

An Integrated Approach to Distributed Version Management and Role-  
based Access Control in Computer Supported Collaborative Writing

Byong G. Lee  
Department of Computer Science  
Seoul Women's University  
Seoul  
South Korea  
byongl@swu.ac.kr

N. Hari Narayanan and Kai H. Chang  
Department of Computer Science & Software Engineering  
Auburn University  
Auburn, AL 36849  
USA

Appears in *Journal of Systems and Software*, 59(2): 119-134, Elsevier Science Publishers, 2001.

# An Integrated Approach to Distributed Version Management and Role-based Access Control in Computer Supported Collaborative Writing

## **Abstract**

Tools to support Computer Supported Collaborative Writing (CSCWriting) allow multiple distributed users to collaborate over a wide area network on constructing a shared document. Prior research on Computer Supported Collaborative Work (CSCW) in general has predominantly focused on synchronous collaboration. Network latency becomes a bottleneck in maintaining shared artifacts during synchronous collaboration. Besides, to enable truly cooperative work asynchronous modes need to be supported as well, so that mobile users can switch between synchronous and asynchronous modes while they disconnect and reconnect to the network. These two considerations motivated the development of a distributed version control system for CSCWriting described in this paper. The most important contribution of our work is the proposal of an Activity IDentification (AID) tag as the fundamental mechanism to support distributed management of multiple versions of a document. The AID tag facilitates the design and implementation of an integrated approach that includes differencing, merging and role-based access control at different levels of granularity, maintaining and visualizing the version structure, and group awareness of document status and operations. The AID tag leads to simple and effective differencing and merging schemes. Its unique address scheme eliminates the need for large storage capacity for version maintenance. Role-based access control can be implemented by associating the access right table and role assignment capabilities with the AID tag. Information for providing group awareness of the changing document is available from the AID tag. In addition, since the system maintains a user-browsable version structure of the evolving document that incorporates AID tag information, any user collaborating in the authoring of a document can easily visualize the historical evolution and current context of the document.

## **1. Introduction**

With the widespread penetration of the Internet and globalization of business, it is quite common for groups of remotely located individuals to collaborate on creating documents. Such collaborative writing may switch between synchronous phases when all members are on-line and working on the document and asynchronous phases when at least some members are off-line and working separately on the document. During collaborative writing, many variants of a document can be created and revised by the group members. Maintaining consistency of the document in these situations becomes a critically important function of the computer tools being used to support CSCWriting. Shared documents that allow shared editing (Ellis et al., 1990) provide one solution to this problem. However, this solution precludes asynchronous and individual work unless off-line members are forced to merge their individual versions with the shared document every time they reconnect or come back on-line. Another problem with maintaining a shared document in a central server is that delays introduced by network latency can become a bottleneck that negatively affects response time and thus degrade the quality of collaboration (DeCouchant et al., 1996). One approach that reduces the dependence of system response time on network latency and at the same time allows both synchronous and asynchronous collaboration is to use a version control and maintenance system that supports differencing and merging of multiple versions of the same document. This allows group members to check out and work on their own local versions of the document. After they finish revising their local versions, these can be checked in.

Dewan and Munson, (1995) defined a version control system as one that manages dependencies among different variants of the same document and organizes

the variants into meaningful structures such as a sequence or tree. With the support of a version control system, group members may copy and manipulate each variant (version) of the document, and then produce a single shared document by merging them. Later, multiple versions may again be generated from this shared common version with another merge point occurring afterwards. The last such merge will result in the final version of the document. The evolution of any particular version can be determined by differencing and tracking version dependencies. Version control techniques developed for single user environments, however, can not be directly applied to collaborative writing environments. These techniques lack mechanisms to maintain and propagate the history of individual actions across versions, to compute differences between versions, and merge versions. Furthermore, in a group setting the merging of different versions from individuals require decisions on document components at different granularities (e.g. words, sentences, sections) and resolving conflicts (e.g. one member changed a sentence while another deleted the same sentence) based on priorities determined by the various roles of group members (such as author, critic, etc.). Thus, version control in a CSCWriting context needs facilities to manage different access privileges and conflict resolution policies at merge time for members of the group. Such facilities are not available in typical single user version control schemes.

The main goal of this paper is to present new techniques to manage document revision history, extend the traditional differencing and merging schemes to allow multiple granularities and role-based access control needed for a CSCWriting environment, and integrate all these into a single seamless architecture based on a unique addressing mechanism called Activity IDentification (AID) tag. We also discuss how the proposed AID scheme improves version management and maintaining group

awareness of the document's evolution. The remainder of the paper is organized as follows. Section 2 discusses a number of important issues that a tool to support CSCWriting needs to address, and extant literature on each of those issues. Section 3 describes the design and implementation of a version control system based on AID tags. In section 4, we present a detailed case study of the use of the system in a collaborative writing context. Section 5 contains a formal analysis of the AID tag approach to version control. Section 6 discusses limitations of the proposed approach and how these will be remedied in future work. Section 7 concludes the paper with a summary of the implications of this research.

## **2. System Requirements for Document Management in CSCWriting**

A system that helps group members to manage the creation and maintenance of the evolving document must provide the following capabilities.

### **2.1 Differencing**

One fundamental component of a version control system is a differencing mechanism that finds differences among versions. Such a mechanism is necessary to facilitate the merging of different versions of the document into a common one. There are three common approaches to finding differences between two versions of a document. The first and the most popular approach is to compare plain text and find the longest common sub-string in the two versions. Then the locations of any changed strings are computed as numeric distances from the common sub-string (Munson and Dewan, 1994). Most differencing tools, such as Diff3 (Munson and Dewan, 1994), RCS (Tichy, 1985), and Flexible Diff (Neuwirth et al., 1992), employ this approach.

The second approach is to use semantic differences by constructing a dependency graph for each object in the document, merging these into one dependency graph, and then checking for differences (Munson and Dewan, 1994; Zhang et al., 1996). Timewarp (Edwards and Mynatt, 1997) utilizes this approach. However, both plain text comparison and semantic differencing impose a large time and space overhead on the system as the number of versions to be compared increases. The third approach, adopted in the Multiple Source Control (MSC) approach (Dix and Beale, 1996), is to utilize the command history. In this approach, differencing is done by exchanging and comparing the histories of document editing commands being maintained in the different versions (Munson and Dewan, 1994). However, this requires that when elements are created anew (e.g., a word, sentence, or paragraph) adjacent elements are forced to change their addresses so that consistency can be maintained among versions. This address changes then need to be propagated through the entire document. Such address space updating is both computation and memory intensive as the history grows in length (Dix and Beale, 1996).

## **2.2 Merging**

A merging mechanism that in a semi-automatic way merges two different versions of a document is highly desirable for CSCWriting. Fully automatic merges are difficult to implement since semantic considerations that come into play when resolving conflicting edit operations on a document component are difficult to automate. Fully manual merges require users to create the merged versions manually, with the system providing no support for detection or resolution of conflicts. Semi-automatic merging represents a viable middle ground between these two extremes. Another reason to

prefer a semi-automatic merging mechanism is that merges are done interactively, allowing group members a say in resolving edit conflicts that cannot be automatically resolved, while not requiring users' time and attention for conflict resolutions that can be automatically done.

Merging allows multiple versions to be joined together, producing a new version representing the union of the actions taken from the previous versions (Neuwirth et al., 1992). However, merging multiple versions is not trivial if conflicts exist among versions. There are three kinds of potential conflicts. If the objects to be merged are different, such as two different words occupying the same position in the same sentence in two versions, an *object conflict* exists. If incompatible operations have been performed on the object, such as replacement and deletion of the same word, an *operation conflict* exists. If the granularities of objects that have been changed are different, such as a word replaced in one version while the sentence containing that word replaced in another, a *granularity conflict* exists. In traditional version control systems, such as RCS (Tichy, 1985), ClearCase (Chang et al., 1998), and MICROCOSM (Fountain et al., 1990; Hall et al., 1992), detection and resolution of such conflicts are not necessary since these systems use an exclusive lock, and merging is accomplished by simply overwriting the previous version.

Traditional version control systems do not allow multiple versions to be created or revised in parallel (Horwitz et al., 1989). This is, however, a highly desirable capability in CSCWriting systems. To resolve conflicts in a collaborative writing environment, there needs to be a way for users to state rules that the system can apply to identify different types of conflict, prioritize conflicting edit operations, and thus decide what the result of merging conflicting operations will be. Concurrent Versions System

(CVS) (Coleman, 1997) takes the semi-automatic approach of automatically merging edit operations in different versions if these do not conflict with each other, but requiring users to manually resolve any conflicts detected. In a collaborative writing environment users may not want to be tied down to one set of merge rules. Instead, they may want to apply different conflict resolution rules for merging during different phases of the collaboration. Thus, a merge mechanism for CSCWriting should provide users with a flexible way of changing these rules interactively during the course of collaboration. In addition, it should be possible to handle various levels of granularity during merging (Neuwirth et al., 1992).

