Version Control for Asynchronous Group Work

Alan J. Dix†and Victoria C. Miles

†H.C.I. Group, Department of Computer Science,
University of York, Heslington, United Kingdom, YO1 5DD.
E-mail: alan@minster.york.ac.uk
Tel: (0904) 432778

Controlling the development of different versions of a document can be a complex task, even for a single author to handle. This task is likely to become more complex as the number of authors increases, and more complex still if those authors are distributed geographically with only limited means of communication, such as electronic mail, to connect them. If this last situation makes version control difficult to manage it also makes it very necessary.
This paper looks at the issue of version control comparing single and multiple user situations. The aim is to focus on requirements for version control that will assist asynchronous distributed group writing. The paper concludes with an informal description of MSC, a multiple-source control system which seeks to fulfill these requirements.

Keywords: *Cooperative work, version control, collaborative writing, computer mediated communication, shared editors, electronic mail.*

1 Introduction

Collaborative document writing has as many forms as there are groups of collaborators. For any given group their collaboration may be at a very fine grain, several heads over the same piece of paper, or very coarse, passing complete versions of the document around from person to person. Our design of a group editing environment [Miles et al., 1991b] tries to support both synchronous and asynchronous working, that is whether or not the collaborators are working at the same time. However, like many groupware systems, it is built upon a local area network and assumes rapid communication between the participants' workstations. Although this can support collaboration within the same building and even at different parts of the campus, it cannot support use over a wide area. Much collaborative writing takes place over a wider area, with communication via e-mail or even floppy disk transfer.

This paper discusses some of the issues connected with collaboration where the participants' workstations communicate over slow infrequent channels such as e-mail. In particular, we will be interested in forms of version control, as a major problem for such work is handling contention for shared objects when there is little opportunity for standard techniques such as centralised data, locking or virtual copresence.

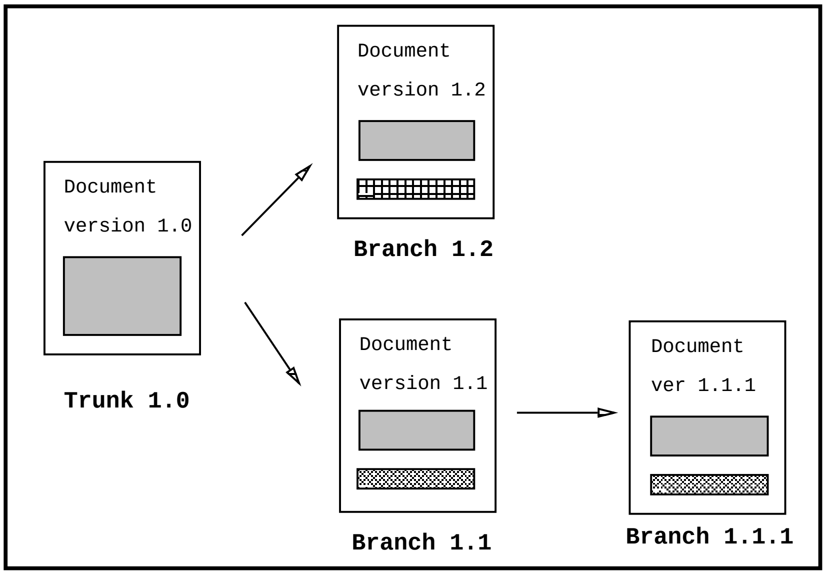

Figure 1: An Example of a Single-User Version Tree.

The work in this paper draws on previous work on electronic conferencing and group editing, but is also part of a study of the general cooperation and information requirements of distributed workers.

We begin, in the next section, by reviewing some of the concepts of 'traditional' version control systems and how they relate to multi-user situations. In Section 3, we look at how version control issues are handled in existing shared editing systems. Finally, in Section 4, we produce a set of requirements for a distributed version control system for group work, and introduce designs for such a system in the form of 'MSC' a multiple-source control tool.

2 Why version control is important

I want to write a document with a colleague. We work in different places and have to send each other the document text by electronic mail. We encounter several problems to do with version control as we try to produce the collaborative document.

I send my co-author a copy of the document I have written. She receives it and decided to make some changes. I too decide to rewrite part of it, and take a copy to revise. When my colleague sends her revisions back, I find I have two different documents. I have to establish what changes have been made by both parties, and how to merge them. If the changes are non conflicting then this is not a problem. If they conflict then I must make some judgement on which text to include and which to ignore

Meanwhile unaware of my deliberations my co-author is busy making yet more changes to her copy of the document. To her, this is the most up-to-date version of the document and therefore the copy she must work on. When she sends her revisions back to me, the problems of putting together a coherent 'official' are exacerbated.

