Achieving Convergence, Causality Preservation, and Intention Preservation in Real-Time Cooperative Editing Systems

CHENGZHENG SUN
Griffith University
XIAOHUA JIA
City University of Hong Kong
YANCHUN ZHANG
The University of Southern Queensland
YUN YANG
Deakin University
and
DAVID CHEN
Griffith University

Real-time cooperative editing systems allow multiple users to view and edit the same text/graphic/image/multimedia document at the same time from multiple sites connected by communication networks. Consistency maintenance is one of the most significant challenges in designing and implementing real-time cooperative editing systems. In this article, a consistency model, with properties of convergence, causality preservation, and intention preservation, is proposed as a framework for consistency maintenance in real-time cooperative editing systems. Moreover, an integrated set of schemes and algorithms, which support the proposed consistency model, are devised and discussed in detail. In particular, we have contributed (1) a novel generic operation transformation control algorithm for achieving intention preservation in combination with schemes for achieving convergence and causality preservation and (2) a pair of reversible inclusion and exclusion transformation algorithms for stringwise operations for text editing. An Internet-based prototype system has been built to test the feasibility of the proposed schemes and algorithms.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distrib-

This work was supported in part by an NCGSS grant from Griffith University, Australia, and a Strategic Research Grant no. 7000641 from City University of Hong Kong.

Authors' addresses: C. Sun and D. Chen, School of Computing and Information Technology, Griffith University, Queensland 4111, Australia; email: {c.sun; d.chen}@cit.gu.edu.au; X. Jia, Department of Computer Science, City University of Hong Kong, Hong Kong; email: jia@cs.cityu.edu.hk; Y. Zhang, Department of Mathematics and Computing, University of Southern Queensland 4350, Australia; email: yan@usq.edu.au; Y. Yang, School of Computing and Mathematics, Deakin University, Victoria 3217, Australia; email: yun@deakin.edu.au.

uted Systems—*distributed applications*; D.2.2 [**Software Engineering**]: Tools and Techniques—*user interfaces*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*synchronous interaction*; *theory and models*

General Terms: Algorithms, Design, Human Factors

Additional Key Words and Phrases: Causality preservation, computer-supported cooperative work, consistency maintenance, convergence, cooperative editing, groupware systems, intention preservation, operational transformation, REDUCE

1. INTRODUCTION

Cooperative editing systems are very useful groupware tools in the rapidly expanding areas of CSCW (computer-supported cooperative work) [Ellis et al. 1991]. They can be used to allow physically dispersed people to edit a shared textual document [Ellis and Gibbs 1989; Knister and Prakash 1993; McGuffin and Olson 1992; Ressel et al. 1996; Sun et al. 1996a], to draw a shared graph structure [Greenberg and Marwood 1994; Karsenty and Beaudouin-Lafon 1993; Moran et al. 1995], to record ideas during a brainstorming meeting [Hymes and Olson 1992], or to hold a design meeting [Olson et al. 1992]. The goal of our research is to investigate, design, and implement cooperative editing systems with the following characteristics: (1) *real-time*: the response to local user actions is quick (ideally as quick as a single-user editor), and the latency for reflecting remote user actions is low (determined by external communication latency only); (2) *distributed*: cooperating users may reside on different machines connected by different communication networks with nondeterministic latency; (3) *unconstrained*: multiple users are allowed to concurrently and freely edit any part of the document at any time, in order to facilitate free and natural information flow among multiple users, as advocated by Ellis and Gibbs [1989], Hymes and Olson [1992], Dourish [1995], Sun et al. [1996a], and Ressel et al. [1996].

The requirements for good responsiveness and for supporting unconstrained collaboration have led us to adopt a replicated architecture for the storage of shared documents: the shared documents are replicated at the local storage of each participating site. One of the most significant challenges in designing and implementing real-time cooperative editing systems with a replicated architecture is consistency maintenance of replicated documents. To illustrate the challenging problems we are facing, consider a scenario in a cooperative editing system with three cooperating sites, as shown in Figure 1. Suppose that an (update) operation is executed on the local replica of the shared document immediately after its generation, then broadcast to remote sites and executed there in its *original form* upon its arrival. Three inconsistency problems manifest themselves in this scenario.

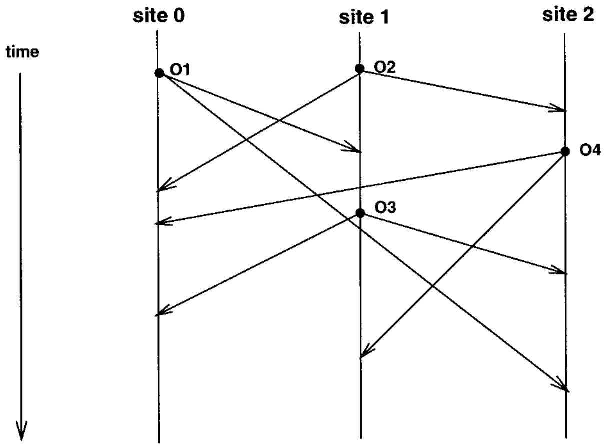


Fig. 1. A scenario of a real-time cooperative editing session.

Divergence. Operations may arrive and be executed at different sites in different orders, resulting in different final results. As shown in Figure 1, the four operations in this scenario are executed in the following orders: O_1, O_2, O_4, and O_3 at site 0; O_2, O_1, O_3, and O_4 at site 1; and O_2, O_4, O_3, and O_1 at site 2. Unless operations are commutative (which is generally not the case), final editing results would not be identical among cooperating sites. Apparently, divergent final results should be prohibited for applications where the consistency of the final results is required, such as real-time cooperative software design and documentation systems. The divergence problem can be solved by any serialization protocol [Bernstein et al. 1987; Birman et al. 1991; Lamport 1978], which ensures the final result is the same as if all operations were executed in the same total order at all sites.

Causality Violation. Due to the nondeterministic communication latency, operations may arrive and be executed out of their natural *cause-effect* order. As shown in Figure 1, operation O_3 is generated after the arrival of O_1 at site 1; the editing effect of O_1 on the shared document has been seen by the user at site 1 at the time when O_3 is generated. Therefore, O_3 may be *dependent* on O_1 (see Section 2 for a more precise definition about *dependence*). However, since O_3 arrives and is executed before O_1 at site 2, confusion may occur to the system as well as to the user at site 2. For example, if O_1 is to insert a string into a shared text document,[1] and O_3 is to delete some characters in the string inserted by O_1, then the execution of O_3 before O_1 at site 2 will result in O_3 referring to a nonexistent context. As another example, suppose the user at site 0 issues O_1 to pose a question by inserting "What public holiday is held in Victoria Australia on the first

[1] Throughout this article, text-editing systems and operations are used for illustration, but many of the illustrated concepts and schemes are generally applicable to other editing systems.

Tuesday in November?" into the shared text document, and then the user at site 1 issues O_3 to answer this question by inserting "Melbourne Cup" into the shared document; the user at site 2 would be puzzled by seeing the *answer* before the *question*. Apparently, out-of-causal-order execution should be prohibited for the sake of system correctness and meeting the requirement of synchronized interactions among multiple users in many applications. This causality violation problem can be solved by selectively delaying the execution of some operations to enforce a causally ordered execution based on state vector timestamps [Birman et al. 1991; Fidge 1988; Raynal and Singhal 1996].

Intention Violation. Due to concurrent generation of operations, the *actual effect* of an operation at the time of its execution may be different from the *intended effect* of this operation at the time of its generation. As shown in Figure 1, operation O_1 is generated at site 0 without any knowledge of O_2 generated at site 1, so O_1 is *independent* of O_2, and vice versa (see Section 2 for the definition about *independence*). At site 0, O_2 is executed on a document state which has been changed by the preceding execution of O_1. Therefore, the subsequent execution of O_2 may refer to an incorrect position in the new document state, resulting in an editing effect which is different from the *intention* of O_2 (see Section 2 for the definition about *intention*). For example, assume the shared document initially contains the sequence of characters "ABCDE." Suppose O_1 = *Insert*["12", 1], which intends to insert string "12" at position 1, i.e., between "A" and "BCDE"; and O_2 = *Delete*[2, 2], which intends to delete the two characters starting from position 2, i.e., "CD." After the execution of these two operations, the *intention-preserved* result (at all sites) should be "A12BE." However, the actual result at site 0, obtained by executing O_1 followed by executing O_2, would be "A1CDE," which apparently violates the intention of O_1 since the character "2," which was intended to be inserted, is missing in the final text, and violates the intention of O_2 since characters "CD," which were intended to be deleted, are still present in the final text. Even if a serialization-based protocol was used to ensure that all sites execute O_1 and O_2 in the same order to get an identical result "A1CDE," this identical result is still inconsistent with the intentions of both O_1 and O_2.