### **2.3 Access control**

Ellis defined access control as "... a mechanism to determine who has the right to access certain information and in what manner..." (Ellis et al., 1990). Thus, access control has to be more sophisticated than merely enforcing read, write and execute permissions on the document file itself. First, access control of components of the document at different levels of granularity, such as characters, words, sentences or paragraphs, is desirable since different members of the group may be in charge of creating these components and may wish to prevent unauthorized (if some parts of the document need to be kept private from some of the group members) or accidental access by other members. Second, access privileges need to be decided based on the role of the group member in the authoring project. Individuals in collaborative writing environments tend to collaborate within the parameters of assigned roles such as writer, editor, reviewer or commentator. For instance, one member may be the leader who has full access to all components of the document, while others may have write access to

only those parts they are responsible for writing. Such *role based access control* should apply to elements with different granularities inside the document versions, and allow users with the appropriate access privileges to interactively specify changes to their access rights at different levels of granularity (or those of others). Third, these access privileges should not have to be decided at the beginning of the project and then remain unchanged during the life of the authoring project. Instead, access privileges need to be dynamic and easily changeable during the course of the project, with restrictions on who is authorized to alter the privileges. Most traditional version control systems for CSCW fail to provide access control mechanisms with these features. For example, the RCS and Liveware systems (Witten et al., 1995) are built on the UNIX file system, and only control user access to the versioned files, not to individual objects inside the files. Users are given the same access rights to all objects in any version.

## **2.4 Version management**

Version management functions that are desirable for CSCWriting include maintaining multiple versions and dependencies among them, trimming versions, allowing users to easily visualize the evolving version structure of the document, and query individual versions for more information. In modern day collaborative projects with members distributed across the globe in different time zones, it is quite likely that collaborative activity will switch between synchronous and asynchronous modes, and members may disconnect and reconnect to the network often. Therefore, the system must keep track of times when a single common version is being worked on by multiple group members simultaneously (synchronous collaboration), when group members are working individually on their own local versions (asynchronous collaboration), and

when both cases (some members collaborating synchronously while others are working individually) are occurring.

In the first case, one version of the document is the current (and changing) one. The system needs to track and resolve multiple edits by different members on the same document component. This should occur regardless of whether edits occur simultaneously or sequentially in a collaborative session, since network latencies may make simultaneous changes appear sequential. So edit operations cannot be committed in the order they are seen by the version management system. In the second case, there will be multiple current versions and the system needs to know of their existence and dependencies among these. The third case is when there are multiple current versions, with some being locally changed by individual users while others being changed by multiple users synchronously.

When different versions of a document proliferate, the group may want to trim versions which are not of interest any more by creating a common version agreed upon by the entire group. This common version then forms the parent of subsequent versions. This requires differencing and merging functions. Proliferation of the number of versions wastes memory, and makes search for particular document elements difficult (Dix and Beale, 1996). The removal of any version, however, should not affect the consistency of the entire version structure. A “savepoint” mechanism, in which a common version is created and saved at intermediate points of the document’s evolution, can be adopted in deciding which version can be deleted safely (Hicks and Haake, 1995). Most existing version control systems do not support such a savepoint mechanism, and thus do not allow users to remove intermediate versions from the version set.

Once the document acquires a rich history, it is useful for the group members to be able to visualize its version structure. Besides providing a context, a sense of how the current version(s) of the document came to be from the very first version, this also allows users to revert to earlier versions if necessary. Users should be able to query this visualization for more information (e.g., Who created a particular version? What are the differences between an earlier and a current version? How does the document structure differ between an earlier and a current version?).

Figure 1 shows an illustrative version structure resulting from four users working on a document. First users 1 and 2 together created version  $V_{12}$  and users 3 and 4 created version  $V_{34}$  in parallel. Then user 1 and user 2 checked out local copies of their common version, created new individual versions  $V_1$  and  $V_2$ , and later came on-line to merge the individual versions into  $V_{12}$ . Meanwhile users 3 and 4 edited individual copies of their common version to create two new local versions each. Finally the common version between users 1 and 2, and the latest individual versions of users 3 and 4, were merged to produce the final version of the document.

## **2.5 Group awareness**

Group awareness mechanisms allow a user to be informed of the work done by others, the synchronous on-line presence of other group members, the creation and availability of new document elements by other users, and changes made to document elements by others. Maintaining awareness is a critical factor in the smoothness and success of collaborative efforts (Edwards and Mynatt, 1997). In CSCWriting, this includes notification of who has checked out which version for edits (thus making it inaccessible to others), who has made any particular change (e.g., a delete) to a

particular document component (e.g., a sentence) in a version, etc. Version structure visualization can provide an effective platform to convey some notifications while other ways of maintaining awareness can be achieved by mechanisms such as color coding within the document edit window (Salcedo et al., 1997).

Timewarp (Edwards and Mynatt, 1997) explored two kinds of awareness: local awareness and global awareness. Local awareness provides a view of the specific path along the current version, while global awareness allows users to see the big picture of a version by displaying all parallel branches of a version. However, Timewarp lacks expressiveness of user activity since only the final states of user activities are recorded in the versions. It does not provide intermediate information on how a user arrived at an operation to be applied to a document element. For instance, if a user changed the word “containing” to “comprising”, and then to “consisting of”, only the information that “containing” was replaced with “consisting of” will be maintained. This problem is also found in Lotus notes (Lotus Notes, 1995) because intermediate versions in Notes are always overwritten by a final version during the server's synchronization process. Systems such as RCS (Tichy, 1985) and CVS (Coleman, 1997) are more expressive. These record all editing operations on each version, but do not store contextual information such as the time at which an operation occurred or the role of the user initiating the operation. This prevents group members from acquiring a complete and contextual view of the evolution history of a document object, such as how a certain sentence evolved in structure and time, and who, acting in which roles, made these changes.

### **3. Design and Implementation of A CSCWriting System**

In order to address the limitation that none of the existing systems provide the full palette of functionalities needed in a tool that supports CSCWriting, we designed and implemented a prototype tool. Its capabilities include differencing, merging, role-based access control, version management, version structure visualization, and group awareness. It also contains a text editor with basic text creation and editing facilities. Our focus in this endeavor was on functionalities needed to effectively support document management in CSCWriting, not word processing. So we did not attempt to build yet another full function text editor. The system is implemented using Solaris 2.5, TCP/UDP/IP protocols, OSF's Motif toolkit set, X-library drawing primitives, X imaging Library, and C++. Figure 2 shows the architecture of the system. Its components are described in the following sub-sections.

#### **3.1 The AID tag**

The most important contribution of our work is the proposal of an Activity IDentification (AID) tag as the fundamental mechanism to support distributed management of multiple versions of a document. The AID tag facilitates design and implementation of an integrated approach that includes differencing, merging and role-based access control at different levels of granularity, version structure maintenance and visualization, and group awareness of document status and operations. The AID tag leads to simple and effective differencing and merging schemes. Its unique address scheme eliminates the need for large storage capacity for version maintenance. Role-based access control can be implemented by associating access right tables and role assignment capabilities with the AID tag. Information for providing group awareness of

the changing document is available from the AID tag. In addition, since the system maintains a user-browsable version structure of the evolving document that incorporates AID tag information, any user can easily understand the historical evolution and current context of the document.

The AID tag consists of three parts: granularity level, element address, and type of operation performed. The granularity level represents objects such as a word, a sentence, or a paragraph. The element address, which is used to locate the exact position of an element in any version of the document, is a unique real number starting at 1.0. As a user creates the initial version of the document, elements are assigned with successive real numbers, e.g., 1.0, 2.0, 3.0. This initial address assignment is performed at each granularity level, e.g., word, sentence, and paragraph. Given this initial address assignment for each element, the address of a newly inserted element is determined by computing the middle point between the addresses of the previous and the next elements at the same level of granularity. For example, if a new line is inserted between lines with addresses 4.0 and 5.0, a new address, 4.5, is assigned to the inserted line. The types of operation defined in AID are ‘create’, ‘insert’, ‘delete’, and ‘replace’. Other edit operations such as ‘cut’ and ‘copy’ can easily be included in AID as needed.

The formal representation of the AID tag is as the Kleene closure of a triple:  $\{ \langle (P|S|W), \text{address}, (C|I|D|R) \rangle \}^*$  where P: Paragraph, S: Sentence, W: Word, C: Create, I: Insert, D: Delete, and R: Replace. An AID tag is inserted at the front of an element at the first time it is edited. It is updated every time the element is edited afterwards. For example, the AID tag  $\langle P, 1.0, C \rangle \langle S, 6.7, R \rangle \langle W, 3.0, R \rangle$  means that the third word in the sentence at address 6.7 in paragraph 1.0 has been edited with the ‘replace’ operation. The sentence 6.7 has also been edited with a ‘replace’ operation, and the paragraph 1.0

has been created with no edit operation performed on it afterwards (at the paragraph granularity level). If the ‘create’ operation is found in a tag, it means that the corresponding element has never been edited since the initial creation.

This new addressing scheme makes differencing simple and straightforward because of its global, stable, and hierarchical characteristics.

