An Integrated Approach to Version Control Management in Computer Supported Collaborative Writing

Byong G. Lee, Kai H. Chang, and N. Hari Narayanan
Department of Computer Science and Engineering
107 Dunstan Hall, Auburn University, Auburn, AL 36849-5347 USA
E-mail: {kchang, byongl, narayan}@eng.auburn.edu

Abstract- Computer Supported Collaborative Writing (CSCWriting) is an interactive application that allows multiple distributed users to collaborate on a shared document. To support the interaction among users, many CSCWriting tools facilitate synchronous communication. However, synchronous communication alone cannot be employed successfully due to slow network environment and various collaborative writing modes. These problems demand a new approach that aims at a simple, low-cost asynchronous collaboration environment supported by a version control system. However, since current version control techniques have been developed for single user environments, these can not be directly incorporated into CSCWriting environments. That is, these do not have the ability to address social/cognitive aspects, such as group awareness and access control. This paper describes issues regarding various physical and social artifacts required for version control in a CSCWriting environment, effective mechanisms for managing such artifacts, and an integrated approach to combining the artifacts.

1. Introduction

Computer Supported Collaborative Writing (CSCWriting) is an interactive application that allows multiple distributed users to collaborate on a shared document. To support the interaction among users, many CSCWriting tools provide various types of communication tools. Until now, many studies on CSCWriting have been focused on the development of synchronous communication tools based on the 'contingency theory' by Galegher and Kraut [1]. The contingency theory indicates that, while text based communication (e.g. electronic mail) is sufficient for the purpose of exchanging brief information or idea, complex intellectual group work such as planning and brainstorming process normally requires high levels of direct, face-to-face communication. It also shows that highly interactive group work can be improved by utilizing multimedia conferencing tools with text, audio, and video capabilities.

However, synchronous communication alone cannot be employed successfully in some collaborative writing contexts due to the following reasons:

- Network bandwidth and delay: Galegher and Kraut [1], in their study, emphasized the importance of interactivity and expressiveness in determining the success of synchronous communication. The interactivity and expressiveness was measured, respectively, by the speed of feedback (responsiveness) and by the degree of intimacy (e.g., eye contact, facial expression, gestures and body language) between group participants [2]. However, these two properties are influenced by network latency. That is, a slow network may degrade the responsiveness and the expressiveness during synchronous communication by causing unacceptable delays and out of synchrony between multiple data streams [3].
- Inconsistency: For a network environment, such as the Internet on a mobile network, the network connection is often unstable. In particular, a mobile network frequently disconnects users from the current state to save power or to relocate the server [4, 5]. As the network connection becomes unstable, information inconsistency may occur. In this case, maintaining consistency during synchronous communication is very expensive.

- Different collaboration modes and paces: It is known that synchronous communication is not necessarily required for all phases of collaborative writing. For example, during the writing and editing phases in collaborative writing, writers are most productive when no interference from other users is involved [6]. That is, users may simply wish to independently manipulate the document or try out different alternatives (versions) of the document. Besides, synchronous communication is not possible among users who are located at different time zones such as East Asia and United States (at least 12 hours difference).

These problems demand a new approach that aims at a simple, low-cost asynchronous collaboration environment such as a replicated architecture. In a replicated architecture, shared information or documents are replicated across all distributed sites. Changes or updates are first reflected at the local site, then at all other sites at some later point in time to maintain consistency (through the merging process). By replicating shared information at different sites, network overhead can be reduced significantly. This concept is well represented and supported in a system called Version Control [7, 8]. In a version control system, documents are replicated across all distributed sites and are organized and maintained so that users can easily navigate, track, and merge different documents with the assistance of differencing and merging utilities.

However, since existing version control techniques have been developed for single user environments, these can not be directly incorporated into CSCWriting environments. That is, most version control systems do not have the ability to address social/cognitive aspects, such as multiple roles, group awareness and access control. For example, during the collaboration, a user may want to see how other users have changed a sentence. He may also want to change the granularity level of the current section he is referring to or may want to have different access rights to different parts of an object. The feasibility of this social/cognitive information is crucial to the success of a CSCWriting system, and is normally represented in terms of interactivity and expressiveness in design criteria. In addition, in existing version control systems, the processes of writing and versioning are not tightly coupled. That is, document editing information is not utilized in version maintenance. This separation of writing from versioning may cause information inconsistency, thus degrading system integrity and flexibility.