We clearly have a problem with multiple versions of the same document. Can we use the language of traditional version control systems to help us?

2.1 Single user version control

If I, as a single user, want to produce a document using my computer, there are various stages that I can go through.

- At the start, when there is nothing yet written, I will *open* a new file giving it a name that identifies the intended content, e.g. doc1.0. The effect is to create a new *version*.
- If there is some text already produced I can *reopen* the file containing the document and edit it. If I choose this option then I am *revising* an existing version.
- If I want to revise a version, but also keep a copy of the pre-revision document, then I will *copy* the original version, perhaps calling the copy doc1.1 and revise the copy.

Even with this simple example one can distinguish between different types of version. Tichy [Tichy, 1985] in his description of RCS a revision control system classifies these different types of version in terms of their position in a tree structure. There are 'trunk' versions and 'branch' versions. RCS is designed to assist in the organisation of versions of software, so an RCS trunk version refers to software releases, (e.g. release1, release2 etc). A branch version is added when a fix is needed for an early release, but development has moved on to a new release. Branches from release 1 would be numbered as, for example, 1.1, 1.2 etc.

In terms of the earlier document production example, a trunk version will be created when I open a new file. A trunk version will be revised when I reopen that file and make changes to it. When I make a copy of that file and revise the copy then I have created a branch version. Consider figure 1. The left-most document is the original version, and is therefore a trunk version. Branch version 1.1 and branch version 1.2 are formed by by taking copies of the trunk version and making revisions to them. Branch version 1.1.1 is formed by revising a copy of version 1.1.

Not all version control systems allow such branching, but it is essential to understanding the multi-user situation. We can now revisit the multi-user example using the language of version control.

2.2 Version control and multiple users

Recall the scenario described at the start of this section. Two distributed authors are trying to write a document together using e-mail as the communications medium. The problems they encounter are illustrated in figure 2. Author 1 and author 2 each make independent revisions to the original trunk version. The effect is to produce two branch versions (1.1) which may contain conflicting revisions. While author 1 tries to disentangle conflicting revisions, author 2, unaware that her version is now out-of-date, makes more revisions to her copy of the document, forming branch 1.1.1.

At first sight there is little to distinguish this from the single user case in figure 1: they both represent a branching version tree. However, there are important differences. For the moment we shall ignore the obvious problems of physical distribution, instead concentrating on the more subtle issue of the *meaning* of the branches in the two figures. In figure 1, the single user case, the branching was intentional, perhaps to have one version of a paper for an internal report and another for publication. In the multi-user case, the *intention* is to have a single document; it is the logistics of communication which induced the branching. Therefore the intent is that eventually the branches should remerge.

In most version control systems, even where branches are allowed, there is no facility for remerging them. Once development paths diverge they are considered altogether separate. Thus we have already a major difference between traditional version control systems and those for distributed collaborative work.

In fact this merging of versions is a complex thing in itself, supporting it at the level of the version control tree is probably the least of our problems.

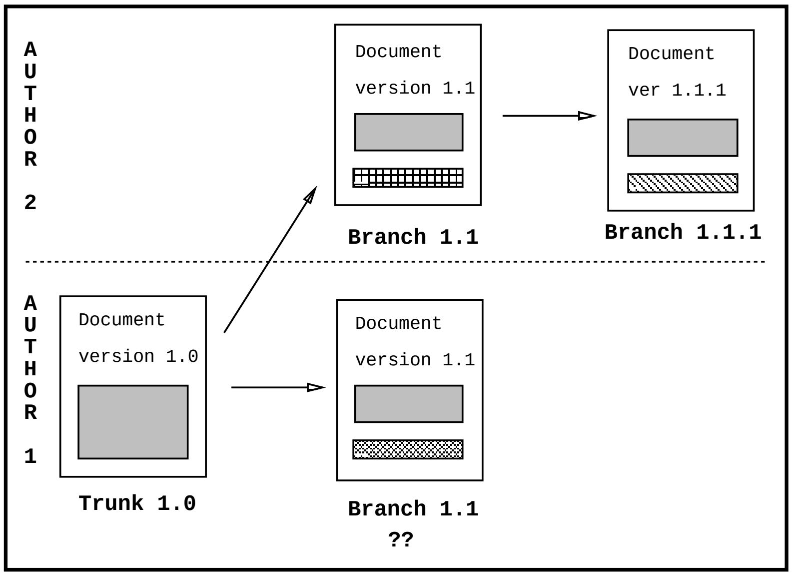

Figure 2: An Example of a Multi-User Version Tree.