- 1) It is global since a particular word, sentence, or paragraph from one version can be easily located and referenced at the *same* address in other versions.
- 2) It is stable since addresses of new elements never change the existing address space. For example, a ‘delete’ operation does not require decrementing addresses of all following elements. Instead, it simply modifies the type of operation field in the AID tag.
- 3) It is hierarchical since the addressing scheme conforms to the hierarchical structure of text: paragraph  $\Rightarrow$  sentence  $\Rightarrow$  word. For example, a word at address 3.0 means that the word is located at the third position with respect to the beginning of the sentence to which it belongs.

These characteristics of the address space save CPU time and memory by eliminating the need for a large database to maintain editing histories. Although a system such as AUGMENT (Engelbat, 1988) uses a similar address scheme for maintaining editing histories, its address space is user assigned integers (i.e., not unique) and relative to the position of other elements. Thus, locating a particular element across versions is time consuming. In our approach, an element will have the same address in all versions.

### **3.2 Role-based conflict resolution while merging**

Merge rules provide a mechanism for resolving conflicts during merging, by

explicitly specifying what action to take when conflicting edits on the same object appear in the versions being merged. Most merge tools lack a flexible and interactive facility for group members to specify, and dynamically update these rules as collaboration progresses. To allow such interactive specification and dynamic updating, we use a variant of the merge matrix proposed in (Munson and Dewan, 1994). It is a square matrix with rows and columns for all editing operations that can be performed on an object. The rows represent the editing operations of one user and the columns represent the operations of another user. Each entry in the matrix specifies the action that will take place when editing operations corresponding to the row and column, executed on the object by the respective users, need to be resolved. This merge matrix is defined for objects at each granularity level. Table 1 shows a sample merge matrix. In the table, 'Both Operations' indicates that both editing operations will be committed in the merged version. In this case there is no conflict to be resolved. 'Conflict' indicates that operations performed by both users are not compatible. These entries will result in merge rules being applied which will select one of the two operations to commit in the merged version, or users being asked to manually resolve the conflict.

The merge matrix as proposed by Munson and Dewan, however, has a fundamental problem in an n-way merging, i.e., merging n versions. A merge matrix for n users and m operations would require  $m^n$  entries, which is time consuming to fill in when the collaborating groups have three or more users. To remedy this problem, we propose the use of a *role-based merge table* in which each dimension represents an authoring role rather than an individual group member. Since group members always assume a certain role in normal collaborative writing activity, assigning roles in a merge matrix is reasonable and natural. The roles can be editor, commentator, writer, etc. This

approach has several advantages.

Since roles are assigned with predefined responsibilities and the number of roles normally remains a small constant, n-way merge is not necessary. Instead, n versions (created by n members assuming one of k roles) can be carried out through a series of n-1 2-way merges. An n-way merge requires that all n versions be available simultaneously. If merging is accomplished by 2-way merges, these can be carried out asynchronously. Two versions can be merged as soon as these are available from the users regardless of the status of the other n-2 versions. It facilitates distributed computation as one computer is not merging n versions. Instead, the 2-way merges can be carried out in parallel at different machines.

This approach reduces both the space needed for storage of merge tables, and, more importantly, allows easier interactive specification and revision of merge policies by group members. The number of m×m (if there are m editing operations allowed) merge tables needed is given by the following equation.

$$\binom{k}{2} + k = \{k + \{k! / [2! (k-2)!]\}\} = (k^2 + k)/2 \text{ where } k \text{ is the number of roles.}$$

This is because we need k tables for resolving conflicts between changes made by pairs of individuals assuming the same role and  $\{k! / [2! (k-2)!]\}$  tables to resolve conflicts between pairs of individuals assuming two different roles from among the possible k roles. The total number of entries then is  $m^2(k^2 + k)/2$  which is considerably less than  $m^n$ .

A third advantage is that flexibility and fluidity of the collaborative authoring process is supported. Individual group members may assume multiple roles (e.g., writer of one section and critic of another) as well as change their roles as collaborative

writing progresses.

Figure 3 illustrates a snapshot of the interface for interactively revising a merge table in the prototype system we implemented. It shows a step in the merging of 6 versions of a document, with the merge table for resolving editor-editor conflicts. The left part of the figure shows that if one editor deletes an object while the other replaces the same object, a conflict arises: should the object be deleted in the merged version or replaced? The right part of the figure shows that the users proposed a conflict resolution rule by changing the two 'C' entries in the table to 2 indicating that in both cases the editing operation of Editor 2 should take precedence. Each button in the merge table can be assigned with the role that has higher precedence (Editor 2 in this case) or with a 'C' indicating that the conflict needs to be resolved manually. In such a case, the system will retain the conflict in the merged version by placing the following information in front of the AID tag: [CF// Version\_name:AID\_tag | Version\_name:AID\_tag] where CF stands for conflict. For example, suppose an editor deletes a word at position 3.25 in version '*dissert.001*' and another editor replaces the same word at the same position in version '*dissert.002*'. If the merge table does not specify which operation takes precedence, the following information is attached in the AID tag of the word in the merged version: [CF// dissert.001:<W,3.25,D,> | dissert.002:<W,3.25,R>]. This allows the system to automatically search for conflicts and the users to then resolve them at the time of merge.

The proposed merge table also allows users to merge versions based on multiple granularity levels. A merge between two versions can be performed at word, sentence, or paragraph level. When two versions are merged with a smaller granularity, e.g., word level, the result should be propagated to the upper level. That is, the AID

tags of sentence and paragraph should also be updated.

### **3.3 Access control**

Access control in the implemented system provides four different roles, i.e., editor, writer, reviewer and creator. An access rights table (Table 2) defines these roles in terms of the execution rights of editing operations associated with each role on objects of any granularity in the document. Table 2 shows that creator has been configured with maximum access rights covering all the editing operations, i.e., insert, delete, replace, and comment. Editor is not allowed to create, and reviewer can only do the commenting operations. The collaborative group leader can specify or change this information anytime during the course of collaboration.

This role specification mechanism allows incorporating access rights into merge policy. To facilitate this capability, the AID tag is extended by adding role information, e.g., {<(P|S|W), address, (C|I|D|R), (T|R|A)>}<sup>\*</sup> where T: Time of creation, R: Role, and A: Author. For example, <S, 3.0, I, 1999:03:09:13:45:33, Writer, Tom> denotes that the third sentence is inserted by Tom as a writer, at 13:45:33, March 9<sup>th</sup>, 1999.

### **3.4 The text editor**

The text editor, in addition to facilitating document manipulation and version creation by users, automatically generates AID tag information. Thus, it automatically generates and maintains address and other information necessary for granularity control, access control, and merging. As a user opens an existing version of a document, the text editor determines which objects (words, sentences, and paragraphs) in that version can be accessed and edited by the current user, and which editing operations are

permitted. This processing is done automatically without user's explicit keyboard activity. When the user makes a change to an object, the text editor propagates information about this editing operation to higher levels of granularity, if necessary, to maintain AID tag consistency through the levels.

Figure 4 shows a snapshot of the text editor. The text editor allows users to change the granularity level of editing at any time through a set of radio buttons at the top of the editor window. The current granularity level is clearly indicated by the automatic selection and underlining of the entire object at the current granularity level, at the cursor's location. The text editor also provides radio buttons for role selection so that a user can change his or her role during collaboration. If more centralized control is desired, a facility by which these buttons can be selectively disabled for individual group members by the group leader can be easily implemented, but this was not done in the prototype. The role selection buttons are also provided at the top of the text editor window for easy access.

### **3.5 Version management editor**

The version management editor provides group members with a way to visualize and navigate the version structure of the evolving document. Each node in the version structure display in this editor represents one version. The leftmost root node indicates the original version, and its child nodes represent versions derived from the original. Nodes in this version structure can be modified, in edit mode, through merging, insertion or deletion. Merging creates a new version from two existing ones. Insertion creates a new child of an existing version. Deletion removes a version. In read-only mode, users are only allowed to checkout, browse a version, or navigate the

version structure. Checking out a version allows a user to create a local copy of the version.

Figures 5a and 5b show interface pictures of the version management editor. Figure 5a indicates that merging the nodes ‘dissert.005’ and ‘dissert.006’ created the node (version) ‘dissert.007’. When the cursor is over a node, a node pop-up menu appears. This menu contains *checkout* and *node query* option items. If the user selects the checkout option from this menu, the editor displays information about the current node (shown in the figure for the node ‘dissert.007’) such as the creator, creation time, users who have already checked that version out, and conflicts, if any, that were found if this version was the result of a merge. Then, the version management editor opens a text editor window and displays the contents of the node.

The version management editor also allows users to perform various search operations on the version structure by selecting the node query item from the pop-up menu. The resulting search window that the editor opens up is shown in Figure 5b. It provides *New*, *And*, *Or*, and *Not* query types. Queries can be issued on attributes, such as the node name, file name, owner and keywords, of a node. After selecting the query type, the user can select a node attribute from *Attributes for Query* button. The *Attributes* column in the lower left part of the window displays selected attributes. This indicates the attributes upon which the search will be based. The user can choose one or more attributes. The user also can customize the query by selecting node attributes from the *Display For* box. After the search is executed, the editor redisplay the version structure showing the query results. This query tool provides users with a way to search for versions with particular attributes. Thus, it is a tool that can help improve group awareness of the document’s evolution.