To remedy the problem, a new integrated design infrastructure is necessary. Considering the four criteria, e.g., interactiveness, expressiveness, integrity and flexibility, this paper discusses:

- Various physical and social/cognitive artifacts required for version control under CSCWriting environment,
- Effective mechanisms for managing such artifacts, and
- An integrated approach satisfying the four criteria.

2. The Role of version control in CSCW

The major feature of version control is its ability in providing structured but relaxed consistency control in a distributed and open collaborative system. Particularly, in a Wide Area Network (WAN) environment, where the long time delay in network communication limits the usefulness of traditional synchronous collaboration, version control provides an alternative mechanism for managing concurrent changes. Some of the important roles of version control are summarized below.

- Optimistic concurrency control: In a tightly connected collaboration, an exclusive locking scheme is used to achieve consistency on logical state of artifacts -- an artifact can not be updated by two or more users at the same time. This scheme becomes fatal when one of the users possesses the lock indefinitely. Optimistic concurrency control is an approach to solving the problem by allowing every user to acquire an immediate tentative approval at anytime [9]. This optimistic approach can be implemented with the support of version control. That is, a user may checkout a copy of document as an approval of tentative lock. Then he can start editing the document with his own pace, and, later, when the user has a true permission to write, the edited version can be merged with either the original document or other user's version.
- Repairing the temporary inconsistency in mobile computing: Since mobile terminals are often disconnected from the network, transactions must often be processed locally on the cached data. Thus, users have to run an application on

the local portable computer while being disconnected from a network and process cached portions of the object for a long period of time [5]. In this case, modifications on two different versions may violate consistency. Exclusive or optimistic lock scheme alone may not be suitable in this case since the lock itself can be lost during the exchange, and implementation of this scheme on an unstable network environment is expensive [5]. A similar problem can be found in a Wide Area Network (WAN). The delay of communication and possible loss of data on WAN will result in inconsistency of exchanged information. Version control can be applied to solve this problem by replicating all the states of shared objects into the local site.

- Asynchronous collaboration: Users may often wish to work independently on the information and try different alternatives in their own work pace. For example, in an individual document writing stage, a user may not want to be bothered by other user's intrusion or interruption. Each user can cache a local copy of the document and proceed with his own work. Later, results from each user are merged to produce a final version of the document by using the version control system.

3. Issues in version control

Dewan [9] defined the term, version control, as "*an application to manage dependencies between different instances of the same objects, organize them into meaningful structures (e.g., sequences, trees or DAGs) and allow operations such as navigation or computation on them*". He further addressed that the content of any particular object can be determined and computed through differencing, merging or tracking among several different variants of the same object. Dewan's definition indicates that versioning, differencing, merging, and tracking play important roles in a version control system. However, in real systems, these mechanisms are not designed to incorporate with various social/cognitive activities involved in CSCWriting. For example, users may want to control the granularity of versions such that one editor may wish to create a version with the whole document as a piece, while others wish to create a version with sections and paragraphs. Users may also want to see who has created a document and how changes have been made to certain parts of the document. Users sometimes may want to select a particular version of a document and make changes to it. Following sections contain detailed discussion of these issues.

3.1 Differencing system

The fundamental component of a version control system is a differencing mechanism that finds plain text differences as well as conflicts among versions. Conflicts arise if either the same object has been edited in multiple versions or operations performed are not compatible with others, or if the ranges of granularity in each version, such as word, sentence, phrase, or entire document, are different [9]. Although more sophisticated systems such as COOP [10] and Timewarp [11] are able to detect conflicts, they only inform the user about the existence of conflicts for the user's attention. They do not provide any effective mechanism to resolve the conflicts.