2.3 Merging versions

Consider the following as a simple example of conflicting revisions. I send my co-author a copy of a document for her to work on. Meanwhile I make a copy of the document in order to make some of my own alterations. My colleague sends the document back. She has deleted a paragraph of text. I examine the revisions I have made and find that some of my revisions are in that paragraph. Which version then should I take as the new 'official' document. In what order should the changes take place? This is the problem of *synchronisation*. Synchronisation involves the coordination of actions, with respect to time. Synchronisation ensures that actions on a shared resource are performed in a particular order, e.g. FIFO.

The problem of synchronisation is one which is faced by designers of distributed computing systems. Distributed systems can be examined in terms of distributed hardware, distributed control and/or distributed data. Taking the collaborative document example, distribution occurs in each of these three areas. In terms of hardware, both authors each use their own computer with its local memory and processors. Data is distributed since there are multiple copies of the document at each site, (data replication). Control is distributed in that each author has complete autonomy over their local copy of the document.

In order for the two authors to collaborate in the writing of the document, there must be some interconnection between them. This is provided by the e-mail link. E-mail is suffers from a number of problems, it can be slow, it can fail, particularly when used over a wide area. The effect, is to have an interaction between the authors which is *asynchronous*. This means that there is a time lag between the sending and receiving of messages. The nature of e-mail is such that this time lag can be highly variable in length.

In distributed database design, time stamping data ensures that the most recent entry will be the 'correct' entry. In distributed writing this maxim may not hold: a newer version does not supersede an older one, both may have been updated. For an example of this we turn to the related area of programming.

Once one of the authors found a bug in a part of a program which he thought had been fully tested many months previously. The program had been developed on two machines, copying versions back and forth between them. After much searching he found the bug, but he remembered finding it before. Sure enough, looking back with the aid of a (standard) revision control tool, the correction was there. It had been corrected on one machine, but the corrected version had been overwritten with a more recent update from the other machine. He had overwritten when he should have merged.

Obviously we need to know what is the last common version before we can decide which version to choose or whether we need a more extensive merging of the two. Ideally we would like support in this activity. There are some tools to support this, highlighting differences between two versions, however as users we should probably still like to compare the two with their latest common version. In some cases the system may be able to perform the merge for us, but even where an automatic merge is possible it may not be desirable. Choosing between conflicting revisions of a text is a subjective exercise, involving criticism and consideration of the overall coherence of text.

2.4 Further problems of distribution

We have seen some of the confusion that asynchronous, distributed work can cause. As one of two distributed authors I find it difficult to keep track of what versions exist. I find that there are multiple versions with contradictory changes, and I cannot establish the ordering of those changes. I do not always know what the 'official' version is, and find myself working on an out-of-date version. I find that I cannot always establish which changes have been made by whom. Of course, this complexity is increased, the more authors there are.

There is considerable interest in the Computer Supported Cooperative Work field (CSCW) in shared editing applications. Shared text editors allow two or more people to write together using a shared environment. Some shared editing applications support synchronous interaction only, some asynchronous interaction only, and some support both modes.

The vast majority of shared editing applications are designed for use over local area networks (LANs). Therefore architectures are not distributed, and the problems of controlling multiple copies of the same document are more easily addressed. A frequently favoured architecture for shared editor applications is the Client-Server architecture. Here the server acts as a central administrator, handling client access to shared objects. The server can enforce mutual exclusion, restricting access to an object to one client at a time. With a distributed architecture the idea of centrality is lost. With no central coordinator enforcing mutual exclusion, several people can work independently, on distributed copies of a document, and the problems outlined earlier can occur.

One can think of the difference between central and distributed data in terms of a shared data structure versus message passing. The notion of a shared data structure can be thought of as the blackboard model [Brehmer, 1991]. Here, the shared data structure acts as a blackboard upon which messages and results are written and where everyone turns for information. Such an architecture ensures that every agent in the systems has access to the same information. In the case of distributed data, each agent in the same has a copy of the data structure, and changes to it must be broadcast, via message passing, to other agents so that their copy of the data structure can be updates. If the message passing medium is slow, as e-mail can be, the the potential for inconsistency is obvious.

What devices are employed by existing CSCW shared editing applications help with multi-user version control? Can we borrow ideas from them for highly asynchronous, distributed version control?

3 Version control in shared editing systems

This section looks at version control techniques in existing CSCW systems. The aim is to evaluate these techniques in terms of their suitability for distributed version control.