The version management editor also facilitates *savepoint* clearance. A savepoint is an existing node (version) that the group members agree to be a current common version, from which future individual versions will be spawned. It is typically (but not necessarily) a version created by merging all or several of the existing versions. It forms the root node of a new version tree. When a group of users selects a node (version) as a savepoint, the editor first checks whether any preceding nodes (ancestors of the savepoint) have been checked out by users who have not finished their revisions on their local copies (i.e. not checked them back in yet). If no nodes with revisions in progress exist, the ancestor nodes of the savepoint are removed.

#### **4. A Case Study**

To further illustrate the benefits of our approach and implemented prototype for supporting CSCWriting, we provide an example with several illustrations in this section. The example shows creating a collaborative authoring group, members of which write, edit, and merge a simple text document.

The first step is to create a new collaborating group. Figure 6 shows the state after the version control system has been started and a new document called ‘dissert.doc’ has been created. Then this initial node has been checked out by two members who are located at different sites, with each of their local versions named ‘dissert.002’ and ‘dissert.003’. Figure 7 shows the text editor window with ‘dissert.doc’ loaded. The AID tags in the document remain hidden.

The next step is to show how edit operations can be executed on a newly created version. Specifically, user A, an editor, wants to insert a new word ‘really’ between the word ‘truly’ and ‘based’ in the fourth sentence of the version ‘dissert.002’.

The user first opens this version by clicking the left mouse button on the node 'dissert.002'. Note that the version management editor window can be viewed on any of the user's computers. A text editor window opens with the contents of 'dissert.002'. To insert the word, user A needs to set the granularity level to 'word' and types in the word (Figure 8). After finishing the editing, user A checks in this edited version. The checking in automatically creates a new version named 'dissert.004'.

Meanwhile, user B wants to delete the fourth, fifth, and sixth sentences from her version, 'dissert.003'. To delete those sentences, she changes the granularity level to sentence, selects the sentences, deletes them, but does not check in the document. Therefore the version structure is not updated to include a new version. The deleted sentences do not disappear, but turn gray indicating deletion. Figure 9 shows the text editor after the deletion.

User B then wants to merge her version with user A's version 'dissert.004', and finds that the two versions are in conflict. However, the merge table applicable in this case, shown in Figure 10, indicates that the deletion operation of user B (editor 2) takes precedence over the insertion operation of user A (editor 1). Thus, the merged version retains user B's editing operation, i. e. the deletion of three sentences. Figure 11 shows the version management editor after merging the two nodes, 'dissert.003' and 'dissert.004' into a new node (version) called 'dissert.005'. Figure 12 shows the contents of the merged node along with the AID tags. The tags clearly indicate that the fourth, fifth, and sixth sentences have been deleted.

Now suppose another group member, user C, checks out the original version 'dissert.doc' and finds that the fourth, fifth, and sixth sentences have been removed by user B from the new merged version. User C, however, wants to keep the fourth

sentence. When she tries to merge her version ('dissert.006') with the latest version (dissert.005), the system generates a conflict message (this can be seen in Figure 5a). The merge table (Figure 3) shows that conflicts exist between editor 1 (user B) and editor 2 (user C) on 'replace' and 'delete' operations. User C then sends email to user B to resolve the conflict. Reading user C's message, user B decides to follow user C's suggestion on saving the fourth sentence, and changes the merge table entries to give precedence to user C. Now user C can successfully merge 'dissert.005' and 'dissert.006', creating 'dissert.007' (see Figure 5a). Figure 13 shows the text contents of this merged node. It indicates that the fourth sentence, which is in black color, has been saved from deletion while the fifth and sixth sentences have been deleted. This illustrates how our approach helps resolve edit conflicts with user intervention.

## **5. A Comparative Analysis**

In this section we compare the AID scheme with semi-automatic merging and command history maintenance approaches, exemplified by systems such as Multiple Source Control (Dix and Beale, 1996) and Concurrent Versions System (Coleman, 1997).

The AID scheme requires less number of operations for the insertion and deletion of a word when compared to approaches in which the addressing scheme is not stable, i.e., if an insertion or deletion causes the addresses of all following words to change. Let us assume that the number of words in a document is  $N_w$ , and that the generation of a new address requires only a constant time operation. In the command history maintenance scheme, the rest of the address space in a document needs to be updated as a new word is inserted or deleted. If a new word is inserted at the beginning

of the document, the addresses of  $N_w-1$  words should be updated. If the word is inserted at the second word location in the document,  $N_w-2$  word addresses should be updated, and so on. If we assume that a word will be inserted or deleted from any of the  $N_w$  positions with equal probability, the expected number of address updates needed for one insert or delete operation is  $1/N_w (1 + 2 + 3 + \dots + (N_w - 1)) = (N_w - 1)/2 = O(N_w)$ . The stability of the address space used by the AID tag approach does not require the changing of any address in the document as a result of an insertion. So the only operation needed is the generation of a new address that is the midpoint between the addresses of the previous and the next words. This is an  $O(1)$  – constant time – operation.

Unlike insertion and deletion operations, the replace operation does not require the update of other addresses in all three approaches being compared. Therefore, replace is an  $O(1)$  operation in all these approaches.

The number of operations needed to search for an element in a version is different for different schemes. In a simple text based differencing and merging scheme, since editing operations can change the address of a word arbitrarily, the word's address in the original version is of no help in finding it in a later version. So, in the worst case, the system has to look at all words in a version to locate a particular word in that version. This operation is  $O(N_w)$ , requiring  $N_w$  steps in the worst case. In the command history mechanism, a particular word from a version can be located in a subsequent version by following links in an address change set that this mechanism maintains (Dix and Beale, 1996). If the maximum number of elements in this set is  $S_w$  and the average number of link is  $L$ , the number of operations for searching an element is bounded by  $O(L*S_w)$ . If the total number of elements in a document is  $N$ , and  $S_w$

tends toward  $N$ , the average number of searching operations is bounded by  $O(L*N)$ . In the AID tag scheme, since the address of a word is stable across versions, a particular word can be found at the same address in any version. So, if the address of the word is known and an address-based storage scheme is used for objects in the document, the word can be found in any version in  $O(1)$  time.

Let us assume that the maximum number of edit operations (insert, delete or replace) performed on any word in the document is  $N$ , and that  $M$  bytes are needed to store each operation script and address. In the command history approach that maintains a history of all edit operations, if the maximum number of edited words is  $N_w$ ,  $O(N * N_w * M)$  space is required to store the operation scripts. This memory requirement increases as  $N$  increases. In the AID tag scheme, the insertion operation generates a new address tag. Replace and delete operations change the content, but do not add to the size, of this tag. The AID tag represents only the final editing action performed on a word, not the entire history of editing actions performed on it. This is because the AID tag addressing scheme eliminates the need to keep track of command history. Therefore, the space requirement is  $O(N_w * M)$ , which is independent of the number of edit operations performed on any word. This advantage becomes more significant as the number of edit operations and the number of versions generated increase during the life cycle of a collaboratively authored document.

Thus, the AID scheme facilitates efficient cross referencing of document objects at any level of granularity across different versions, makes locating and tracing the evolution of an object through multiple versions easier, and eliminates the burden of recording and maintaining long sequences of update information. These advantages are achieved by extending the address space *constructively* as new objects are inserted, and

by using this stable address space as the basis for differencing and merging. In other words, these advantages stem from the unique addressing scheme our approach employs, which is both global and stable (as explained in Section 3.1).

## **6. Future Work**

Aims of future research include extending the current capabilities of the version management system and improving the space and time efficiency of storing and manipulating AID tags by addressing one limitation of the current approach. This is that the AID tags considerably increase the size of a document. For example, the document used in the illustrative example of Section 5 required 13,285 bytes of storage as plain text. Its size increased to 32,067 bytes when AID tags were included. Although such increase in size does not appear to degrade overall editing performance of the text editor, it can affect the speed of reading and writing tag information from and to a disk file. If additional role information is added to the AID tag as a result of adding additional levels of granularity, it will increase storage requirements even more. One way to address this is to use, instead of characters as in the current prototype, an octal numbering scheme (similar to how the Unix file system specifies read, write and execute permissions) for specifying access rights, roles and granularity information. Defining more roles and granularities will not increase the storage space significantly under this modification.

The AID address generation mechanism also contributes to increased storage demands and reduced speed of reading and writing tag information. Since a new address is computed as the real number (the preceding address + the following address)/2, each new address increases in size by one digit. This situation can be

improved by defining smaller intervals between two existing addresses for creating new addresses. For instance, if the interval between two consecutive addresses is divided into 10 sub-intervals instead of the present 2, the size of a new address will increase by one digit only once every 10 consecutive insertions. For example, if we insert a new word between the 7<sup>th</sup> and 8<sup>th</sup> words, the new word is assigned address 7.1. The next inserted word, if it follows the previous insertion, gets the address 7.2, and so on. The next level of addresses will start at 7.\*1 where \* is 1...9.