Generally, there exist three different approaches for a differencing system. The first and the most popular approach is to compare plain text, which finds the longest common sub-string of two versions, and then to compute a distance delta from the common sub-string. The distance delta is simply an edit script consisting of deletion and insertion commands with distance values that can generate one version from the other. The second approach is to use semantic differences by constructing a dependency graph for each object, merges them into one dependency graph and checking for conflicts [12, 13]. Both plain text comparison and semantic differencing methods may impose significant overhead on the system as the number of versions to be merged increases and thus may degrade user responsiveness. It is also difficult to associate additional social and cognitive information into the comparison. The last approach is to keep the command history that performed the editing [12]. Consistency is maintained by exchanging history commands between versions. In this technique, however, redundant commands (as the history grows in length) producing the same effect may waste time and memory space.

3.2 Merging

Merging allows multiple versions to be joined together, producing a new version representing the union of the actions taken in the previous versions [14]. However, merging is not trivial if conflicts of social/cognitive information exist. That is, two edited versions may be merged with different granu-

larities or different roles. Most merging tools, however, are lack of specifying and resolving these kinds of conflicts. The effective merge tool should provide users with a reliable way to define, detect, and resolve various types of conflicts.

3.3 Version management and propagation

In collaborative writing, multiple versions are frequently created when a user updates a shared version. The newly created version can be either a local version that is maintained and accessed by only the local user, or a global version that is visible to all users. Then, a version control tool should provide the capability to maintain those different versions as appropriate. For example, as a user creates a local version, a corresponding local version tree should be created and maintained. Later when a user decides to merge a specific version from the local version tree, the tool should be able to specify the version and to maintain dependency of the related versions across the local and global versions. The tool should also be able to propagate changes made on objects, such as word, sentence or phrase, through successive versions.

3.4 Group awareness

Group awareness allows each user to be informed of the work done by others, including the presence of other users, the creation and availability of new objects, changes to objects, etc. Most synchronous collaboration systems provide user awareness through color, shared scrollbars, observation view, audio beeps, or video conferencing, etc [15]. However, the solution should be different for an asynchronous system. In asynchronous mode, a new awareness mechanism, called delayed awareness, is introduced. That is, a user can see the contributions made by others after a delay of time. In Alliance [15], all editing operations are recorded first at the local site and then transmitted to the other users after a delay of time. However, transmitting a long, repeated sequence of editing operations can be tedious and time consuming, thus may easily distract a user's focus. Although Alliance provides an option to select and transmit the particular changes by adopting some form of user validating mechanism, it can impose a new overhead on the system as the system needs to maintain the dependencies of every operation.

Timewarp [11] provides asynchronous awareness in terms of the creator of each specific version, specific operations performed, differences between versions, and summarization of the changes. However, this approach cannot effectively provide expressiveness in user activity because only the final states of operations are recorded in the versions. It does not provide intermediate information such as how a user arrived at a conclusion and what is the user's intention of a particular operation. Therefore, it is desirable to design a new mechanism that will provide rich awareness information while reducing system overhead.

3.5 Access control

Access control used in non-collaborative environments does not meet the requirements of collaborative environments due to the dynamic nature of roles in CSCWriting. Shen and Dewan [16] suggested that collaborative access control should associate various levels of granularity with collaboration rights so that it allows individuals to specify changes in access rights at any time. In addition, various access rights such as read, freeze, or version derivation permission should be considered [17].

3.6 Central or distributed architecture

Distributed architecture processes most of the version functionality at the local computer and maintains a version tree at each site. By localizing the major portion of operations, the distributed architecture gives better interactive performance.

On the other hand, centralized architecture implements most of the version functionality at a single, central location by designating it as the primary. When a replicated file is to be updated, all changes are sent to the primary server, and the server should maintain consistency. Although this architecture may have a point of failure problem (when the primary server crashes), its implementation is easier compared to the distributed architecture.

4. Related work

Many existing systems offer only a limited combination of the multiple functionalities described earlier. This section provides a high-level description and comparison of some of the available version control systems in the commercial and research domains.

RCS (Revision Control System) [18] is the most popular public domain version control system.