It should be pointed out that the above-identified three inconsistency problems are *independent* in the sense that the occurrence of one or two of them does not always result in the others. Particularly, intention violation is an inconsistency problem of a different nature from the divergence problem.[2] The essential difference between divergence and intention violation is that the former can always be resolved by a serialization protocol, but the latter cannot be fixed by any serialization protocol if operations were always executed in their original forms [Sun et al. 1997b]. Although

[2] An example has been given in the previous discussion that identical final results may not be intention preserved. Another example can be found in Section 12, which illustrates that intention-preserved results may not be identical.

the issue of consistency maintenance has attracted a great deal of attention in the areas of both CSCW and distributed computing [Dourish 1996; Ellis and Gibbs 1989; Greenberg and Marwood 1994; Karsenty and Beaudouin-Lafon 1993; Ressel et al. 1996], the *nonserializable* intention violation problem has not been properly distinguished from the divergence problem, and the nature and complications of the intention violation problem (in combination with the problems of divergence and causality violation) have not been well understood, which we believe is one of the major reasons why other existing real-time cooperative editing systems were unable to *correctly* solve all three inconsistency problems under the constraints of a short response time and support for unconstrained cooperative editing in a distributed environment (see the discussion of related work in Section 12). In this article, a novel and integrated approach to solving all three problems in real-time cooperative editing systems is proposed and discussed in detail.

Apart from the *syntactic* inconsistency problems such as these identified above, there are still other *semantic* inconsistency problems, as pointed out by Zhang and Yang [1994], Dourish [1995], and Sun et al. [1996a]. For example, suppose a shared document contains the text

There will be *student* here.

In this text there is a grammatical error, i.e., in the text it should be "a student" or "students" or the like. Assume that one user at site 0 issues an operation O_1 to insert "a" at the starting position of "student"; another user at site 1 issues an operation O_2 to insert "s" at the ending position of "student." Suppose the system preserves the intentions of independent operations (by means of an intention-preserving scheme to be presented in this article). Then, after the execution of these two operations at all sites, the text would be

There will be *a students* here.

This result is syntactically correct (since all sites have the same document contents, and the intended syntactic effects of all operations have been achieved), but semantically incorrect (since there is still a grammatical error in it). In other words, the system is able to ensure the plain *strings* be inserted/deleted at proper positions, but unable to ensure these strings make a correct *English* sentence. This kind of semantic inconsistency problem cannot be resolved by underlying consistency maintenance mechanisms without the intervention of the people in collaboration and additional supporting mechanisms. To the best of our knowledge, none of the existing cooperative editing systems has attempted to maintain semantical consistency automatically. Therefore, the consistency maintenance approach to be presented in this article is not supposed to address the semantic consistency maintenance problem either.

The rest of this article is organized as follows. In Section 2, a novel consistency model with properties of convergence, causality preservation, and intention preservation is defined. Then, the schemes for achieving causality preservation and convergence are presented in Sections 3 and 4, respectively. Next, the basic issues and complications involved in achieving intention preservation are discussed in Section 5, and a generic transformation-based intention-preserving scheme is presented in Section 6. The intention-preserving scheme is integrated with the convergence scheme to provide a solution to both divergence and intention violation in Section 7. In addition, a garbage collection scheme for history buffer management is discussed in Section 8. The design of application-dependent intention-preserving transformation algorithms for text editing is discussed in Section 9. An integrated example is given in Section 10 to illustrate how the whole system works when the generic components and the application-dependent components are integrated. An Internet-based prototype system has been developed to test the feasibility of our approach, and our preliminary experiences with the prototype system are discussed in Section 11. In Section 12, alternative and related approaches are examined and compared to our approach, with special regard to their capability of achieving convergence, causality preservation, and intention preservation. The major contributions and future work of our research are summarized in Section 13.

2. A CONSISTENCY MODEL

In this section, a consistency model [Sun et al. 1996a] is proposed to provide a conceptual framework for devising schemes and algorithms to solve the inconsistency problems identified in the previous section.

Following Lamport [1978], we first define a causal (partial) ordering relation on operations in terms of their generation and execution sequences as follows.

Definition 1 (*Causal Ordering Relation* "→"). Given two operations O_a and O_b, generated at sites i and j, then $O_a \rightarrow O_b$, if and only if (1) $i = j$, and the generation of O_a *happened* before the generation of O_b; (2) $i \neq j$, and the execution of O_a at site j *happened before* the generation of O_b; (3) there exists an operation O_x, such that $O_a \rightarrow O_x$ and $O_x \rightarrow O_b$.

Definition 2 (*Dependent and Independent Operations*). Given any two operations O_a and O_b, (1) O_b is said to be *dependent on* O_a if and only if $O_a \rightarrow O_b$; (2) O_a and O_b are said to be *independent* (or *concurrent*) if and only if neither $O_a \rightarrow O_b$, nor $O_b \rightarrow O_a$, which is expressed as $O_a \parallel O_b$.

For example, the dependency relationship among operations in Figure 1 could be expressed as $O_1 \parallel O_2$, $O_1 \parallel O_4$, $O_3 \parallel O_4$, $O_1 \rightarrow O_3$, $O_2 \rightarrow O_3$, and $O_2 \rightarrow O_4$.

Definition 3 (*Intention of an Operation*). The intention of an operation O is the execution effect which can be achieved by applying O on the document state from which O was generated.

It is worth explicitly pointing out that, throughout this article, the *effect* of an operation refers only to the *syntactic effect* of applying the operation on a document state. For example, the syntactic effect of a text-editing operation is simply inserting/deleting plain characters at specific positions of a text document, which is regarded as a sequence of plain characters. In other words, the term of an operation *effect* does not cover the *semantic meaning* of the operation (which might be inserting/deleting an English sentence) on a document (which might be a poem, a novel, etc). Consequently, the *intention* of an operation, as defined above, refers only to the syntactic effect of applying the operation on a particular document state, i.e., the one from which the operation was originally generated.

Definition 4 (*A Consistency Model*). A cooperative editing system is said to be consistent if it always maintains the following properties:

(1) *Convergence:* When the same set of operations have been executed at all sites, all copies of the shared document are identical.

(2) *Causality preservation:* For any pair of operations O_a and O_b, if $O_a \rightarrow O_b$, then O_a is executed before O_b at all sites.

(3) *Intention preservation:* For any operation O, the effects of executing O at all sites are the same as the intention of O, and the effect of executing O does not change the effects of independent operations.

In essence, the *convergence* property ensures the consistency of the final results *at the end* of a cooperative editing session; the *causality preservation* property ensures the consistency of the execution orders of dependent operations *during* a cooperative editing session; and the *intention preservation* property ensures (1) that the effect of executing an operation at remote sites achieves the same effect as executing this operation at the local site at the time of its generation and (2) that the execution effects of independent operations do not interfere with each other. The consistency model imposes an execution order constraint on dependent operations only, but leaves it open for execution order of independent operations as long as the convergence and intention preservation properties are maintained. The consistency model effectively specifies, on the one hand, what assurance a cooperative editing system promises to its users and, on the other, what properties the underlying consistency maintenance mechanisms must support.

3. ACHIEVING CAUSALITY PRESERVATION

To capture the causal relationship among all operations in the system, a timestamping scheme based on a data structure—State Vector (SV)—can be used [Ellis and Gibbs 1989; Sun et al. 1996a]. Let N be the number of cooperating sites in the system. Assume that sites are identified by integers $0, \ldots, N - 1$. Each site maintains an SV with N components. Initially, $SV[i] := 0$, for all $i \in \{0, \ldots, N - 1\}$. After executing an operation generated at site i, $SV[i] := SV[i] + 1$. An operation is executed at the local site immediately after its generation and then multicast to remote sites with a timestamp of the current value of the local SV.