This scheme has one drawback. It relies on the assumption that when text objects such as words or sentences are inserted consecutively in the forward direction. The worst case can occur, which is when objects are inserted backwards, i.e. every new insertion is placed *before* the previous insertion. For example, a new word inserted between words 7 and 8 will get the address 7.1. The next inserted word is placed before the word at 7.1, i.e., between words 7 and 7.1. So it is assigned the address 7.11. If the next word is inserted before this word, it will be given 7.111 as its address, and so on.

A minor modification of this scheme will take care of this problem. Divide the interval between any two consecutive addresses into eight sub-intervals. Then assign a new word an address that is the midpoint of the interval between the addresses of the preceding and following words. The available addresses between 7 and 8 are then 7.1, 7.2, 7.3, 7.4, 7.5, 7.6 and 7.7. If a new word is inserted between words at positions 7 and 8, it will be assigned the address 7.4. The next insertion, if it is placed after the word at 7.4, will get the address 7.6. If it is placed before the word with address 7.4, it will be assigned the address 7.2. This way, the size of an address will increase by one digit only after at least 3 insertions. This scheme can be generalized to defining N sub-

intervals between any two consecutive addresses, where  $N$  is some power of 2. Then the address size will increase only after at least  $\log N$  insertions. This is another modification planned for the system in future.

Users can choose to view and change document objects at different levels of granularity. This use of granularity can help improve the degree of group awareness by facilitating various levels of notification. Implementing a notification mechanism that can alert group members of changes occurring at the word, sentence or paragraph level in a particular version is an easy future extension of the current system. For instance, if a user is editing a paragraph in a version, it can be grayed out in all other text editor windows displaying the same version. Other planned extensions include providing more control to group members in adding or deleting new roles with associated access controls, and additional granularity levels such as chapters and sections.

The AID tags facilitate studying the behaviors of group members in collaborative authoring situations. Information encapsulated by these tags, together with the version management editor and the query tool it incorporates, provide an excellent mechanism for tracing the history of evolution of a collaboratively authored document. This enables not only better group awareness in asynchronous collaborative modes typical in mobile environments, but also post-hoc analyses of the contributions of members to the collective document.

## **7. Conclusion**

In this paper we introduced the AID tag, a new mechanism for distributed version management in computer-supported collaborative authoring of documents. It makes differencing and merging document versions straightforward. Its unique address

scheme eliminates the requirement for large storage capacity in order for consistency maintenance across versions. Access control is improved by determining access privileges based on roles, providing role-based merge tables, and allowing dynamic assignment of roles to group members. It allows a user to assume and change roles dynamically during the course of collaboration. The corresponding version control system provides access at various levels of granularity. Users can easily define and modify these access rights. All of these enhancements contribute to better group awareness of members' roles and actions.

In addition, since the AID information is incorporated inside each version, any user can easily understand the work context (history of the document's evolution) by simply browsing and querying the document version structure in the version management editor. Group members can also adopt various roles and change their roles during collaboration. With each assumed role, the activity of a group member is regulated with defined responsibilities and access rights. The group can specify and modify role-based rules (embodied in the merge tables) by which conflicts are resolved, or choose to resolve conflicts manually, during the merging of multiple versions. This reduces the chance of unexpected conflicts, provides a clear mechanism to resolve conflicts when they occur, and enhances group coordination and awareness. In summary, the approach to CSCWriting described in this paper lends much more flexibility to the collaborative authoring process than was hitherto possible.

## References

- Chang, K., Murphy, L., Fouss, J., Dollar, T., Lee, B., and Chang, Y., 1998. Software Development and Integration in Computer Supported Cooperative Work, *Software-Practice and Experience*, 28(6), 65--679.
- Coleman, D., 1997. *Groupware: Collaborative Strategies for Corporate LANS and Intranets*, Prentice Hall PTR.
- Decouchant, D., Quint, V., and Salcedo, M., 1996. Structured and Distributed Cooperative Editing in a Large Scale Network, In Roy Rada (Ed.), *Groupware and Authoring*, Academic Press, pp. 265--295.
- Dewan P. and Munson, J., 1995. The Role of Version Control in CSCW Application: A Position Statement, *Proceedings of the Workshop on the Role of Version Control in CSCW Application, European Computer Supported Collaborative Work Conference*.
- Dix, A. and Beale, R., 1996. Information Requirements of Distributed Workers, In Alan Dix and Russell Beale (Eds.), *Remote Cooperation*, Springer-Verlag, pp. 113--143.
- Edwards, W. and Mynatt, E., 1997. Timewarp: Techniques for Autonomous Collaboration, *Proceedings ACM Conference on Human Factors in Computer Systems (CHI'97)*, pp. 218--225.
- Ellis, C, Gibbs, S., and Rein, G., 1990. Design and Use of a Group Editor, In G. Cockton (Ed.), *Engineering for Human Computer Interaction*, pp. 13--25.
- Engelbat, D., 1988. Authorship Provisions in AUGMENT, In Irene Greif (Ed.), *Computer Supported Cooperative Work: A Book of Readings*, Morgan Kaufmann Publishers, pp. 10--126.

- Fountain, A., Hall, W., Heath, I., and Davis, H., 1990. Microcosm: An Open Model for Hypermedia with Dynamic Linking, *Proceedings of the European Conference on Hypertext*, Cambridge University Press.
- Hall, W., Heath, I., Hill, G., Davis, H., and Wilkins, R., 1992. Microcosm: State of the Art, Computer Science Technical Report CSTR 92-18, University of Southampton, Southampton, England.
- Hicks, D. and Haake, A., 1995. Workshop Summary, *Proceedings of the Workshop on the Role of Version Control in CSCW Applications*, pp. 7--16.
- Horwitz, S., Prins, J., and Reps, T., 1989. Integrating Non-interfering Versions of Programs, *ACM Transactions on Programming Languages and Systems*, 11(3), pp. 345--387.
- Lotus Notes, 1995. Lotus Notes Groupware - Communication, Collaboration, Coordination, Executive Summary on Lotus Notes, Lotus Development Corp., <http://www.lotus.com/bible>.
- Munson, J. and Dewan, P., 1994. A Flexible Object Merging Framework, *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'94)*, pp. 230--242.
- Neuwirth, C., Chandhok, R., Kaufer, D., Erion, P., Morris, J., and Miller, D., 1992. Flexible Diff-ing in a Collaborative Writing System, *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'92)*, pp. 147--154.
- Salcedo, M. and Decouchant, D., 1997. Structured Cooperative Authoring for the World Wide Web, *Computer Supported Cooperative Work: The Journal of Collaborative Computing*, Kluwer Academic Publishers, 6(2-3), pp. 157--174.

- Tichy, W., 1985. RCS - A System for Version Control, *Software-Practice and Experience*, 17(7), pp. 637--654.
- Witten, I., Thimbly, H., Coulouris, G., and Greenberg, S., 1995. Liveware: A New Approach to Sharing Data in Social Networks, *International Journal Man-Machine Studies*, Vol. 34, pp. 337--348.
- Zhang, K., Wang, J., and Shasha, D., 1996. On the Editing Distance between Undirected Acyclic Graphs, *International Journal of Foundations of Computer Science, Special Issue on Computational Biology*, 7(1), pp. 4--57.

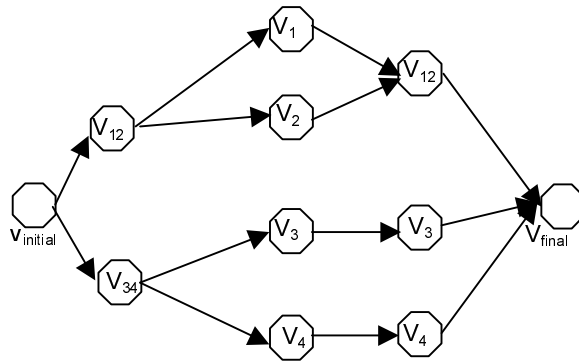


Figure 1. A version structure sample

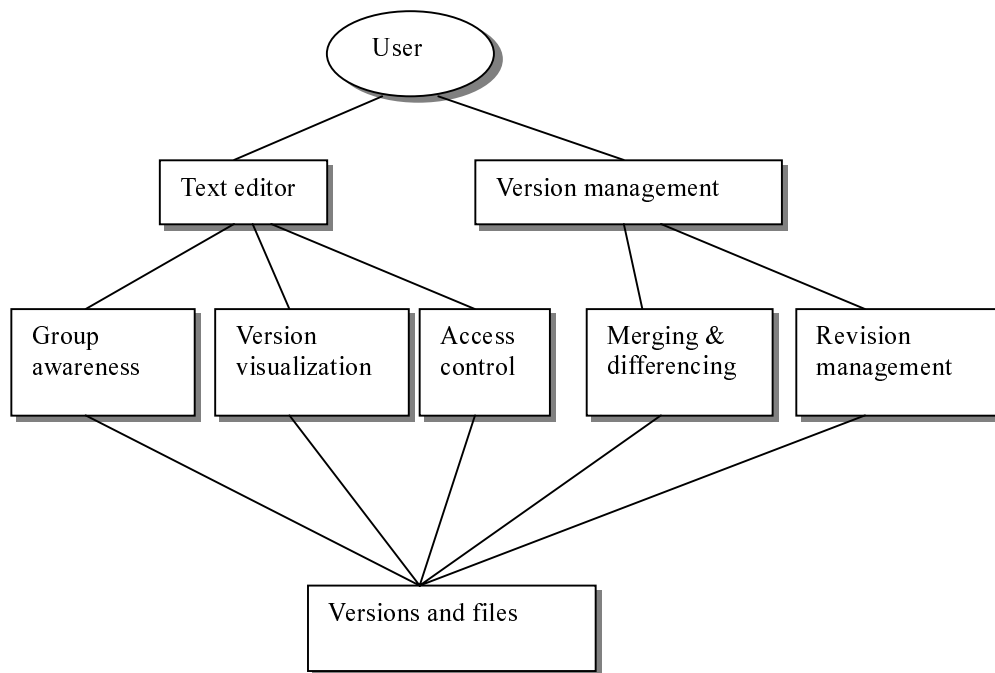


Figure 2. Architecture of the CSCWriting system

	User A			
User B		Insert	Delete	Replace
	Insert	Both Operations	Both Operations	Both Operations
	Delete	Both Operations	Both Operations	Conflict
	Replace	Both Operations	Conflict	Conflict