A new version is created when an existing version is edited and is stored in the normal file system. The created versions are organized in a tree structure by using version management commands such as Create, Check-In, Check-Out, and Cancel-Check-Out. The tool offers facilities for merging and automatic marking of versions using serial numbers. For consistency control, an exclusive lock is used to prevent conflicts in the modification on the same version. Comparison and finding differences among different versions is done by using a text based differencing system, but the system can not detect or determine a conflict out of differences. Additional information such as author, date, or revision status can be embedded in the versioned file using special keyword delimiters. However, since none of these features is supported in a GUI, usability of these features is neither intuitive nor simple.

CVS (Concurrent Versions System) [19] is a collection of tools on top of RCS. However, unlike the RCS, the checked-out documents or versions are not locked in CVS, thus parallel work on the same document is possible. It also supports automatic merging as long as the modified objects are compatible with each other. If not, manual merge is triggered.

MICROCOSM [20] is an open hypermedia system, originally developed for the PC environment. Although it is based on the RCS system, users can create and select a new version by allowing navigation in the version tree (RCS does not allow this navigation). However, this system does not support true parallel editing on the same document. It uses the same locking scheme, like RCS, to prevent conflicts across versions. Furthermore, the lock not only freezes the selected version but also all the derived versions. Another drawback of this system is that it lacks of access control. That is, it does not allow user to have or change access rights. An interesting feature of MICROCOSM is that it separates the version link information from the version editing information so that any loss of version editing information does not affect the link relation among versions.

In COOP [10], complete information about version link and editing history is stored at the last version. Older versions can be reconstructed by tracking the changes backward [21]. Merging is guided by ACTIVE DIFF [21], which provides a set of default rules to identify conflicts and to produce a suggested merged version. COOP also provides a primitive awareness mechanism by simply enabling all users to share and view the same version graph. However, it does not provide any mechanism for users to deal with granularity change.

PREP [14] supports an enhanced mechanism for supporting social roles and cognitive activities. This reduces the coordination problem by specifying responsibilities and patterns of interaction. The system also supports flexible change of collaboration mode in each cognitive phase. Specific cognitive activities are also defined such as jotting, drawing, writing, and gestures. However, its differencing mechanism is based on plain text comparison.

Timewarp [11] is the most recent version control tool available in the research-oriented domain. It contains version management, merging, and group awareness tools. Timewarp also provides a GUI to handle version insertion, deletion and modification. In addition, an effective propagation mechanism maintains version consistency throughout the version tree. However, Timewarp is not designed for text editing, but for graphics editing. This means that there is only a single granularity level available, e.g., each graphical object. It is not possible to control hierarchical levels of granularity such as phrase, sentence, or word. It also does not allow true parallel editing.

The systems reviewed in this section show the following common deficiencies;

1) True parallel editing of a document is not possible in most systems.
2) Functionality of differencing and merging is confined to finding only the differences (not detecting conflicts).
3) Social and cognitive aspects are not well supported.
4) Versioning process is separate from the writing process.

Based on this review, a new version control system is proposed and introduced in the next section.

5. Design of version control

Figure 1 shows the different components of the proposed system. Each component is described in the following sub-sections.

5.1 Differencing tool

From the earlier review, common problems in most existing differencing tools include:

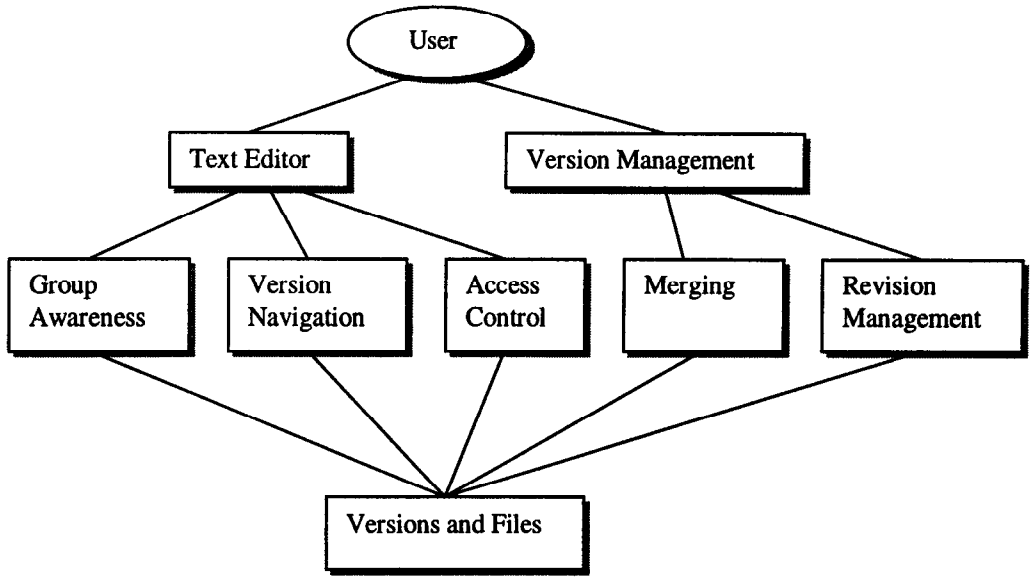

Figure 1 Version Control System

1) They are time and space consuming, due to the burden of differencing scheme, thus resulting in the degradation of system performance.
2) They do not allow user to interact with various social and cognitive aspects such as resolving conflicts, changing granularity and role, and managing access control.
3) They are lack of a reliable and efficient storage model to include social and cognitive information.

One possible solution to reduce the overhead of a differencing tool is to keep track of all the editing operations and to make each change unique and apparent. In order to achieve this, an appropriate editing notation has been defined by using a unique Activity Identification (AID). That is, every object, such as a sentence or word in each document, should be addressable from appropriate activity types. The editing activities defined in AID include Create, Insert, Delete, and Replace. Other activities, such as Cut and Copy, are not included since they can be represented by a sequence of the basic AID activities. The address of every element in a document is assigned with a unique real number starting at 1.0 so that the number complies with the object offset in the document. The address is also global, hierarchical and stable, thus a particular sentence or word from one version can be easily detected at the same address in other versions. For the purpose of granularity control, AID facilitates the granularity specification with Phrase, Sentence, and Word. The formal syntax of AID is:

*{< (P|S|W) : object_offset : (C|I|D|R) >} **

where P: Phrase, S: Sentence, W: Word, C: change, I: insert, D: delete, R: replace.

A new object address is determined by finding the middle point between the addresses before and after. For example, if a new sentence is inserted between the 4th and the 5th sentences, a new address, 4.5, is assigned to the inserted sentence with AID of <S:4.5:I>. This address tag is inserted to the front of an element every time the element is edited or updated (e.g., <P:1.0:I> <S:6.7:D> <W:3.0:R>). Note that the number generation scheme conforms to the hierarchical structure of a document such as phrase->sentence->word, and that

the object offset always originates from its immediate parents' object IDs.

This addressing scheme looks similar to the history command mechanism, in that each edited operation is represented by the performed activity and its corresponding address. However, in the history command mechanism, a newly created object would force other objects to change their addresses to maintain consistency. For example, if user A inserts a new line between lines 2 and 3, the rest of the line numbers in the document should be updated accordingly, i.e., 3->4, 4->5, 5->6, and etc. In addition, the updated address information should be kept at somewhere so that it can be used to merge with other versions. For example, from the previous case, if user B deletes line #5 from his own version, then line #6 should be deleted from user A's version, not line #5. Maintaining such a long sequence of update information is a time and space consuming process.

The AID scheme always creates the address space constructively and uses this space as the basis for differencing and merging. That is, it always creates a new address and never destroys or changes an existing address space. For example, as a user inserts a new sentence between the 4th and the 5th lines, a new address, 4.5 for the inserted line, is created by computing (4+5)/2=4.5. Even with a deletion or a replace operation, AID does not affect other addresses. It simply replaces the existing activity tag of AID (e.g., <P:1.0:R> <S:6.7:R> <W:3.0:D>). This unique AID scheme saves the time wasted on updating other address spaces, and makes navigation across different versions easier and more straightforward (e.g., sentence #5 in one version can be directly found at the same address in other versions). Also, there is no space wasted on recording and maintaining the long sequences of update information.