Definition 5 (*Conditions for Executing Remote Operations*). Let O be an operation generated at site s and timestamped by SV_O. O is *causally ready* for execution at site d ($d \neq s$) with a state vector SV_d only if the following conditions are satisfied:

(1) $SV_O[s] = SV_d[s] + 1$ and
(2) $SV_O[i] \leq SV_d[i]$, for all $i \in \{0, 1, \ldots, N - 1\}$ and $i \neq s$.

The first condition ensures that O must be the next operation in sequence from site s, so no operations originated at site s have been missed by site d. The second condition ensures that all operations originated at other sites and executed at site s before the generation of O must have been executed at site d. Altogether these two conditions ensure that all operations which causally precede O have been executed at site d. It can be shown that if a remote operation is executed only when it satisfies the above two conditions, then all operations will be executed in their causal orders, thus achieving the causality preservation property of the consistency model [Sun et al. 1996b].

4. ACHIEVING CONVERGENCE

The causality-preserving scheme imposes causally ordered execution only for dependent operations and allows an operation to be executed at the local site immediately after its generation (for achieving good responsiveness). This implies that the execution order of independent operations may be different at different sites. A question arises: how to ensure the convergence property in the presence of different execution order of independent operations? To solve this problem, we first define a total ordering relation among operations as follows.

Definition 6 (*Total Ordering Relation* "$\Rightarrow$"). Given two operations O_a and O_b, generated at sites i and j and timestamped by SV_{O_a} and SV_{O_b}, respectively, then $O_a \Rightarrow O_b$, if and only if (1) $sum(SV_{O_a}) < sum(SV_{O_b})$ or (2) $i < j$ when $sum(SV_{O_a}) = sum(SV_{O_b})$, where $sum(SV) = \Sigma_{i=0}^{N-1} SV[i]$.

It can be shown that the total ordering relation "$\Rightarrow$" is consistent with the causal ordering relation "$\rightarrow$" in the sense that if $O_a \rightarrow O_b$, then $O_a \Rightarrow O_b$ [Sun et al. 1996b].

In addition, each site maintains a *history buffer* (*HB*) for saving executed operations at each site. Based on the total ordering relation and the history buffer, the following *undo/do/redo* scheme is defined.

Algorithm 1 (*The Undo/Do/Redo Scheme*). When a new operation O_{new} is causally ready, the following steps are executed:

(1) **Undo** operations in *HB* which totally *follow* O_{new} to restore the document to the state before their execution.

(2) **Do** O_{new}.
(3) **Redo** all operations that were undone from HB.

One assumption made by the above scheme is that all operations in the cooperative editing system are *reversible*. It is required that buffered operations should contain enough information in order to be undone and redone. It should be noted that the *undo* operation involved in the undo/do/redo scheme is only an *internal* operation, rather than an *external* operation initiated from the user interface [Prakash and Knister 1994]. Therefore, the implementation of the undo/do/redo scheme should flush on the user interface only the final result instead of all intermediate results produced in the undo/do/redo process. It can be shown that under the undo/do/redo scheme, when all sites have executed the same set of operations, the editing effect will be the same as if all operations were executed in the total order "$\Rightarrow$" at all sites, thus ensuring the convergence property [Sun et al. 1996b].

5. COMPLICATIONS IN ACHIEVING INTENTION-PRESERVATION

Achieving intention preservation is much harder than achieving convergence and causality preservation. This is because the intention violation problem is not related to the execution order of operations and cannot be resolved by just rescheduling operations, as in the case of achieving convergence and causality preservation. To achieve intention preservation, a causally ready operation has to be transformed before its execution to compensate the changes made to the document state by other executed operations [Sun et al. 1997b].

To transform an operation against another operation, an *inclusion* transformation strategy (see Section 6.1) may be applied, which transforms an operation O_a against another independent operation O_b in such a way that the impact of O_b is effectively included into O_a. To illustrate, consider the two independent operations O_1 and O_2 in Figure 1. When O_2 arrives site 0, it needs to be transformed against O_1 before its execution. Suppose the shared document initially contains "ABCDE," $O_1 = Insert[\text{"12"}, 1]$, and $O_2 = Delete[2, 2]$. Based on the comparison of the position parameters of O_1 and O_2, i.e., $2 > 1$, and the length 2 of the string inserted by O_1, the underlying inclusion transformation algorithm at site 0 is able to correctly derive that O_2 should be transformed into $O'_2 = Delete[2, 4]$. The execution of O'_2 at site 0 will result in the document state "A12BE," which apparently preserves the intentions of both O_2 and O_1. The correctness of this inclusion transformation relies on the fact that both O_1 and O_2 are generated from the same document state, so their position parameters are comparable and can be used to derive a proper adjustment to O_2.

The inclusion transformation could have been always directly applicable if independent operations were all generated out of the same document state, like O_1 and O_2 in Figure 1. Unfortunately, it is possible that some independent operations are generated from different document states in an unconstrained cooperative editing environment. To illustrate, consider the relationship between another pair of independent operations O_1 and O_4 in Figure 1. Since O_4 is generated after the execution of O_2 at site 2, the document state at site 2 at the time of O_4's generation is different from the document state at site 0 at the time of O_1's generation. For independent operations generated from different document states, it is no longer possible to correctly reason about their relative positions by simply comparing their position parameters. For example, suppose the initial document state is "ABCDE," $O_1 = Insert$["12", 1], and $O_2 = Insert$["23", 0]. After the execution of O_2 at site 2, the document state becomes "23ABCDE." Suppose $O_4 = Insert$["45", 2], which is to insert "45" between "23" and "ABCDE." When O_4 arrives at site 0, if the inclusion transformation is directly applied, i.e., O_4 is transformed against O_1 based on the comparison of the position parameter "2" in O_4 and the position parameter "1" in O_1, O_4 would be incorrectly transformed into $O'_4 = Insert$["45", 4]. After the execution of O'_4 at site 0, the document state would become "23A1452BCDE" at site 0, instead of "2345A12BCDE" which is what the document state should be if intentions of O_1, O_2, and O_4 are all preserved. The trouble here is that the position parameter "2" in O_4 and the position parameter "1" in O_1 refer to different document states and hence are not comparable.

How to make two independent operations generated from different document states, such as O_4 and O_1, *effectively* share the same document state so that the inclusion transformation can be applied? Our approach is to apply another *exclusion* transformation (see Section 6.1) to transform O_4 against its causally preceding operation O_2 to produce O'_4 in such a way that O_2's impact on O_4 is excluded. Consequently, O'_4 effectively shares the same document state with O_1 and then can be applied with the inclusion transformation against O_1.

Things would be much easier if the dependency relationship among operations is always as simple as the relationship between O_1 and O_4, which could be expressed by a single dependency expression: $O_1 \parallel (O_2 \rightarrow O_4)$. Consider another pair of independent and incomparable operations O_3 and O_4 in Figure 1. Their dependency relationship is rather irregular and could only be expressed by two dependency expressions: $(O_1 \parallel O_2) \rightarrow O_3$ and $O_2 \rightarrow O_4$. Under this circumstance, it is not obvious how to apply proper transformations to make O_4 effectively share the same document state with O_3 before applying the inclusion transformation to O_4 against O_3 (e.g., at site 1). It is this diverse and irregular dependency relationship among operations that necessitates a sophisticated control algorithm to determine when and how to apply the inclusion/exclusion transformation to which operations against which others.

To cope with the complexities involved in the design (and discussion) of the intention-preserving scheme, we divide the whole scheme into two parts: one is a generic part, which can be applied to different cooperative editing systems and determines which operations need to be transformed against which others and in what order based solely on the causal and total ordering relationships among operations; and the other is an application-dependent part, which relies on semantics of the operations involved, and does the real operation transformation. The generic intention-preserving scheme will be discussed in the following section, and the treatment of concrete inclusion/exclusion transformation algorithms for a text-editing system will be delayed until Section 9.

6. THE GENERIC INTENTION-PRESERVING SCHEME

Since operations may be transformed before their execution, their execution form in *HB* may be different from their original form at the time of their generation. To stress this fact, the list of operations in *HB* is denoted as $HB = [EO_1, EO_2, \ldots, EO_n]$, where EO_i is the execution form of O_i. Moreover, all operations in *HB* are sorted according to their total ordering relation, i.e., $EO_i \Rightarrow EO_{i+1}$. Apparently, the document state at any instance of time can be determined by sequentially executing operations in *HB* in their total order on the initial document state (maybe empty).