Table 1 A merge matrix

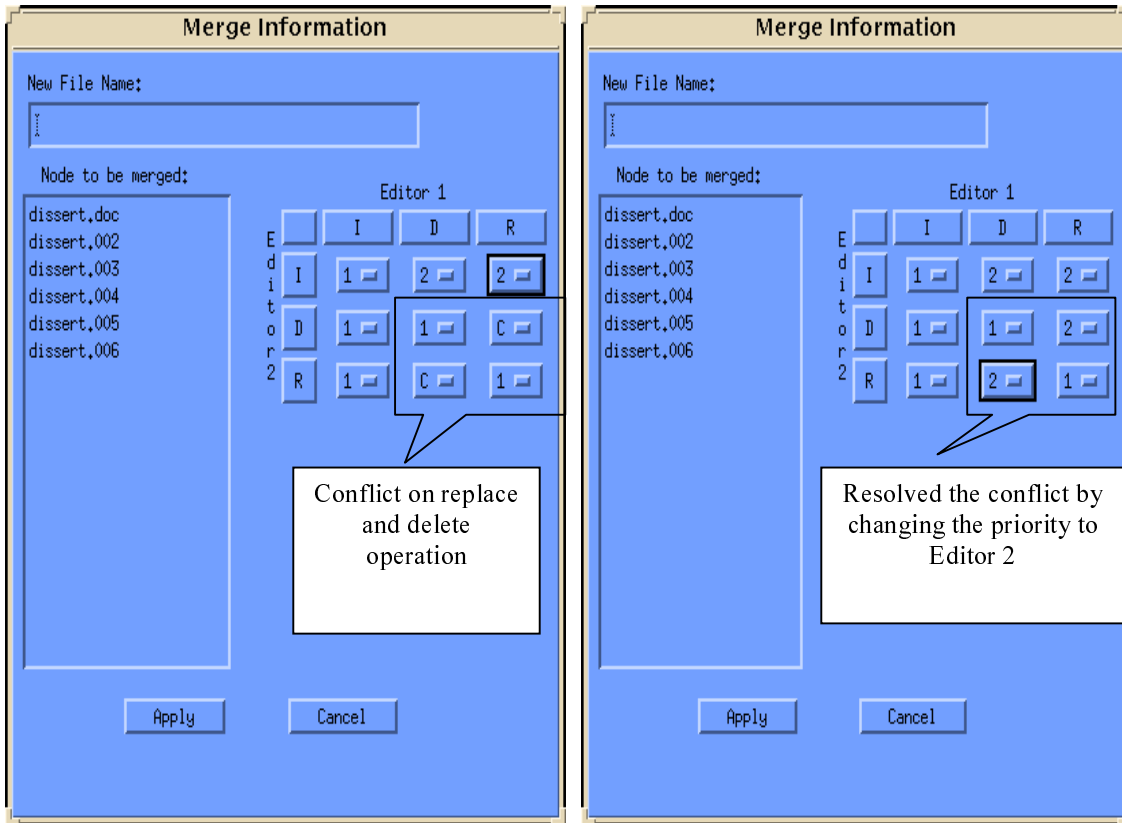


Figure 3. Merge table interface

	Create	Insert	Delete	Replace	Comment
Editor	No	Yes	Yes	Yes	Yes
Writer	Yes	Yes	Yes	Yes	No
Reviewer	No	No	No	No	Yes
Creator	Yes	Yes	Yes	Yes	Yes