Looking at the literature on shared editing systems one is struck by the lack of any specific help with version control. Many shared editing applications are for synchronous interaction only, where participants interact in the same time period. One example of such a system is GROVE, a shared outliner. Ellis, Gibbs and Rein, describe users having a 'session', a period of synchronous interaction, with GROVE. It is assumed that version control will not be required for a single synchronous session. Instead, GROVE and other concurrent shared editors such as ShrEdit and Aspect, use techniques such as locking and rapid updates to assist with concurrency control.

These concurrency control techniques can be thought of in terms of version control, because they deal with the same problems of interference that were discussed earlier. For example, in a synchronous shared editor, locking out all but one user from editing a line of text, ensures that potentially conflicting edits are avoided. The problem of conflicting revisions was one faced by the example of the two authors in the previous section. Rapid updating of each user's text ensures that all participants see the most up-to-date text. When updates are not visible between authors, as was the case with the two distributed authors, then there is the potential for participants to be working on out of date copies.

Thus we see the relationship between concurrency control techniques used in synchronous shared editors, and version control in asynchronous distributed collaborative writing. The relationship can be considered in terms of *grain size* and *duration*. Concurrency control mechanisms such as locking, can be applied at to a line of text in a shared text editor (ShrEdit, MULE), this would be a fine grained application. It could be applied only as long as a user is actively editing that line, therefore the duration is likely to be short. When authors are distributed and communication is highly asynchronous, locks can be applied to the whole document, to ensure that only one person can make changes to it. This would be a coarse grain application of locking. In this example the duration is likely to be long, since the author has the complete text to work on and the inadequacies of the e-mail connection to contend with.

Let us now take a closer look at concurrency control, visibility and specific version tools that have been demonstrated in existing shared editing systems.

3.1 Concurrency control

To control concurrency is to stop two or more events happening at the same time. In shared editing the aim of currency control is to stop two or more revisions happening at the same time, to the same text. As mentioned earlier concurrency control can be applied at different levels and for different durations. Therefore systems can restrict concurrent access to a line of text in a shared editor, or the complete document. Similarly, access can be restricted from a faction of a second to days.

There are various ways that concurrent access can be restricted. These are discussed below.

Locking or mutual exclusion

Input locking *closes* access to a shared resource, by allowing only one user or process to change it. Thus imposing locks restricts interference between multiple agents. Interaction with the shared resource can therefore take place under similar conditions to those inherent in single user applications.

Locking data before it is written is a concurrency control mechanism that is commonly used in the design of group editing applications. In single entry-point applications control of the single cursor means control of the entire text editor, restricting other users from changing any part of the text [Olson et al., 1990]. The cost of restricting access to data in this way, is to reduce the shared context which facilitates understanding within interaction [Suchman, 1985]. The need for simultaneous access to shared data has been recognised in the development of multiple entry-point applications. Such systems require locking at a finer grain, so that different users may simultaneously change different pieces of text in a shared editor. The grain size of the lock varies between applications, from section or paragraph, to sentence, word or character. In single entry-point applications, or multiple entry-point applications with large grain locking the duration and relinquishing of locks becomes an important issue. If an application is slow to respond to a request for a lock then this can hamper the whole group-authoring task. With an e-mail connection this lag is extreme.

Consider the following scenario. Three authors are writing a document together using e-mail as the distribution medium. Author 1 wants to have unique write access to the document. He must send an e-mail message to author 2, and author 3, asking for unique write permission. They should then send back their replies. If both agree to allow the lock, then author 1 can revise the document knowing that the others will desist from editing until he reports that he wants to relinquish the lock. When he receives permission to lock, author 1 should write to the other two authors informing them that the lock is to be applied. He must let them know since since the lack of communication between authors 2 and 3 means that they will not know whether the other sanctioned a lock or not. If author 1 broadcasts that he is permitted unique write access, then authors 2 and 3 will be informed of the current state of the document. Of course, without any specific system enforcement, agreeing to a lock means a mutual agreement that only one person will edit the document.

Application of a protocol

Concurrent access can be avoided by applying a protocol to the interaction, which sequentialises access the a common resource. 'Round robin' provides an example of such a protocol. Protocols can be enforced by mutual exclusion. Thus the effect is to lock access to a shared resource to one person at a time, in a given order.

Returning to our three authors writing a document by e-mail, one can envisage a baton passing procedure to enforce a round robin protocol. For example, when author 1 has completed part of the document, he sends the document to author 2. Author 2 works on the text and then forwards the document to author 3. Once author 3 has completed his revisions to the text, the document is sent back to author 1.