To facilitate the following discussion, some special notations are introduced below. Let L, L_1, and L_2 be lists of executed operations. $|L|$ denotes the length of L; L^{-1} denotes the reverse of L; $L[i, j]$, $i < j$, denotes a sublist of L containing the operations from EO_i to EO_j inclusively; $L[i]$, $1 \leq i \leq |L|$, denotes the ith operation in L; $Tail(L)$ denotes all operations in L except the first one, and $L_1 + L_2$ denotes the list of operations which is obtained by concatenating operations in L_1 and L_2. For example, if $L = [EO_1, EO_2, EO_3]$, then $|L| = 3$, $L^{-1} = [EO_3, EO_2, EO_1]$, $L[1, 2] = [EO_1, EO_2]$, $L[1] = EO_1$, and $Tail(L) = [EO_2, EO_3]$. If $L_1 = [EO_1, EO_2]$, and $L_2 = [EO_3]$, then $L_1 + L_2 = [EO_1, EO_2, EO_3]$.

6.1 Pre- and Postconditions for Transformation Functions

Conceptually, an operation O is associated with a *context*, denoted as CT_O, which is the list of operations that need to be executed to bring the document from its initial state to the state on which O is defined. The significance of context is that the effect of an operation can be correctly interpreted only in its own context. When an operation is generated, it is associated with an *original context*, which is the list of operations in *HB* at the site and the time of its generation. For any two operations O_a and O_b, O_b is in the original context of O_a if and only if $O_b \rightarrow O_a$. The context of an operation can be changed by explicitly applying the primitive transformation functions specified in this section. For specifying pre- and postconditions of transformation functions, two context-based relations among operations are defined below.

Definition 7 (*Context-Equivalent Relation* "$\sqcup$"). Given two operations O_a and O_b, associated with contexts CT_{O_a} and CT_{O_b}, respectively, O_a and O_b are *context equivalent*, i.e., $O_a \sqcup O_b$, if and only if $CT_{O_a} = CT_{O_b}$.

Apparently, the context-equivalent relation is transitive, i.e., given operations O_a, O_b, and O_c. If $O_a \sqcup O_b$ and $O_b \sqcup O_c$, then $O_a \sqcup O_c$. In contrast, the independence relation is not transitive, i.e., given operations O_a, O_b, and O_c. If $O_a \parallel O_b$ and $O_b \parallel O_c$, it is *not* guaranteed that $O_a \parallel O_c$. For example, in Figure 1, $O_2 \parallel O_1$ and $O_1 \parallel O_4$, but $O_2 \rightarrow O_4$.

Definition 8 (*Context-Preceding Relation* "$\mapsto$"). Given two operations O_a and O_b, associated with contexts CT_{O_a} and CT_{O_b}, respectively, O_a is *context preceding* O_b, i.e., $O_a \mapsto O_b$, if and only if $CT_{O_b} = CT_{O_a} + [O_a]$.

From the definition of "$\sqcup$" and "$\mapsto$", we know that, given operations O_a, O_b, and O_c, if $O_a \mapsto O_b$ and $O_a \mapsto O_c$, then we can derive that $O_b \sqcup O_c$. It should be noted that the context-preceding relation "$\mapsto$" is different from the dependence (i.e., the causal ordering) relation "$\rightarrow$": the dependence relation is transitive whereas the context-preceding relation is not transitive by definition.

The current list of operations in *HB* at any site determines the current document state at that site and defines a context for executing new operations. Obviously, the execution context at a site keeps changing as new operations are executed at that site. When a causally ready operation has its original context being the same as the current execution context at the destination site, it can be executed as it is; hence, the document state determined by the current execution context is the same as the document state determined by the operation's original context and since the operation's intention is automatically guaranteed. However, when a causally ready operation has its original context being different from the current execution context (due to preceding executions of independent operations), this operation needs to be transformed before its execution in the current execution context in order to preserve its intention. The intention-preserving scheme uses the following two primitive transformation functions to include/exclude one operation into/from the context of another operation to produce a new operation.

To include operation O_b into the context of O_a, the *inclusion transformation* function $IT(O_a, O_b)$ is called to produce O'_a, as specified below.

Specification 1 ($IT(O_a, O_b) : O'_a$).

(1) Precondition for input parameters: $O_a \sqcup O_b$.
(2) Postcondition for output: $O_b \mapsto O'_a$, where O'_a's execution effect in the context of $CT_{O'_a}$ is the same as O_a's execution effect in the context of CT_{O_a}.

To exclude operation O_b from the context of O_a, the *exclusion transformation* function $ET(O_a, O_b)$ is called to produce O'_a, as specified below.

Specification 2 ($ET(O_a, O_b) : O'_a$).

(1) Precondition for input parameters: $O_b \mapsto O_a$.
(2) Postcondition for output: $O_b \sqcup O'_a$, where O'_a's execution effect in the context of $CT_{O'_a}$ is the same as O_a's execution effect in the context of CT_{O_a}.

In addition to the postconditions, the two primitive transformation functions must meet the *reversibility* requirement as defined below.

Definition 9 (*Reversibility Requirement*). Given two operations O_a and O_b,

(1) if $O_a \sqcup O_b$, then $O_a = ET(IT(O_a, O_b), O_b)$;
(2) if $O_b \mapsto O_a$, then $O_a = IT(ET(O_a, O_b), O_b)$.

To simplify the expression of applying the two primitive transformation functions repeatedly to a list of operations, two additional utility functions $LIT(\ )$ and $LET(\ )$ are defined below.

To include a list of operations OL into the context of operation O, the following function $LIT(O, OL)$ can be called.

```
FUNCTION 1. $LIT(O, OL)$
{
  if $OL = [\ ]$ $O' := O$;
  else $O' := LIT(IT(O, OL[1]), Tail(OL))$;
  return $O'$;
}
```

The preconditions for input parameters of $LIT(O, OL)$ are (1) $O \sqcup OL[1]$ and (2) for any two consecutive operations $OL[i]$ and $OL[i + 1]$ in OL, $OL[i] \mapsto OL[i + 1]$.

To exclude a list of operations OL from the context of operation O, the following function $LET(O, OL)$ can be called.

```
FUNCTION 2. $LET(O, OL)$
{
  if $OL = [\ ]$ $O' := O$;
  else $O' := LET(ET(O, OL[1]), Tail(OL))$;
  return $O'$;
}
```

The preconditions for input parameters of $LET(O, OL)$ are (1) $OL[1] \mapsto O$ and (2) for any two consecutive operations $OL[i]$ and $OL[i + 1]$ in OL, $OL[i + 1] \mapsto OL[i]$.

Preconditions are required by these transformation functions to facilitate the *correct* derivation of the adjustment to one operation's parameters according to the impact of the other operation. How to ensure the preconditions of the input operations is one of the main tasks of the higher-level control algorithm to be discussed in Section 6.2. Given the input operations satisfying the preconditions, how to produce the output operation which satisfies the postconditions and meets the reversibility requirement is the responsibility of the lower-level inclusion and exclusion transformation functions. The concrete definitions of $IT(O_a, O_b)$ and $ET(O_a, O_b)$ depend on the semantics of the input operations and hence is application dependent. The inclusion and exclusion transformation functions for a text-editing system with two primitive operations—*Insert* and *Delete*—will be discussed in Section 9.

6.2 The Generic Operation Transformation Control Algorithm

Based on the concept of context and the specifications of inclusion and exclusion transformation functions, the *Generic Operation Transformation* (*GOT*) control algorithm has been devised and is discussed in this section.

Let O_{new} be a new causally ready operation, associated with its original context $CT_{O_{new}}$, and $HB = [EO_1, EO_2, \ldots, EO_m]$. Assume that

(1) $EO_1 \Rightarrow EO_2 \Rightarrow \ldots \Rightarrow EO_m \Rightarrow O_{new}$ and
(2) $EO_1 \mapsto EO_2 \mapsto \ldots \mapsto EO_m$.

The objective of the GOT control algorithm is to determine the execution form of O_{new}, denoted as EO_{new}, such that

(1) $EO_m \mapsto EO_{new}$ and
(2) the effect of executing EO_{new} in the context of $CT_{EO_{new}}$ $(= HB)$ achieves the same effect as executing O_{new} in the context of $CT_{O_{new}}$.