Table 2 Access rights table

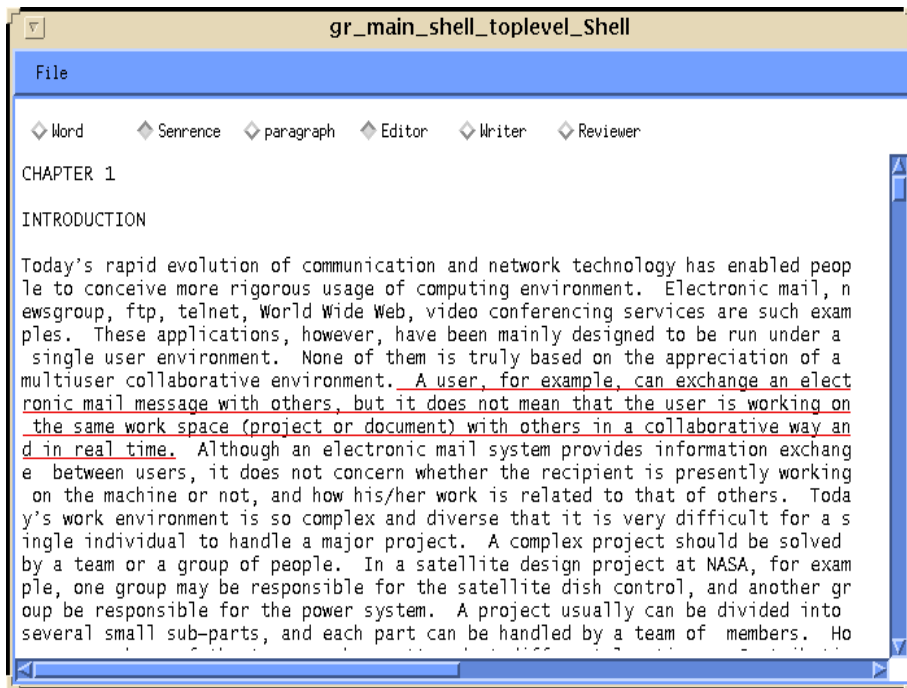


Figure 4. The text editor

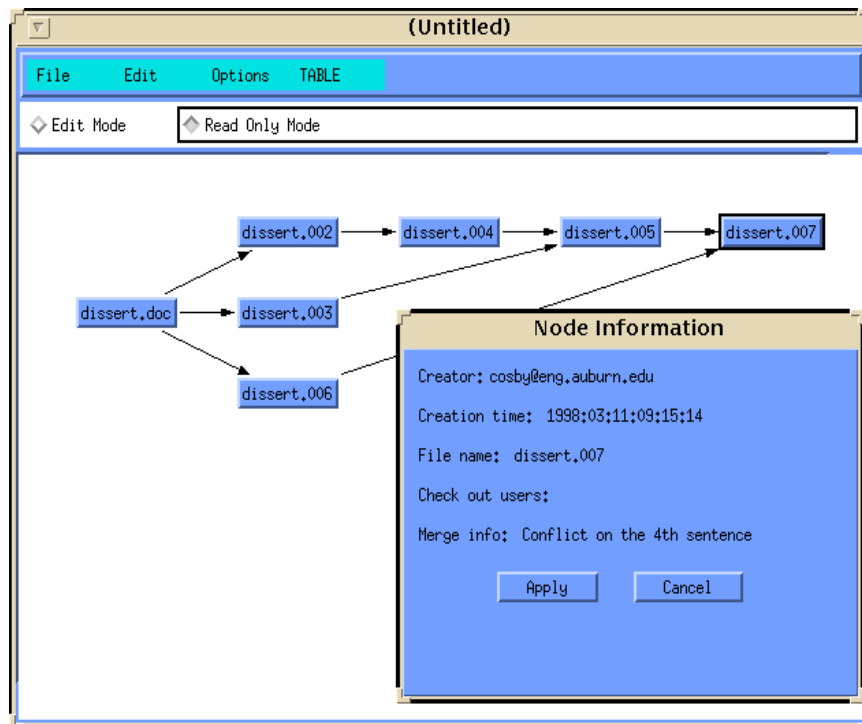


Figure 5a The version management editor

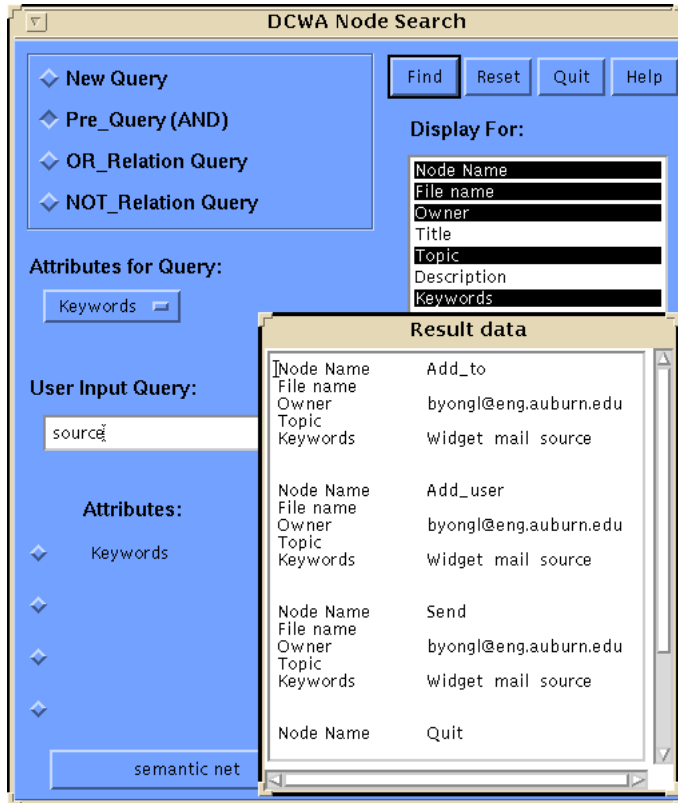


Figure 5b The query tool

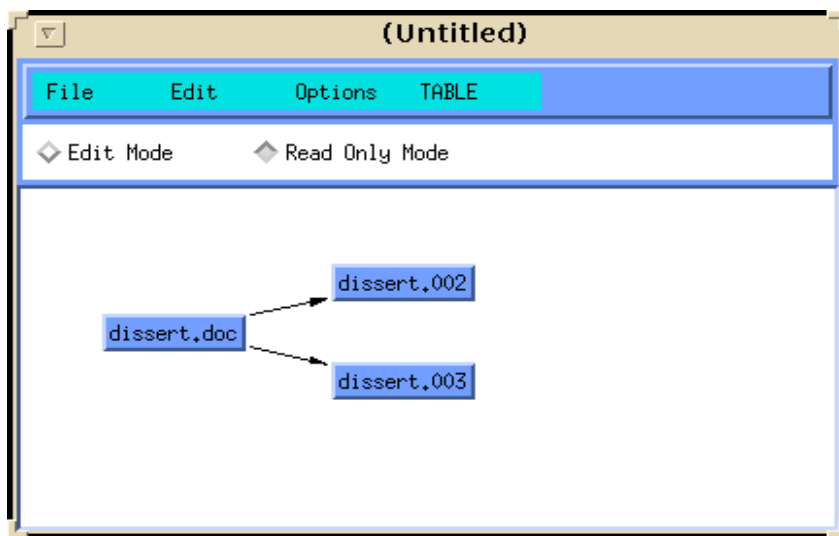


Figure 6. The version management editor with two new versions

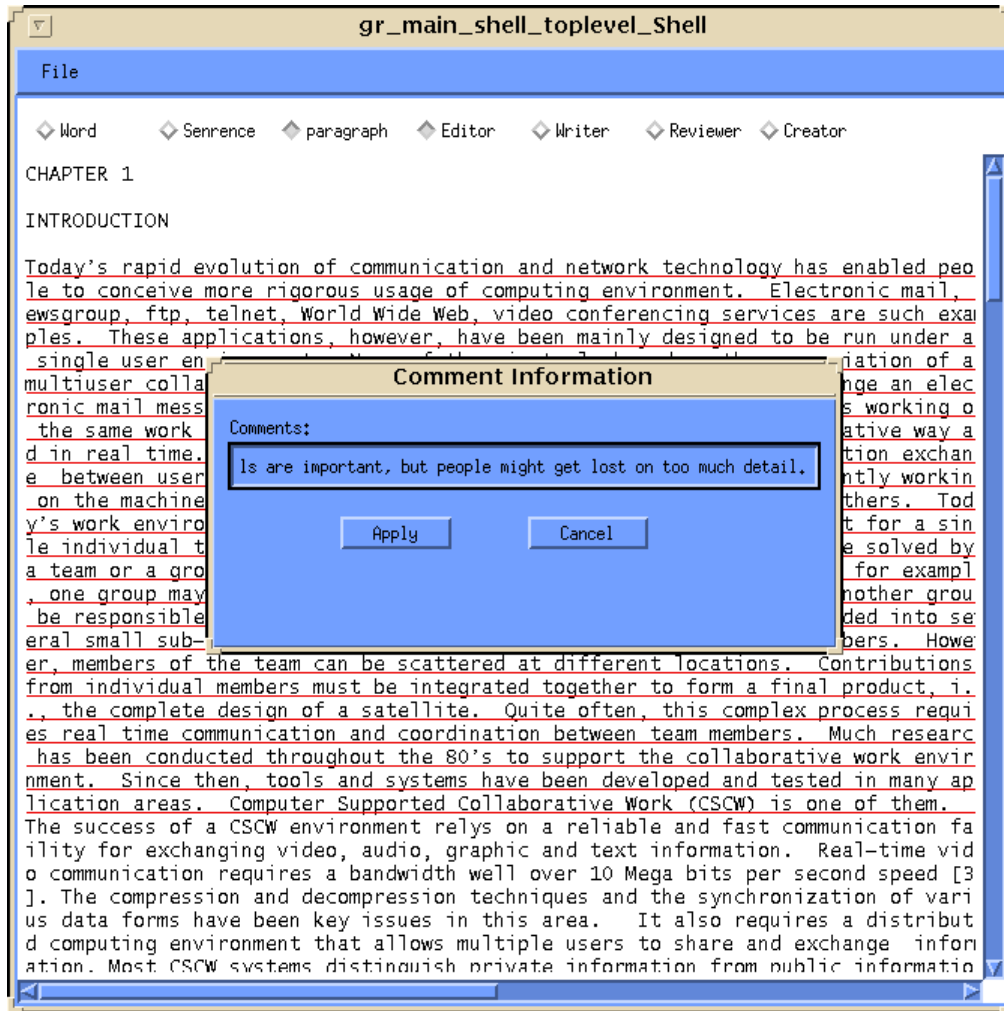


Figure 7. Content of the initial document

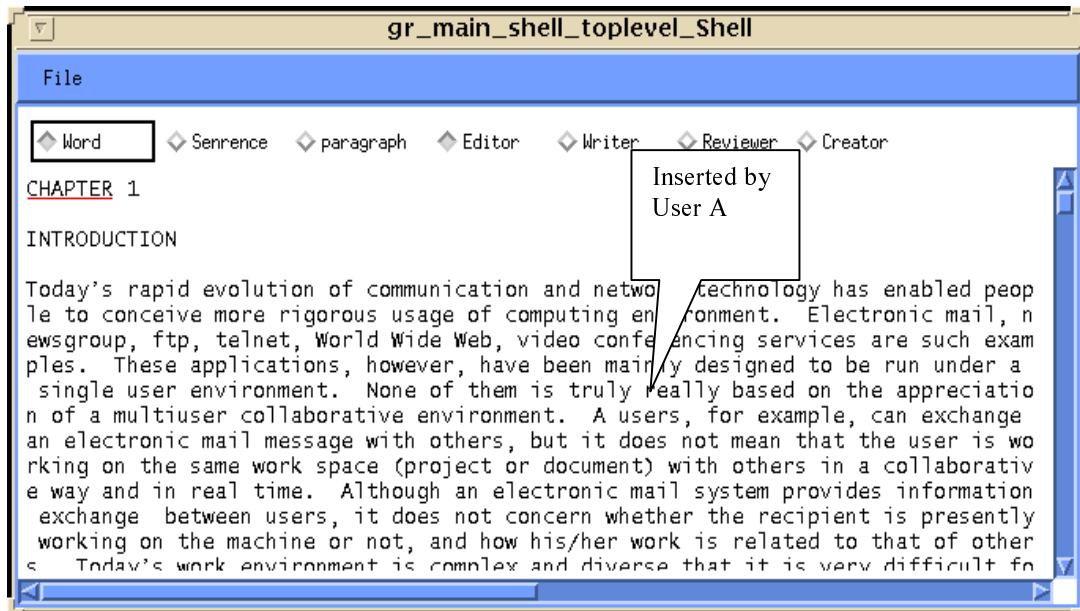


Figure 8. Version edited by user A