Another advantage of AID over the history command method is that it eliminates the redundancy problem, which appears in some editing activities. In normal writing, it is possible that a user may repeat the 'Replace' operation on a word or a line many times, and the result of applying the sequence only reflects the effect of the last operation. This redundancy problem is eliminated in AID mechanism since each address space always represents and reflects the last state of the activity.

The AID scheme also enables flexible granularity merging by allowing different types of granularity within the AID, i.e., merge can be achieved at various levels of granularity at any time. If the specified granularities are different between users, merge is performed at the larger granularity level by default. However, there is one problem. For example, let's assume that user A made a change on an existing sentence # 7, thus modified the tag from <P:1.0:C><S:7.0:C> to <P:1.0:C><S:7.0:R>. Now user B deletes the third word in sentence #7 which yields <P:1.0:C><S:7.0:C><W:3.0:D>. Each tag is placed in front of each edited element. When these two users want to merge their work at the sentence level, the system may not detect or indicate any conflict, thus producing the wrong result in the merged version. Looking at those AID tags at the sentence level, the system is not able to tell if there is a conflict (e.g., there is only one 'replace' operation in both AID tags). Since the deletion of a word also affects the sentence and the phrase that the word belongs to, the system should change the state of the higher granularity level as well, e.g., to <P:1.0:R><S:7.0:R><W:3.0:D>. This new AID tag indicates a conflict in sentence #7 because both versions have the replace activity mark in AID. If the conflict can not be resolved by the system automatically, the user would have to resolve the conflict with the assistance of a proper interface tool. This tool will be discussed in section 5.2.

There is a special case of conflict detection that should be considered in the AID scheme. That is, there is a possibility that both users insert a new sentence, e.g., 4.5, in their individual versions. Should this be treated as a conflict? Actually, insertion of two sentences usually does not harm the consistency of free style text writing. If it is not considered as a conflict, one of the versions should have a different address by assigning the next available real number between 4.5 and 5.0. In this case either one of the address spaces should be assigned with the new address 4.75. Then this set of changes should be recorded in the version information for navigational purpose: {version#2:4.5<-4.75, version#2:5.75<-5.5...}.

This new improved AID scheme will eliminate the need of a separate differencing system. Unlike the history command, comparison is straightforward and redundancy problem is removed. AID also allows users to find the differences and merge the changes at various granularity levels. The AID also helps keep storage size reasonable since the maintenance of update information is not necessary.

5.2 Merging tool with access control

The described merge problem can be solved by a merge matrix that is proposed by Dewan [12].

Sequence	Insert Element #	Delete Element #	Replace Element #
Insert Element #	Both Operations	Both Operations	Both Operations
Delete Element #	Both Operations	Row Operations	Conflicts
Replace Element #	Both Operations	Conflicts	Conflicts

Table 1 Dewan's merge matrix

It utilizes an access control matrix and has a row and a column for each editing operation that can be performed on an object. Each row represents one operation of a user and each column represents one operation of another user. The entries of the matrix specify the resulting action. A merge matrix can be defined for each structural level of a document. Table 1 shows a sample merge matrix.

This scheme has a fundamental problem in an N way merging, i.e., merging N versions of an object. The merge matrix of N dimensions and M operations would require space complexity of $O(m^n)$. Filling in such numbers of table slot is too tedious and time-consuming [12]. To remedy this problem, merge table can be modified such that each dimension represents a social role such as writer versus commentator, writer versus editor, editor versus commentator. Thus the table entries are constantly bounded by $O(m^{\#\text{ of roles}})$. Since everyone assumes a role in collaborative writing, assigning such roles in merge matrix is reasonable and natural. However, there may exist a role conflict where both users want to edit the same object with the same role. To resolve this role conflict, extra table entries should be added such as writer versus writer, editor versus editor, commentator versus commentator, etc. In this case, the maximum table space can be bounded to $O(m^{\#\text{ of roles}} + (\#\text{ of roles})^2)$, which is still less than $O(m^n)$ for n > (# of roles). Normally, the value N increases much faster than the number of defined roles, as the number of derived versions increases over time. In addition to reducing the space, this role assignment within a merge matrix satisfies the need of associating access control and merging policy into version control.

