, *ACM Transactions on Programming Languages and Systems*, 9(2), pp. 277-295, April 1987.

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

[HM88] Hedin, G., Magnusson, B., The Mjølner Environment: Direct Interaction with Abstractions, *Proceedings of ECOOP'88, 2nd European Conference on Object-Oriented Programming, Lecture Notes in Computer Science*, vol. 322, Springer-Verlag, 1988.

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

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

[Kat90] Katz, R.H., Toward a Unified Framework for Version Modeling in Engineering Databases, *ACM Computing Surveys*, 22(4), 1990.

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

[KP90] Knister, M.J., Prakash, A., DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors, *Proceedings of CSCW'90, ACM 1990 Conference on Computer Supported Cooperative Work*, Los-Angeles, 1990.

[LCM85] Leblang, D.B., Chase, Jr., R.P., McLean, Jr., G.D., The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts. *Proceedings of the 1st International Conference on Computer Workstations.* IEEE, November 1985.

[LCD+89] Lie, A., Conradi, R., Didriksen T., M., Karlsson, E.-A.: Change Oriented Versioning in Software Engineering Database, *ACM SIGSOFT Software Engineering Notes*, Vol 17, No 7, Nov 1989.

[LO92] Lippe, E. and Oosterom, N. van, Operation-Based Merging, *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, 17(5):78-87, December 1992.

[MM93] Minör, S., Magnusson, B., A Model for Semi-(a)Synchronous Collaborative Editing, *Proceedings of ECSCW'93, Third European Conference on Computer Supported Cooperative Work*, Milano, Kluwer Academic Publishers, 1993.

[MMH90] Magnusson, B., Minör, S., Hedin, G., et al., An Overview of the Mjølner Orm Environment, *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, Paris, 1990.

[MO92] McGuffin, L.J., Olson, G.M., ShrEdit: A Shared Electronic Workspace, Cognitive Science and Machine Intelligence Laboratory, Tech. report #45, University of Michigan, Ann Arbor, 1992.

[MSC+86] Morris, J.H., Satyanarayanan, M., Coner, M.H., Howard, J. H., Rosenthal, D.S.H., Smith, F. D: Andrew: A Distributed Personal Computing Environment, *CACM*, Vol 29, No. 3, March 1986.

[NCK+92] Neuwirth, C.M., Chandhok, R., Kaufer, D.S., Erion, P., Morris, J., Miller, D., Flexible Diff-ing In a Collaborative Writing System, *Proceedings of CSCW'92, ACM 1992 Conference on Computer-Supported Cooperative Work*, Toronto, 1992.

[NSE] Sun Micro Systems. *Introduction to NSE.*

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

[San85] Sandberg, R., The Design and Implementation of the Sun Network File System. *Proceedings Usenix*, June 1985.

[Teamware] Teamware Users' Guide, SunPro Manual set. Sun Micro Systems, Mountain View, To appear.

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

[Tic88] Tichy, W.F., Tools for Software Configuration Management, *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, Germany, 1988.