This protocol might be efficient in terms of avoiding interference and conflicting revisions, but it is not conducive to the collaborative writing process. The authors are forced to 'wait their turn', and this may stifle creativity, and frustrate the writing effort.

division of labour

Another means of avoiding interference is to employ a division of labour policy. Division of labour can be by social role. Consider the case of our three distributed authors. Let us say that author 1 is designated 'editor', and authors 2 and 3 are 'reviewers'. In this case, author 1 will be responsible for writing the document, while the other two authors will be restricted to making comments about the text. It will be up to the editor, author 1, to assimilate reviews from his colleagues and incorporate them into the text as he feels fit. The division of labour policy is used in the QUILT collaborative writing system [Leland et al., 1988].

Division of labour can also be based on a model of the document under preparation. Here the authors decide to split the document between them, each author writing a different part. This document model is a favoured method of collaborative writing according to the work of Posner and her colleagues [Posner et al., 1991], and has been adopted in the MultimETH system, a multi-media conferencing and collaborative writing system [Lubich and Bernhard, 1990].

liveware

Thimbleby and his colleagues have produced a mechanism called Liveware [Witten et al., 1991] for handling distributed group databases. Described as 'personal distributed CSCW', Liveware uses an analogy with computer viruses to transmit information from site to site. Versions of the database are copied onto floppy disks, and when a floppy disk is put into a system with the same Liveware database, the two versions compare themselves with one another. Information is updated by the use of timestamps. For information such as an address book the most up-to-date information is all that is required, but, we have already seen that this is not sufficient for collaborative writing where newest does not necessarily mean correct. Because of this, the Liveware system currently implemented using HyperCard [Apple, 1987] normally requires each card to have a single owner, who alone can make modifications. Thus although update is distributed for the stack as a whole, it is centralised for each item. Despite these limitations, Liveware is unique in its handling of highly distributed group information.

dependency detection and threads

Dependency detection is based on the observation that contention is relatively rare – users are unlikely to edit exactly the same place in a document. In the context of synchronous groupware keeping a strictly WYSIWIS (what you see is what I see) view of the document requires vast amounts of network traffic transmitting each and every user action, and subsequent delays at the interface. However, given most actions are not conflicting it is possible to update each user's copy of the document instantly and only transmit information about updates in blocks (probably still quite frequently, every few seconds or so). Normally such transmissions would require no acknowledgement. Only when two actions which potentially conflict happen concurrently does the system react – after the event – warn the user's of the conflict and take some default corrective action, perhaps undoing the action of one or both commands. In the context of synchronous work the granularity of such a correction is small and deemed acceptable.

Although these mechanisms are used over closely coupled distributed systems they are very similar to the issues regarding multiple versions for more widely distributed working. We call these very short lived, light weight versions of the document *threads* by analogy with light weight processes. This concept has proved useful in analysing the behaviour of synchronous editing and especially the problems of undo in shared editors [Abowd and Dix, 1991]. A version thread can be produced automatically when the system detects that divergence between versions is likely. Periodically the system can resynchronise threads of the same object, (merge them) only querying the users when its rules fail. Threads may be eager – only allowing known independent actions (such as character by character typing and deleting) on each thread, but forcing synchronisation before potentially dependent actions such as cut and paste. Alternatively, they may be lazy – allowing any update to the individual versions and only alerting the users to dependencies when periodic updates occur. Eagerness requires synchronous activity with efficient communications. Distributed work makes eager resynchronisation impossible. The divergence between threads is likely to be larger in the lazy case, and the users would have to do more work to deal with conflicting revisions.

The application of dependency detection and threads to highly asynchronous, distributed work has some problems. Conflict resolution cannot take the form of throwing away updates as the grain of update is too large. However applying an automatic merge will usually not be sensible. Even if different author's revisions were non-conflicting, the problem of the coherency of the text must be addressed. It is unlikely that the merging of independent revisions of a common text, would produce a document that read smoothly. There are also difficulties inherent in the manual intervention required to deal with conflicts. Who should decide on which revision become part of the 'official' version? If negotiation is to be conducted over e-mail, then this could become a very protracted exercise, possibly holding up the rest of the writing task.

3.2 Visibility

In synchronous cooperative work the effect of visibility is to show up concurrent access, rather than control it, although concurrent control mechanism such as locking and rapid update are most often used together.

Visibility can be applied to asynchronous work too, in the form of change tours, showing updates on replicated data. Pendergast and Beranek, are members of a team researching collaborative workstation design. In their paper [Pendergast and Beranek, 1991], they suggest that groupware must allow both synchronous and asynchronous interaction, and that asynchronous features should include *version control* and *change tours*. Pendergast and Beranek argue that in a shared work environment versioning should allow the group to keep track of the sequence of changes and the persons responsible for those changes. They suggest using version control information to replay changes in sequence, 'change tours', as a means of keeping all project members up-to-date on all changes that have been recommended or made.