5.3 Version management and navigation tool

The simplest approach to version management is simply to keep the most recent version. This approach is adopted in many file systems such as UNIX and Liveware [13]. However, when divergent versions are generated, user intervention is usually necessary. Minor and Magnusson [22] emphasized the use of an appropriate navigation tool that allows users to interact with a version at any point in its version tree. Users may refer to a specific piece of information from another version, and sometimes may want to change a portion of the contents. Thus, an effective navigation tool should provide the ability to select any version and to propagate changes to later versions along the version tree. However, in most version control systems, when an early version is changed, they do little more than creating a new version branch off the version tree. They do not provide an effective propagation mechanism. These changes, without proper propagation mechanism, may cause inconsistencies between versions along the tree. To propagate the changes of an object correctly, the system needs to know where a particular object is located in other versions. For this, Dix [5] proposed to use dynamic pointers.

Dynamic pointers allow user to relate a location in one version to the same location in another version, even if the version is located on a different branch [5]. For instance, if user A inserted two sentences between the second and the third sentence, and user B deleted the second sentence, then the corresponding relation for each of the sentences would be

x: O-> A = { 1->1, 2->2, 3->5, 4->6}, y: O-> B = { 1->1, 3->2, 4->3}

If user B wants to find the third sentence in the version which was created by user A, the function can be applied as p@B= 3. By applying an inverse function such that $p@O = y^{-1}$ (p@B), the sentence position 4 is computed from the set. Then applying p@A= x(p@O) =6 gives the corresponding sentence position for the user A's version. This scheme is very useful for navigating multiple versions along the version tree. However, it may impose an overhead on the system. For example, if a user inserts a sentence in the beginning of a docu-

ment, the changes in the rest of the sentence numbers should also be kept and maintained in a set.

The AID eliminates this overhead since the address assigned to each object would never be changed as changes are made to the version. Finding a particular object is easy by simply referring a unique ID.

5.4 Awareness control

Since each address space in AID always represents the final state of operations, it may not provide effective awareness service. To increase the awareness effects, a mechanism called object link (OL) can be utilized. Instead of providing all the intermediate operations as seen in Alliance [15], the object link tool relates a specific editing operation or social/cognitive information upon user request. For example, a user can use an OL tool to see how a specific sentence has been changed and evolved across versions by linking every operations performed on that object in timely order. The OL tool should be able to display the related information in any way that a user wants to display. To utilize this object link, AID needs to be revised to include additional social/cognitive information, e.g., {< (P|S|W): object_offset : (C|I|D|R) (T|R|A)>} *
where T: Time of creation, R: Role, A: Author or Creator. Each of these elements can be added upon user request and linked across versions. In addition, the proposed system will adopt a preventive facility to reduce possible conflicts in advance. Any previously updated or modified objects including conflict objects will be displayed on the document with a different color so that a user can be informed of possible conflicts.

6. Conclusion

CSCWriting systems must provide extensive social/cognitive activity capabilities such as means of communication, scheduling and coordination, support for social roles, access control, and awareness control to improve the interactiveness and responsiveness. With all these requirements, network latency remains a bottleneck in the success of a CSCWriting system. A version control system can be adopted to counteract network latency. However, traditional version control systems do not provide effective mechanisms for the social/cognitive aspects.

In this paper, we described a new version control system that provides better flexibility, integrity, responsiveness and expressiveness. With the AID tag, granularity control, version navigation, differencing, and merging become simple and easy. The unique address scheme also eliminates the requirement for large storage capacity for the navigation. Access control is improved addressing social roles to the merge table. With merge tables, a user can change or update the specification at any time during the collaboration. In addition, since AID is incorporated into the document itself, system integrity is enhanced.

However, the proposed system does not eliminate the need of synchronous communication tools. Synchronous tools are proven effective in tightly coupled collaboration. The proposed system can be a complementary tool where synchronous collaboration is problematic. Currently, the proposed system is under development. Upon completion, it will be integrated with our previous synchronous CSCWriting system, DCWA [23], and evaluated for its usability and flexibility.

References

[1] J. Galegher and R. Kraut, "Computer-mediated Communication for Intellectual Teamwork: An Experiment in Group Writing", Groupware and Authoring, Roy Rada (Eds), Academic Press, pp. 127--160, 1996.