In the following, an exhaustive case analysis is conducted to derive the components of the GOT control algorithm. To begin with, consider the simplest case: all operations in HB are causally preceding O_{new}. This case could occur (1) if O_{new} is a newly generated local operation or (2) O_{new} is a remote operation, and its original context contains the same list of operations as the local HB at the time when O_{new} becomes causally ready for execution. In this case, it must be that $CT_{O_{new}} = HB$. Therefore, $EO_m \mapsto O_{new}$, and O_{new} can be executed in its original form without transformation, i.e., $EO_{new} := O_{new}$.

If, however, there exists any operation EO_k in HB, such that $EO_k \parallel O_{new}$, then operation EO_k must not be in the context of O_{new}, so $CT_{O_{new}} \neq HB$. Therefore, O_{new} needs to be transformed in order to include the impact of EO_k and other independent operations in HB. Suppose $HB = [EO_1, \ldots, EO_k, \ldots, EO_m]$, where EO_k is the oldest operation in HB which is independent of O_{new}. EO_k can be identified by scanning operations in HB from left to right until an operation independent of O_{new} is met.[3] Obviously, $HB[1, k - 1]$ must prefix the original context of O_{new}. The technical challenge here is how to ensure that the preconditions required by the primitive transformation functions are always met in the process of including independent operations in the range of $HB[k, m]$ into the context of O_{new}.

In case that all operations in the range of $HB[k, m]$ are independent of O_{new}, it must be that $CT_{O_{new}} = HB[1, k - 1]$. Under this special circumstance, we have $O_{new} \sqcup EO_k$ and $EO_k \mapsto EO_{k+1} \mapsto \ldots \mapsto EO_m$, so we can directly apply the list inclusion transformation function to produce EO_{new}, i.e., $EO_{new} := LIT(O_{new}, HB[k, m])$.

[3] If no such EO_k is found in HB, then the situation is the simplest case, as discussed before.

The complication comes when there is a mixture of independent and dependent operations in the range of $HB[k, m]$. Let $EOL = [EO_{c_1}, \ldots, EO_{c_r}]$ be the list of operations in the range of $HB[k + 1, m]$, which are causally preceding O_{new}. Under this circumstance, it must be that $CT_{O_{new}} = HB[1, k - 1] + EOL'$, where $EOL' = [EO'_{c_1}, \ldots, EO'_{c_r}]$, and EO'_{c_i} is the corresponding form of EO_{c_i} at the time of O_{new}'s generation. Apparently, $CT_{EO'_{c_i}} \neq CT_{EO_{c_i}}$ because there exists at least one operation EO_k, which is in $CT_{EO_{c_i}}$ but not in $CT_{EO'_{c_i}}$. The dilemma we are facing here is that, on one hand, we cannot directly apply the list inclusion transformation function to include all operations in $HB[k, m]$ into $CT_{O_{new}}$, because $CT_{O_{new}}$ contains a list of dependent operations in EOL', and hence O_{new} is not context equivalent with EO_k. On the other hand, we cannot directly apply the list exclusion transformation function to exclude all dependent operations in EOL (in reverse order) from $CT_{O_{new}}$ to make O_{new} context equivalent with EO_k either, because (1) it is not the case that $EO_{c_r} \mapsto O_{new}$ and (2) there is no guarantee that $EO_{c_r} \mapsto EO_{c_{i+1}}$, for $1 \leq i < r$, due to the mixture of independent and dependent operations in the range of $HB[k, m]$.

To make O_{new} context equivalent with EO_k, what we should do is to apply the exclusion transformation function to exclude the operations in EOL', instead of EOL, from the context of O_{new}, because (1) $EO'_{c_r} \mapsto EO_{new}$ and (2) it is assured that $EO'_{c_i} \mapsto EO'_{c_{i+1}}$ for $1 \leq i < r$. However, we have only EOL, not EOL', available in HB. Then, the problem becomes how to obtain each operation EO'_{c_i} in EOL' from the corresponding operation EO_{c_i} in EOL?

For the first operation in EOL, i.e., EO_{c_1}, we have observed that $CT_{EO_{c_1}} = HB[1, k - 1] + HB[k, c_1 - 1]$, but $CT_{EO'_{c_1}} = HB[1, k - 1]$. Therefore, EO'_{c_1} can be obtained by excluding operations in the range of $HB[k, c_1 - 1]$ (in reverse order) from the context of EO_{c_1}, i.e., $EO'_{c_1} := LET(EO_{c_1}, HB[k, c_1 - 1]^{-1})$.

For the second operation in EOL, i.e., EO_{c_2}, we observed that $CT_{EO'_{c_2}} = HB[1, k - 1] + HB[k, c_2 - 1]$, but $CT_{EO'_{c_2}} = HB[1, k - 1] + [EO'_{c_1}]$. Therefore, EO'_{c_2} can be obtained by first excluding operations in the range of $HB[k, c_2 - 1]$ (in the reverse order) from the context of EO_{c_2}, and then including EO'_{c_1} into the intermediate result. So, the following two steps need to be executed:

(1) $TO := LET(EO_{c_2}, HB[k, c_2 - 1]^{-1})$;[4]
(2) $EO'_{c_2} := IT(TO, EO'_{c_1})$.

Generally, for the ith operation in EOL, i.e., EO_{c_i}, $2 \leq i \leq r$, the following two steps need to be executed:

(1) $TO := LET(EO_{c_i}, HB[k, c_i - 1]^{-1})$;
(2) $EO'_{c_i} := LIT(TO, [EO'_{c_1}, \ldots, EO'_{c_{i-1}}])$.

[4] TO stands for Temporary Operation, which is a notation used for representing an intermediate result during transformation.

Once EOL' has been obtained from EOL, we can now apply the list exclusion transformation function to exclude all operations in EOL' (in reverse order) from $CT_{O_{new}}$ to produce an O'_{new}, which is context equivalent with EO_k, and then apply the list inclusion transformation function to include all the operations in $HB[k, m]$ into $CT_{O_{new}}$, i.e.,

(1) $O'_{new} := LET(O_{new}, EOL'^{-1})$;
(2) $EO_{new} := LIT(O'_{new}, HB[k, m])$.

Based on the above analysis and reasoning, the GOT control algorithm can be specified below.

Algorithm 2 (*The GOT Control Algorithm*). Given a new causally ready operation O_{new}, and $HB = [EO_1, EO_2, \ldots, EO_m]$, the execution form of O_{new}, denoted as EO_{new}, is obtained as follows:

```
(1) Scan $HB$ from left to right to find the first operation $EO_k$ such that
    $EO_k \parallel O_{new}$. If no such an operation $EO_k$ is found, then **return** $EO_{new}$
    $:= O_{new}$.
(2) Otherwise, scan $HB[k + 1, m]$ to find all operations which are
    causally preceding $O_{new}$. If no single such operation is found, then
    **return** $EO_{new} := LIT(O_{new}, HB[k, m])$.
(3) Otherwise, let $EOL = [EO_{c_1}, \ldots, EO_{c_r}]$ be the list of operations in
    $HB[k + 1, m]$ which are causally preceding $O_{new}$. The following steps
    are executed:
    (1) Get $EOL' = [EO'_{c_1}, \ldots, EO'_{c_r}]$ as follows:
        —$EO'_{c_1} := LET(EO_{c_1}, HB[k, c_1 - 1]^{-1})$.
        —For $2 \leq i \leq r$,
            (a) $TO := LET(EO_{c_i}, HB[k, c_i - 1]^{-1})$;
            (b) $EO'_{c_i} := LIT(TO, [EO'_{c_1}, \ldots, EO'_{c_{i-1}}])$.
    (2) $O'_{new} := LET(O_{new}, EOL'^{-1})$.
    (3) **return** $EO_{new} := LIT(O'_{new}, HB[k, m])$.
```

According to the above analysis and specification of the GOT control algorithm, we know that the preconditions required by the transformation functions are always guaranteed by the GOT control algorithm. Therefore, if the postconditions are always ensured by the transformation functions, then EO_{new} obtained by the GOT control algorithm will have the following property: the execution of EO_{new} in the context of $CT_{EO_{new}}$ $(= HB)$ will achieve the same effect as the execution of O_{new} in the context of $CT_{O_{new}}$. Since $CT_{O_{new}}$ is the original context of O_{new}, the execution of EO_{new} in the context of HB will preserve the intention of O_{new}.