Pendergast and Beranek discuss the implementation of version control and change tours. For version control they suggest a change log, which records each editing change. Each entry in the log has the change itself, the time of the change and a user id. This is exactly how the York group editing environment implements its current versioning feature. To implement the change tours, the change log can be scanned, and changes, that have occurred since the participant was last using the system, replayed. Again this is how the replay facility in the York group editing environment works.

3.3 Versioning tools

There are surprisingly few explicit versioning tools for shared writing systems. One system which does provide versioning support is a shared program file editor described by Beaudouin-Lafon in [Beaudouin-Lafon, 1990]. Beaudouin-Lafon introduces 'the embedded context-dependent version model', which is presented as a way of supporting collaborative software development A document is defined as a hierarchy of fragments. Fragments are supposed to represent meaningful entities of the document. Each fragment can have an arbitrary number of versions. These versions are seen as part of the document, therefore they are called 'embedded versions'. Each version had identifiers like who wrote it and a version number. When opening a document, a user can see one version of each fragment, which is dependent on the 'context' of the display of the document. A user can have: the 'official' version which is the original one; the version of a given user; the user's own version. By default, when a user asks to see his own or another user's version the most recent is shown. This interesting approach to software version control is feasible because a centralised data structure can be maintained, where the context-dependent versions can be stored and accessed.

3.4 Summary

We have looked at various techniques for handling versioning in group work. Use of protocols and agreed division of labour can be applied virtually directly to distributed work. By their nature they are coarse grained both in the frequency of communication and the size of document unit. Their only demands upon tool support is that the transmission of document versions from person to person is easy to use and efficient in communication costs.

Locking can be used, but the low frequency of communication means that user's must look ahead. Even a simple locking protocol, such as in our example, takes at least one message send and return before the writer can continue work. For e-mail communications this may mean several hours between requesting the lock on a section of document and being allowed to proceed. This is likely to put some break on the writer's creativity!

Given the slow rate of interaction yielded by any of the above means, the users are likely to resort to additional means such as making copies of the document, editing it and then manually merging their version when the baton passes to them, or when a lock is acquired. Thus even if the users choose to use various conflict avoidance measures the system should support the inevitable divergence of versions.

We thus propose that distributed group editing be built over a distributed versioning tool allowing divergent versions of the same document, merging of versions and browsing facilities for users to 'catch up' on one another's activity.

4 A Multiple Source Control Tool

In this section we make some suggestions for a multiple source control tool to support asynchronous, distributed version control. Multiple source control (MSC) is a system for maintaining source (unstructured text) documents at different sites. Editing is expected to take place at these sites asynchronously and the job of the MSC is

- to maintain version information distributed over the sites;
- to synchronise that version information periodically;
- to supply the means whereby users can easily merge different revisions to the same version produced at different sites.

For a detailed formal specification of the MSC internal structures see [Dix, 1991a]. The rest of this section describes MSC informally and discusses how it relates to the requirements outlined so far in this paper.

4.1 Underlying structure and communications mechanism

The fundamental data structure underlying MSC is a version 'tree' at each site; a site in this context being a group of closely coupled user workstations. The trees allow both version branching and merging, so are strictly directed acyclic graphs (DAGs), but we will continue to use the term tree. At any moment the trees at each site will contain different information, but will be compatible in that they will agree up to some point, and then differ where new updates have occurred at different sites. These versions will be constantly converging as MSC transmits information back and forth between sites, but the trees will not agree entirely unless all the users are quiescent.

The different sites are assumed to be linked primarily be e-mail, although we do intend to support floppy disk transfer as well. The system's communications will be in reasonably large chunks, these will happen periodically (perhaps once a day) but also when the users explicitly request synchronisation (for instance, if they want to discuss the latest changes over the telephone). At some sites, the e-mail facilities may allow direct channelling of appropriate messages to MSC, at others the users may have to recognise the MSC message from its header and invoke the MSC receiving tool by hand. Where the latter applies, it is especially important that messages are infrequent and in large chunks as we do not want to put a burden on the user.