[2] M. Sasse and M. Handley, "Collaborative Writing with Synchronous and Asynchronous Support Environments, Groupware and Authoring, Roy Rada (Eds), Academic Press, pp. 205--218, 1996.

[3] M. Prycker, Asynchronous transfer mode, Ellis Horrwood series, Chapter 1, pp. 14--25, 1993.

[4] T. Imeilinski and B. Badrinath, "Mobile Wireless Computing Challenges in Data Management", DataMan Project Technical Report, NSF (SGER) IRI-9307165.

[5] A. Dix and R. Beale, "Information Requirements of Distributed Workers", Remote Cooperation, Alan Dix and Rullsel Beale (Eds.), Springer-Verlag, pp. 113--143, 1996.

[6] S. Baydere, T. Casey, S. Chuang, M. Handley, N. Ismail and M.Sasse, "Multimedia

Conferencing as a Tool for Collaborative Writing: A Case Study", Computer Supported Collaborative Writing, Mike Sharples (Eds.), Springer-Verlag, pp. 113--136, 1993.

[7] P. Dewan, "Principles of Designing Multi-User User Interface Development Environments", Technical Report, Computer Science, Purdue University, 1995.

[8] P. Dewan, "Multiuser Architectures", Technical Report, Dept of Computer Science, University of North Carolina, 1996.

[9] P. Dewan and J. Munson, "The role of Version Control in CSCW Applications:A Position Statement", Workshop on the Role of Version Control in CSCW applications, Proceedings of the ECSCW, 1995.

[10] B. Magnusson, "Fine-Grained Version Control in COOP/Orm", Workshop on the Role of Version Control in CSCW Applications, September 1995.

[11] W. Edwards and E. Mynatt, "Timewarp:Techniques for Autonomous Collaboration", Human Factors in Computer Systems, Proceedings of CHI 97, pp. 218--225, 1997.

[12] J. Munson and P. Dewan, "A Flexible Object Merging Framework", Proceedings of CSCW 94, pp. 230--242, October 1994.

[13] S. Horwitz, J. Prins and T. Reps, "Integrating Non-interfering Versions of Programs", ACM Transactions on Programming Languages and Systems, 11(3), pp. 345--387, 1989.

[14] C. Neuwirth, R. Chandhok, D. Kaufer, P. Erion, J. Morris, and D. Miller, "Flexible Diff-ing In a Collaborative Writing System", Proceedings of CSCW 92, pp 147--154, Nov. 1992.

[15] D. Decouchant, V. Quint and S. Kiesler, "Structured and Distributed Cooperative Editing in a Large Scale Network", Groupware and Authoring, Roy Rada (Eds), Academic Press, pp. 265--295, 1996.

[16] H. Shen and P. Dewan, "Access Control for Collaborative Environments", Proceedings of CSCW 92, pp. 51--66, October 1992.

[17] D. Hicks and A. Haake, "Workshop Summary", Workshop on the Role of Version Control in CSCW Applications, pp. 7--16, September 1995.

[18] W. Tichy, "RCS-a System for Version Control", Software-Practice and Experience,17(7), pp. 637--654, July 1985.

[19] D. Coleman, Groupware: Collaborative Strategies for Corporate LANS and Intranets, Prentice Hall PTR, 1997.

[20] W. Hall, I. Heath, and H. Davis. "Microcosm: an Open Model for Hypermedia with Dynamic Linking", Proceedings of the European Conference on Hypertext '90, France, Cambridge University Press, November 1990.

[21] T. Mikkelsen and S. Pherigo, Practical Software: Configuration Management, Hewlett-Packard Professional Books, 1997.

[22] S. Minor and B. Magnusson, "A model for Semi-(a)synchronous Collaborative Editing", Proceedings of the Third European Conference on Computer Supported Cooperative Work, ECSCW 93, pp. 219--231, September, 1993.

[23] K. Chang, Y. Gong, T. Dollar, T. Gajuwala, B. Lee, and W. Wear, "On Computer Supported Collaborative Writing Tools for Distributed Environments," Proceedings of ACM conference 1995, pp 222-229, 1995.