It is worth pointing out that the GOT control algorithm works solely on a linear data structure HB, and no additional data structures need to be maintained for deriving and checking context-based relationships among operations. In fact, the context-based relationship among operations required by the transformation functions is *implicitly* enforced, instead of being *explicitly* checked, by the GOT control algorithm. However, the dependence relationship among operations needs to be explicitly checked by the GOT control algorithm. To check whether an operation is dependent/independent of another operation, the following well-known property of the state vector timestamping has been used [Raynal and Singhal 1996]: given two operations O_a and O_b, generated at sites i and j and timestamped by SV_{O_a} and SV_{O_b}, respectively, $O_a \rightarrow O_b$ if and only if $SV_{O_a}[i] \leq SV_{O_b}[i]$.

7. INTEGRATING THE GOT CONTROL ALGORITHM WITH THE UNDO/DO/REDO SCHEME

In this section, we discuss how to achieve both intention preservation and convergence by integrating the GOT control algorithm with the *undo/do/redo* scheme.

Let O_{new} be a new causally ready operation, and let $HB = [EO_1, \ldots, EO_m, \ldots, EO_n]$, where EO_1 is the oldest operation in HB; EO_m is the youngest operation which is totally preceding O_{new}; and EO_n is the youngest operation in HB.

The integrated scheme starts by undoing all executed operations in the range of $HB[m + 1, n]$ to restore the document to the state before their execution. Then, O_{new} is transformed into EO_{new} by the GOT control algorithm and executed. Finally, all undone operations are transformed and redone one by one to take into account the impact of EO_{new}. The strategy for transforming and redoing undone operations is discussed below.

For the first operation in the range of $HB[m + 1, n]$, i.e., EO_{m+1}, we observed that $CT_{EO_{m+1}} = HB[1, m]$ and $CT_{EO_{new}} = HB[1, m]$, so $EO_{m+1} \sqcup EO_{new}$. Therefore, the new execution form of EO_{m+1}, denoted as EO'_{m+1}, can be obtained as follows: $EO'_{m+1} := IT(EO_{m+1}, EO_{new})$.

For the second operation in $HB[m + 1, n]$, i.e., EO_{m+2}, we observed that $CT_{EO_{m+2}} = HB[1, m] + [EO_{m+1}]$, but $CT_{EO_{new}} = HB[1, m]$, so EO_{m+2} is not context equivalent with EO_{new}. Therefore, we need to first exclude EO_{m+1} from the context of EO_{m+2}, and then include EO_{new} and EO'_{m+1} into the context of the intermediate result. So two steps are needed:

(1) $TO := ET(EO_{m+2}, EO_{m+1})$;
(2) $EO'_{m+2} := LIT(TO, [EO_{new}, EO'_{m+1}])$.

Generally, for the ith operation in $HB[m + 1, n]$, i.e., EO_{m+i}, $2 \leq i \leq (n - m)$, the following two steps are executed:

(1) $TO := LET(EO_{m+1}, HB[m + 1, m + i - 1]^{-1})$;
(2) $EO'_{m+1} := LIT(TO, [EO_{new}, EO'_{m+1}, \ldots, EO'_{m+i-1}])$.

It can be shown that EO'_{m+i} obtained in the above way will have the following property: the execution of EO'_{m+i} in the context of $CT_{EO'_{m+i}}$ (= HB) will achieve the same effect as the execution of EO_{m+i} in the context of $CT_{EO_{m+i}}$.

Based on the above analysis, the integrated scheme is defined below.

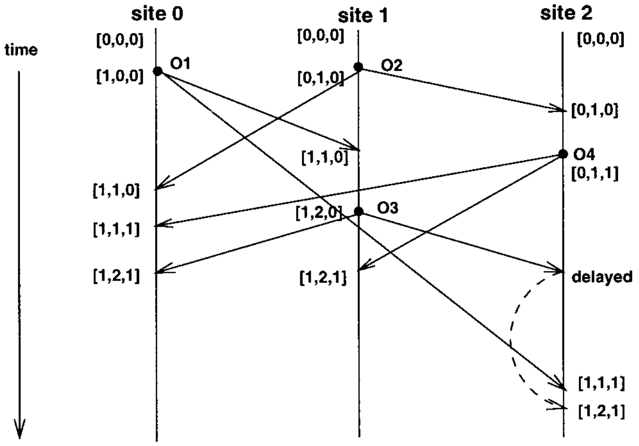


Fig. 2. A scenario of a real-time cooperative editing session augmented with both the causality-preserving scheme and the undo/transform-do/transform-redo scheme.

Algorithm 3 (*The Undo/Transform-Do/Transform-Redo Scheme*). Given a new causally ready operation O_{new}, and $HB = [EO_1, \ldots, EO_m, \ldots, EO_n]$, the following steps are executed:

(1) **Undo** operations in HB from right to left until an operation EO_m is found such that $EO_m \Rightarrow O_{new}$.
(2) **Transform** O_{new} into EO_{new} by applying the GOT control algorithm, and **Do** EO_{new}.
(3) **Transform** each operation EO_{m+i} in $HB[m + 1, n]$ into the new execution form EO'_{m+i} as follows:

```
—$EO'_{m+1} := IT(EO_{m+1}, EO_{new})$.
—For $2 \leq i \leq (n - m)$,
    (1) $TO := LET(EO_{m+i}, HB[m + 1, m + i - 1]^{-1})$;
    (2) $EO'_{m+i} := LIT(TO, [EO_{new}, EO'_{m+1}, \ldots, EO'_{m+i-1}])$.
```

Redo $EO'_{m+1}, EO'_{m+2}, \ldots, EO'_n$, sequentially.

After the execution of the above steps, the contents of the history buffer becomes

$$HB = [EO_1, \ldots, EO_m, EO_{new}, EO'_{m+1}, \ldots, EO'_n].$$

It can be shown that when all sites have executed the same set of operations, their *HBs* must have the same list of executed operations in the same order and in the same format; hence convergence is ensured. Moreover, each executed operation in *HB* is obtained by either the GOT control algorithm or by step (3) of the undo/transform-do/transform-redo scheme; hence intention preservation is ensured.

Example 1. To illustrate how the undo/transform-do/transform-redo scheme works, consider the scenario shown in Figure 2, with each site augmented with the causality-preserving scheme and the undo/transform-do/transform-redo scheme. The values of state vectors after executing an operation at each site are indicated explicitly as well.

The sequence of events happening at each site is explained below.

—*Site 0:*

(1) When O_1 is generated, it is executed as it is (since it is local), i.e., $EO_1 := O_1$, and $HB = [EO_1]$.

(2) When O_2 arrives, it needs to be applied with the inclusion transformation against EO_1 before its execution, i.e., $EO_2 := IT(O_2, EO_1)$, since $EO_1 \parallel O_2$ and $EO_1 \sqcup O_2$. After the execution of EO_2, $HB = [EO_1, EO_2]$.

(3) When O_4 arrives, it needs to be transformed before its execution, since $EO_1 \parallel O_4$. However, O_4 is not context equivalent with EO_1, since the context of O_4 contains O_2, but the context of EO_1 does not. Therefore, O_4 needs to be first applied with the exclusion transformation against O_2, which can be obtained by applying the exclusion transformation on EO_2 against EO_1. Therefore, $EO_4 := LIT(ET(O_4, ET(EO_2, EO_1)), [EO_1, EO_2])$. After the execution of EO_4, $HB = [EO_1, EO_2, EO_4]$.

(4) When O_3 arrives, it needs to be applied with the inclusion transformation against EO_4, i.e., $EO_3 := IT(O_3, EO_4)$, since $O_3 \parallel EO_4$ and $O_3 \sqcup EO_4$. Finally, $HB = [EO_1, EO_2, EO_4, EO_3]$.

—*Site 1:*

(1) When O_2 is generated, it is executed as it is, i.e., $EO_2 := O_2$, and $HB = [EO_2]$.

(2) When O_1 arrives, EO_2 needs first to be undone, since $O_1 \Rightarrow EO_2$. Secondly, O_1 is executed as it is; $EO_1 := O_1$. Thirdly, the new execution form of O_2 becomes: $EO_2 := IT(O_2, EO_1)$, since $EO_1 \sqcup O_2$. Finally, $HB = [EO_1, EO_2]$.

(3) When O_3 is generated, it is executed as it is, i.e., $EO_3 := O_3$, and $HB = [EO_1, EO_2, EO_3]$.