In common with all version control systems the actual storage of the tree will be based around deltas which record the difference between successive versions. This is normally for storage efficiency at a single site, but also has the advantage that it allows easy transmission of incremental changes between sites. Deltas come in two forms: forward deltas, which say how to get from an older version to the next one, and backward deltas, which allow you to move from a newer version to the previous one. For example, a forward delta needs to store the contents of an insertion but only the extent of a deletion. On the other hand, a backward delta need not store the contents of an insertion, merely where it is, but needs to know what was deleted in order to restore the deleted text. There are arguments for each form of delta, but for various reasons, some described below, MSC will store both forward and backward deltas.

4.2 Branches and threads

Some of the branches in the version tree will be 'real' ones representing different versions of the document for different purposes. These are not interesting in the present context as they represent a standard version control issue. More important are those which come into being to support divergence between sites. These we also refer to as *threads*. They are slightly more heavy weight than the threads described for synchronous working but share the same essential feature: threads are branches where the *intention* is that they remerge. Whereas real version branches represent different objects (e.g. internal report vs. publication) threads are temporarily different versions of the *same* object.

Threads should be easy to produce, or automatically produced. In fact we would expect the system to maintain as a normal practice one thread per site (`doc@site1`, `doc@site2`, etc.) and perhaps one thread representing the 'official' version of the document. Although the threads are labelled with the sites, each site must maintain the threads for the other sites as part of its copy of the version tree: its own thread is merely where it by default adds its own updates. At any time a user at a site may (with system help) notice that there are two divergent versions of the document and merge the updated one with her own thread.

Possibly the responsibility for such merging, especially if there is also an official thread will be delegated to one individual (the editor) or possibly this responsibility may be delegated. We do not wish to force this policy but allow the participants to develop their own – that is we adopt local as opposed to global structuring [Miles et al., 1991a].

4.3 Merging

Merging should also be light weight and easy, but we have seen that this is not always possible. MSC aids this process as much as possible. As different threads of the same object are intended to merge, MSC will highlight such divergences to the user. In the simplest case, the version generated at another site is a direct descendant of the current version at mine. In this case the default action is to simply regard this as the up-to-date version for my site – there is no conflict.

In the case where real conflict occurs, the version tree information allows the system to trace back to the last common point. So, unlike a standard difference tool, we can see not only the difference between two versions, but where they originated. If the two differ at a paragraph, but one is simply the same as the original, we are likely to just accept the change. However, if both paragraphs have changed we know we require more serious thought. The user is aware of all changes, but has attention focused on the most important.

Furthermore, the version graph can be annotated alongside each delta with reasons for changes, this is normal even in single-user version control systems, but in distributed collaborative writing this forms an important communication between the authors. In the context of merging it means that they can understand the reasons for each change and thus produce a better combined document.

4.4 Annotations and change tours

As we just said the transitions between versions will allow annotations by the authors as to why a particular change has been made. Thus it can be seen as an extension of the conversation transcripts which we already use in our existing group editing environment [Miles et al., 1991b].

Such annotations will also be useful during change tours. Recall that these were a way for asynchronous users to catch up with one another by means of a play back of each other's changes. As the user browses back and forth through the versions to the document, she can see the reasons for each change. Because the version tree includes both backward and forward deltas, it is easy to browse backward and forward through the versions. We can easily supply a video-recorder type interface similar to that in our existing group editor.

In [Miles et al., 1991b] we also suggested that annotations be allowed on the document text in the form of balloons or post-it notes. These are particularly useful in the distributed context as they are a *monotonic* communication media and they do not require serialisation as a text transcript would – we don't want the users having to worry about version control of their conversation! To support such notes we require mappings between the positions in different versions of a document, even when there have been branching and merging in between. This facility is just what dynamic pointers [Dix, 1991b] provide, and the appropriate *pull* function, mapping positions between successive versions, will be included with the deltas.

A crucial feature in any collaborative writing tool is the ease with which the authors can integrate their discussion of the work with the document itself. They need to be able to establish context in the document and include deictic references in their conversation. Thus annotations and similar facilities are not an optional extra, but an essential part of MSC.

4.5 User interface

By its nature MSC is likely to run on a variety of platforms. We will initially produce two interfaces: one a command based one, with tools for various tasks, similar to a traditional version control tool, the other a graphical interface under X utilising our existing group editor interface, with the addition of the relevant extra functionality.

Two particular interface issues are obvious. One is how to make the branching version tree information easy to comprehend. This is not perhaps as difficult as it may seem, as in the absence of 'real' branches, the threads form a nearly linear development path. However, we may want to allow facilities such as replaying the document from a particular site's viewpoint (not necessarily your own) or graphical views of the version graph.