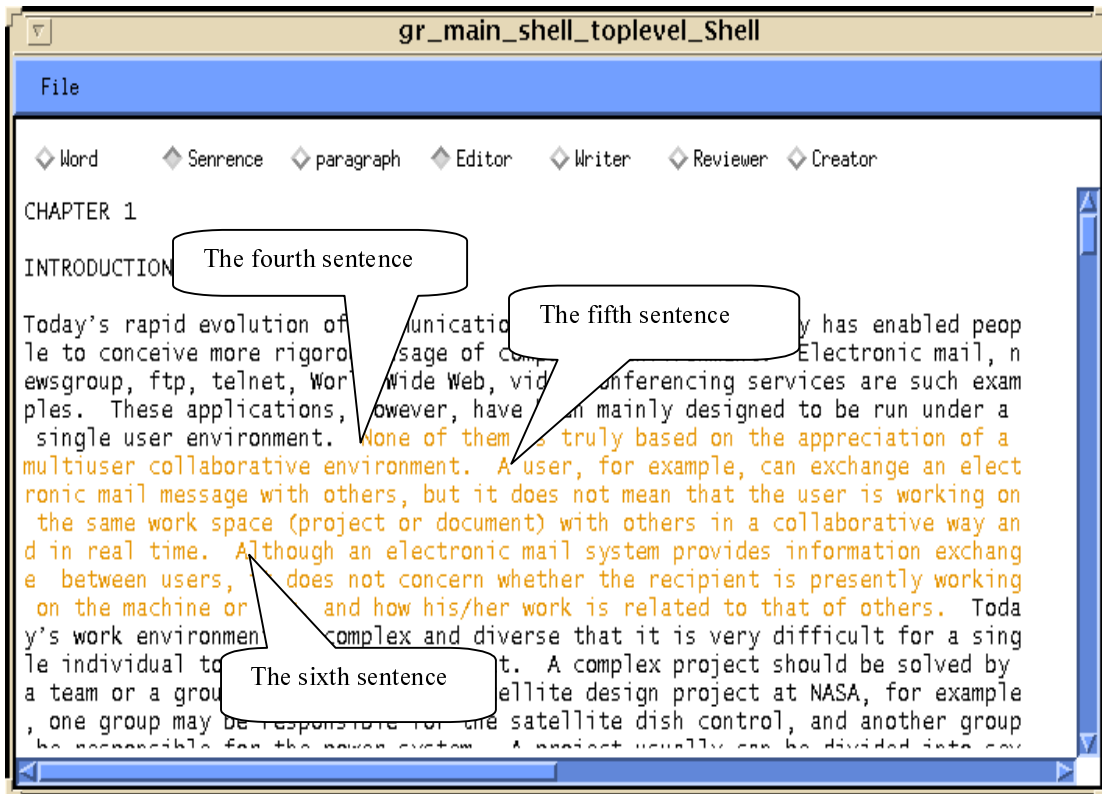


Figure 9 The text editor marks the deleted sentences in gray

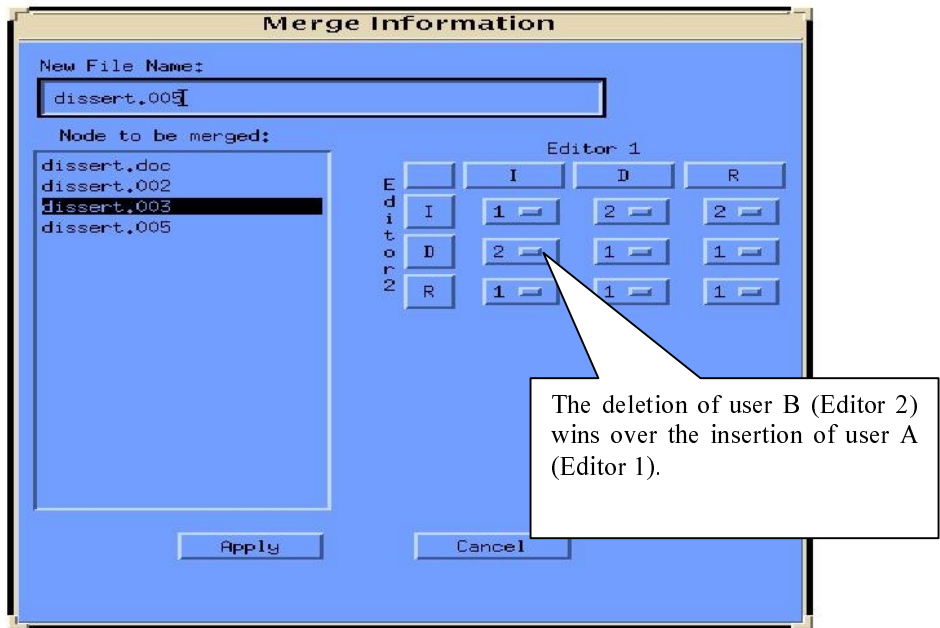


Figure 10 Merge table applicable to user A (Editor 1) and user B (Editor 2)

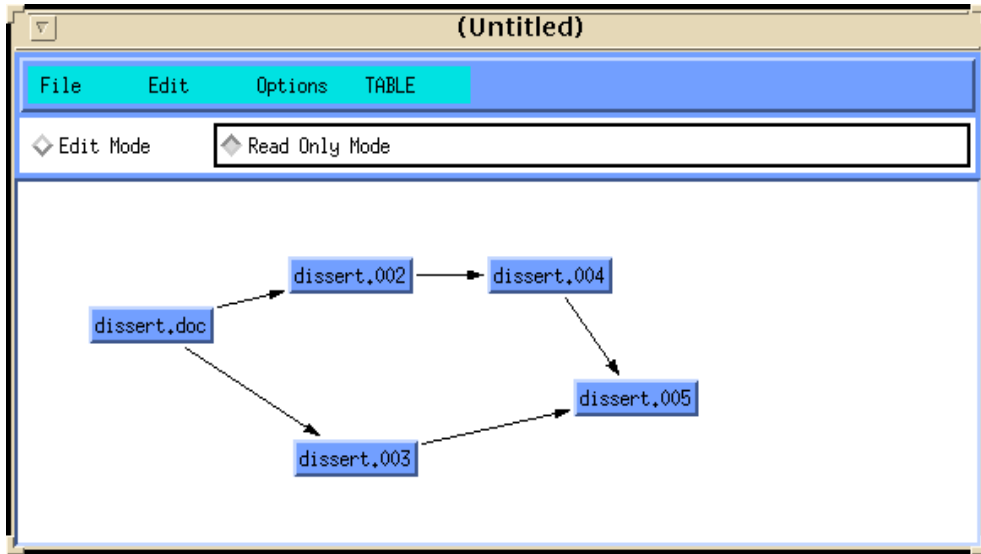


Figure 11. Version structure after the merge of 'dissert.003' and 'dissert.004'



Figure 12. Contents of 'dissert.005' with AID tags visible

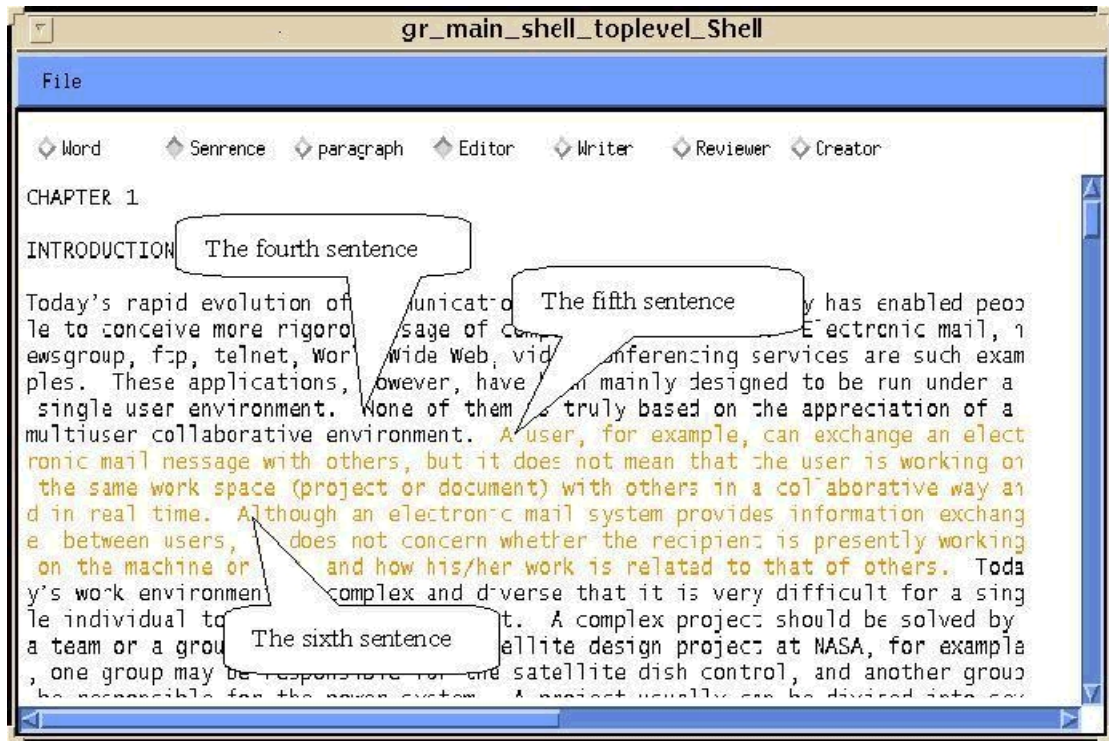


Figure 13. Contents of the merged node 'dissert.007'