(4) When O_4 arrives, EO_3 needs first to be undone, since $O_4 \Rightarrow EO_3$. Secondly, O_4 needs to be transformed to get EO_4 in the same way as step (**3**) at site 0. Thirdly, the new execution form of O_3 becomes: $EO'_3 := IT(EO_3, EO_4)$,[5] since $EO_3 \sqcup EO_4$ due to the fact that $EO_2 \mapsto O_3$ and $EO_2 \mapsto EO_4$. Finally, $HB = [EO_1, EO_2, EO_4, EO'_3]$.

—*Site 2:*

(1) When O_2 arrives, it is executed as it is, i.e., $EO_2 := O_2$, and $HB = [EO_2]$.

(2) When O_4 is generated, it is executed as it is, i.e., $EO_4 := O_4$, and $HB = [EO_1, EO_4]$.

(3) When O_3 arrives it is suspended, since $O_1 \rightarrow O_3$, but O_1 has not yet

[5] EO'_3 here equals EO_3 at site 0, since they both are equal to $IT(O_3, EO_4)$.

executed at site 2, which is detected by comparing the values of the state vector (i.e., [1, 2, 0]) associated with O_3 and the local state vector (i.e., [0, 1, 1]), according to the causality-preserving scheme.

(3) When O_1 arrives, both EO_4 $(= O_4)$ and EO_2 $(= O_2)$ need first to be undone, since $O_1 \Rightarrow O_2 \Rightarrow O_4$. Secondly, O_1 is executed as it is, i.e., $EO_1 := O_1$. Thirdly, the new execution form of O_2 becomes $EO_2 := IT(O_2, EO_1)$, since $EO_1 \sqcup O_2$. To obtain the new execution form of O_4, O_4 needs first to be applied with the exclusion transformation against O_2 to become context equivalent with EO_1 and then applied with the inclusion transformation against EO_1 and EO_2 in sequence, i.e., $EO_4 = LIT(ET(O_4, O_2), [EO_1, EO_2])$. Finally, $HB = [EO_1, EO_2, EO_4]$.

(4) After the execution of O_1, the suspended O_3 can be executed, and its execution form is $EO_3 := IT(O_3, EO_4)$, since $O_3 \sqcup EO_4$ due to the fact that $EO_2 \mapsto O_3$ and $EO_2 \mapsto EO_4$. Finally, $HB = [EO_1, EO_2, EO_4, EO_3]$.

From this example, we can see that convergence and intention preservation are ensured by the fact that all sites have effectively executed the same sequence of properly transformed operations: EO_1, EO_2, EO_4, and EO_3, where $EO_1 = O_1$, $EO_2 = IT(O_2, EO_1)$, $EO_4 = LIT(ET(O_4, O_2), [EO_1, EO_2])$, and $EO_3 = IT(O_3, EO_4)$, and causality is preserved by suspending O_3 until the execution of O_1 at site 2.

8. THE GARBAGE COLLECTION SCHEME

In the preceding discussions, there is an implicit assumption: each site maintains an *HB* with an infinite storage capacity. In practice, memory buffers always have limited sizes. Unless something is done to prevent it, *HB* will soon fill up with executed operations. Since the purpose of saving executed operations in *HB* is to support the *undo/transform-do/transform-redo* scheme, an operation needs not to be kept in *HB* if it is no longer involved in the *undo/transform-do/transform-redo* scheme. In this section, we will discuss the concept of garbage and devise the techniques for garbage collection.

In the design of the *undo/transform-do/transform-redo* scheme, we observed that given a new operation O_{new}, if we can identify an operation O_k in *HB* which is the oldest operation independent of O_{new}, then the executed operations in the range of $HB[1, k - 1]$ are surely not involved in the *undo/transform-do/transform-redo* scheme with respect to O_{new}, because operations in the range of $HB[1, k - 1]$ are all causally preceding O_{new} and must have been included in the context of O_{new}.

It follows directly from this observation that if operations in the range of $HB[1, j]$ are surely causally preceding *all* forthcoming new operations, then they will definitely not be involved in the *undo/transform-do/transform-redo* scheme any more and can be collected as garbage. Based on the above analysis, we can envisage a garbage collection scheme, which scans *HB* from the left to right to check if the current operation under inspection is surely causally preceding all forthcoming new operations. If yes, then delete it from HB; otherwise stop scanning.

The above envisaged scheme relies on the assessment of whether an executed operation in HB will surely be causally preceding all forthcoming new operations. To discover the information for this assessment, each site maintains a State Vector Table (SVT) with N state vectors, one per site. Let SVT_k be the state vector table at site k. Initially, $SVT_k[i][j] := 0$ for all i, $j \in \{0, \ldots, N-1\}$. After executing an operation O from a remote site r, timestamped by SV_O, the rth vector of SVT_k is updated as follows: $SVT_k[r][i] := SV_O[i]$, for all $i \in \{0, \ldots, N-1\}$. Obviously, $SVT_k[i]$ at site k represents the view (or knowledge) of site k about the state at site i, for any $i \in \{0, 1, \ldots, N-1\}$. In particular, $SVT_k[k] = SV_k$, i.e., the kth vector of SVT_k is the local state vector.

If one site happens to be silent for an unusually long period of time, other sites will not know what its state is. Therefore, it is required for a site to broadcast a short *state message* containing its state vector when it has not generated an operation for a certain period of time and/or after executing a certain number of remote operations. Upon receiving a state message from a remote site r, site k simply updates its rth state vector in SVT_k with the piggybacked state vector.

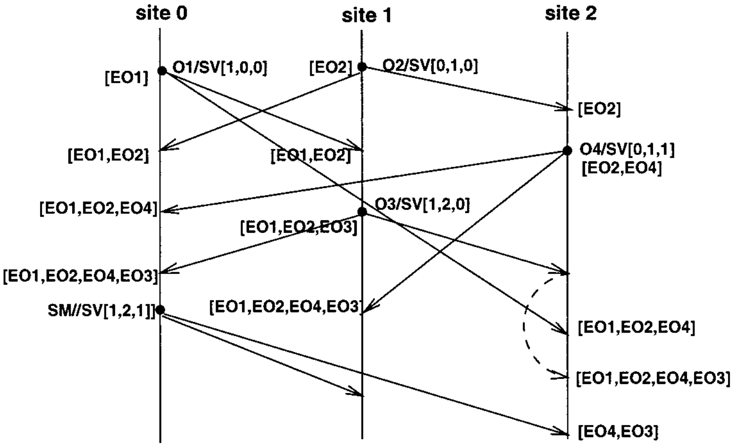
In addition, each site maintains a Minimum State Vector (MSV), with N elements, one per site. Let MSV_k be the minimum state vector at site k. Initially, $MSV_k[i] := 0$ for all $i \in \{0, \ldots, N-1\}$. After executing an operation and updating the SVT_k, or processing a state message from any remote site, MSV_k is updated as follows: $MSV_k[i] := min(SVT_k[0][i], \ldots, SVT_k[N-1][i])$, for all $i \in \{0, 1, \ldots, N-1\}$.

It can be shown that if the value of the ith element in MSV_k becomes m, i.e., $MSV_k[i] = m$, then the first m operations generated at site i must have been executed at all sites. Under the assumption that messages sent from one site to another are received in the same order as they are sent, it must be that all operations generated at each remote site before the execution of the mth operation from site i have been executed at site k. Therefore, the first m operations from site i will certainly precede all forthcoming operations from any site. For any executed operation EO in HB, generated at site i and timestamped by SV_{EO}, if $SV_{EO}[i] \leq m$ $(= MSV_k[i])$, then EO must be one of the first m operations generated at site i and hence will certainly causally precede all forthcoming operations to site k. Based on the above discussion, a garbage collection procedure is defined as follows:

Algorithm 4 (*The Garbage Collection Procedure*). Scan HB from left to right. Let EO be the current operation under inspection. Suppose that EO was generated at site i and timestamped by SV_{EO}.

(1) If $SV_{EO}[i] \leq MSV_k[i]$, then EO is removed from HB and continue scanning.

(2) Otherwise stop scanning and return.



SVT:	[1,2,1]	[1,2,1]	[1,2,1]
	[1,2,0]	[1,2,1]	[1,2,0]
	[0,1,1]	[0,1,1]	[1,2,1]
MSV	[0,1,0]	[0,1,1]	[1,2,0]

Fig. 3. A scenario of garbage collection in a real-time cooperative editing system.