Another issue is the presentation of difference information during thread merging. Various visual difference tools exist which present two versions side by side highlighting differences, but we suggested that it was useful to also present the common source. Does this mean we need a triple A4 screen? One option is to present the two versions side by side, only showing the common source on demand. However, whereas a normal tool would merely be able to note that the two differed at some point, the MSC based tool would be able to annotate the two versions by their relation to the common source. For instance, we may have a paragraph that is different in both, but is marked in one as updated. We thus, as in the scenario presented earlier, are able to decide more easily which is appropriate in the merged version.

5 Conclusion

We have seen that problems of concurrent access in highly asynchronous, distributed group editing are best understood in the light of version control mechanisms. We have reviewed techniques from other CSCW systems and shown how our proposed distributed version control tool MSC deals with various issues. In particular, it allows divergent versions to be produced by users at different sites, but supports the remerging of these versions. It also supports annotation for communication between participants and browsing through your own and co-author's old versions, both important in supporting close collaboration.

6 Acknowledgements

This work was carried out whilest Vicky Miles was at the University of York, funded by SERC grant GR/F/01895. Alan Dix is funded by a SERC Advanced Fellowship B/89/ITA/220.

References

[Abowd and Dix, 1991] Abowd, G. D. and Dix, A. J. (1991). Giving undo attention. Working paper.

[Apple, 1987] Apple (1987). *HyperCard User's Guide.* Apple Computer Inc., Cupertino, CA.

[Beaudouin-Lafon, 1990] Beaudouin-Lafon, M. (1990). Collaborative development of software. In Gibbs, S. and Verrijn-Stuart, A., editors, *Multi-User Interfaces and Applications.* Elsevier Science Publishers B.V. (North-Holland).

[Brehmer, 1991] Brehmer, B. (1991). Distributed decision making: Some notes on the literature. In Rasmussen, J., Brehmer, B., and Leplat, J., editors, *Distributed Decision Making Cognitive Models for Cooperative Work*, News Technologies and Work, chapter 1, pages 3–14. John Wiley and Sons.

[Dix, 1991a] Dix, A. (1991a). Multiple source control. Working paper.

[Dix, 1991b] Dix, A. J. (1991b). *Formal Methods for Interactive Systems.* Academic Press, London.

[Leland et al., 1988] Leland, M., Fish, R., and Kraut, R. (1988). Collaborative document production using quilt. In *CSCW'88*, pages 206–215.

[Lubich and Bernhard, 1990] Lubich, H. and Bernhard, P. (1990). A proposed model and functionality definition for a collaborative editing and conferencing system. In Gibbs, S. and Verrijn-Stuart, A., editors, *Multi-User Interfaces and Applications*, pages 215–232, Amsterdam, The Netherlands. Elsevier Science Publishers.

[Miles et al., 1991a] Miles, V., Johnson, C., McCarthy, J., and Harrison, M. (1991a). Supporting prediction in complex dynamic systems. In *Proceedings of HCI'91*.

[Miles et al., 1991b] Miles, V. C., McCarthy, J. C., Dix, A. J., Harrison, M. D., and Monk, A. F. (1991b). Exploring designs for a synchronous-asynchronous group editing environment. In *Proceedings of the Workshop on Collaborative Writing*. Springer-Verlag. In Press.

[Olson et al., 1990] Olson, J., Olson, G., Mack, L., and Wellner, P. (1990). Concurrent editing: the group's interface. In Diaper, D., Gilmore, D., Cockton, G., and Shackel, B., editors, *Human-Computer Interaction—INTERACT'90*, pages 835–840, Amsterdam, The Netherlands. Elsevier Science Publishers.

[Pendergast and Beranek, 1991] Pendergast, M. and Beranek, M. (1991). Coordination and control for collaborative workstation design. In Diaper, D. and Hammond, N., editors, *People and Computers VI - Proceedings of the HCI '91 Conference*, pages 157–167. British Computer Conference Series, Cambridge University Press.

[Posner et al., 1991] Posner, L., Baecker, R., and Mantei, M. (1991). How people write together. Technical report, Computer Systems Research Institute and Department of Computer Science, University of Toronto, 6 Kings College Road, Toronto, Ontario, M5S 1A1, Canada. Submitted for Publication to ECSCW '91.

[Suchman, 1985] Suchman, L. A. (1985). Plans and situated actions : The problem of human-machine communication. Technical Report ISL-6, XEROX Research Center, Palo Alto.

[Tichy, 1985] Tichy, W. (1985). RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654.

[Witten et al., 1991] Witten, I. H., Thimbleby, H. W., Coulouris, G., and Greenberg, S. (1991). Liveware: a new approach to sharing data in social networks. *International Journal of Man-Machine Studies*, 34:337–348.