The garbage collection procedure can be invoked periodically, after processing each remote operation/message, or when the number of buffered operations in *HB* goes beyond a preset threshold value.

Example 2. To illustrate the garbage collection scheme, consider the scenario in Figure 3, which is obtained by adding an extra *state message* (SM) at site 0 to the scenario in Figure 2. The timestamp piggybacked in each message, the contents of *HB* at each site after processing an operation, and the contents of the *SVT* and *MSV* at each site after processing all messages in this scenario are illustrated.

—At site 0, after processing all operations, the local MSV is [0, 1, 0]. The first executed operation in *HB* is EO_1 and $SV_{EO_1}[0] = 1$, which is greater than $MSV[0]$ $(= 0)$; so the garbage collection procedure terminates itself without removing any operation from *HB*. It should be noted that the second operation in *HB* is EO_2 and $SV_{EO_2}[1] = 1$, which is equal to $MSV[1]$ $(= 1)$, but EO_2 cannot be collected as garbage, since EO_2 may still be involved in the *undo/transform-do/transform-redo* scheme with respect to forthcoming operations which may be independent of EO_1.[6]

—At site 1, after processing all operations and the SM, the local MSV equals [0, 1, 1]; no operation in *HB* could be removed for the same reason as in site 0.

[6] Although all operations which are independent of EO_1 have already arrived in this scenario, site 0 does not have sufficient knowledge about the state of other sites at this stage and cannot collect EO_1 as garbage.

—At site 2, after processing the delayed O_3 and the SM, the local MSV equals [1, 2, 0]. Since the first executed operation in HB is EO_1 and $SV_{EO_1}[0] = 1$, which is equal to $MSV[0]$ (= 1), EO_1 is removed as garbage from HB. The second executed operation in HB is EO_2 and $SV_{EO_2}[1] = 1$, which is less than $MSV[1]$ (= 2), so EO_2 is removed as garbage from HB as well.

From this example, we can see that each site collects garbage from its local HB based on its own knowledge about the states of other sites in a completely distributed fashion.

9. APPLICATION-DEPENDENT TRANSFORMATION ALGORITHMS

In this section, a pair of reversible inclusion and exclusion transformation algorithms for stringwise operations [Sun et al. 1998] are presented to demonstrate how the application-dependent part of the intention-preserving scheme can be designed.

9.1 Application Environment

9.1.1 *A Text Document Data Model.* A text document (with no formatting) is modeled by a sequence of characters, referred to (or addressed) from 0 to the end of the document. Each primitive editing operation on the document state has one *position* parameter, which determines the *absolute* position in the document at which the operation is to be performed.

It should be pointed out that the above text document data model is just a conceptual view of the text document presented to the user, and it does not dictate the actual data structure which is used to implement the document state. This conceptual data model could be implemented in various different internal data structures, such as a single array of characters, the linked-list structures, the buffer-gap structure, and virtual-memory blocks [Valdes 1993].

9.1.2 *Two Primitive Editing Operations.* It is assumed that the document state can only be changed by executing the following two primitive editing operations:

(1) $Insert[S, P]$: insert string S at position P.
(2) $Delete[N, P]$: delete N characters started from position P.

That means, a new character C_n is added to the document state if and only if C_n is in the string of an executed *Insert* operation; and an old character C_o is deleted from the document state if and only if C_o is in the deleting range of an executed *Delete* operation. It has been shown that practical text-editing systems, such as *vi* and *Emacs*, can be implemented on top of these two primitives [Knister and Prakash 1993; Ressel et al. 1996; Valdes 1993].

9.1.3 *Criteria for Verifying Intention-Preserved Effects.* According to the postconditions and the reversibility requirement of the inclusion and exclusion transformation functions, the execution of a transformed operation in the new context should achieve the same editing effect as the execution of the corresponding operation before transformation in the old context. The concrete criteria for verifying this postcondition for *Insert* and *Delete* are given below.

Specification 3 (*Verification Criteria*). Given an operation O associated with context CT_O, let O' be an operation obtained by applying an inclusion/exclusion transformation to O against another operation, and $CT_{O'}$ be the context associated with O'. The execution of O' in the context of $CT_{O'}$ achieves the same effect as the execution of O in the context of CT_O if the following criteria are satisfied:

(1) *In case that* $O = Insert[S, P]$ *and* $O' = Insert[S', P']$: It must be that (1) $S' = S$, which means the string S in its entirely appears in the document after the execution of O' in the context of $CT_{O'}$; (2) for any $O_x = Insert[S_x, P_x]$ included in $CT_{O'}$, if O_x is independent of O, then S' does not appear in the middle of S_x; (3) for any character C which exists in both the document state determined by CT_O and the document state determined by $CT_{O'}$, if C is at the left/right side of position P in the document state determined by CT_O, then C must be at the left/right side of position P' in the document state determined by $CT_{O'}$.

(2) *In case that* $O = Delete[N, P]$, *and* $O' = Delete[N', P']$: Let $R_O = [C_P, C_{P+1}, \ldots, C_{P+N}]$ be the list of characters within the deleting range of O in the document state determined by CT_O, and $R_{O'} = [C_{P'}, C_{P'+1}, \ldots, C_{P'+N'}]$ be the list of characters within the deleting range of O' in the document state determined by $CT_{O'}$. It must be that (1) all characters in R_O disappear from the document after the execution of O' in the context of $CT_{O'}$; and (2) for any $O_x = Insert[S_x, P_x]$ included in $CT_{O'}$, if O_x is independent of O, then $R_{O'}$ does not include any character in S_x.

When the above criteria are satisfied, the execution effects of independent *Insert/Delete* operations will not interfere with each other in the following sense: an *Insert* operation may never insert a string into the middle of another string inserted by an independent operation; and a *Delete* operation may never delete characters inserted by independent operations. Moreover, if multiple independent *Delete* operations have overlapping deleting ranges, the combined deleting effect will be the union of individual deleting ranges, i.e., the overlapped ranges are merged (an effect which could not be achieved by serializing independent operations in any order). If multiple independent *Insert* operations insert strings at the same position, all strings will appear in the document state as if they were inserted in some total order, which could be enforced by a high-level control scheme (e.g., the undo/do/redo scheme). In case that one of these strings, e.g., “AB,” is a prefix of another string, e.g., “ABCD,” there exist two possible combined inserting effects: (1) merged effect: “ABCD”; or (1) nonmerged effect: “ABABCD” (provided that “ABCD” was inserted before “AB” is inserted).

Both merged and nonmerged effects satisfy the verification criteria, but the nonmerged effect is chosen by our transformation functions (see Section 9.2) for its simplicity.

9.1.4 *Notations.* To facilitate the description of transformation functions, the following notations are introduced: (1) $P(O)$: the position parameter of operation O; (2) $L(O)$: the length of operation O; for *Insert*, it is the length of the string to be inserted; for *Delete*, it is the number of characters to be deleted; (3) $S(O)$: the string of the *Insert* operation.

As will be seen in the following sections, in a number of exceptional cases, some information in the parameters of input operations may get lost during transformation, and the information contained in the parameters of the output operation may not be adequate to ensure the reversibility of the inclusion and exclusion transformation functions. To cope with these exceptional cases, each operation is assumed to have, in addition to the explicit parameters, an internal data structure, which maintains whatever information necessary for ensuring reversibility. To hide away the details of the internal data structure from the description of transformation algorithms, a number of utility routines will be introduced which access and utilize information in this internal data structure (without giving the implementation details for clarity).

9.2 Inclusion Transformation

To apply the inclusion transformation to O_a against O_b, $IT(O_a, O_b)$ is invoked, which in turn, according to the operation types (*Insert/Delete*) of O_a and O_b, calls one of the four subfunctions to do the real transformation. The four subfunctions are defined in Figure 4.

9.2.1 *Basic Transformation Strategy.* A precondition for O_a and O_b, i.e., $O_a \sqcup O_b$, is required to ensure that the relation of the operation ranges of O_a and O_b can be *correctly* determined by simply comparing their parameters, so that the following basic inclusion transformation strategy can be applied: (1) compare the parameters of O_a and O_b to determine the relation of their operation ranges, (2) assume that O_b has been executed to find O_a's operation range in the new document state, and (3) adjust O_a's parameters to make O'_a, according to the comparison result in (1), the impact of the assumed execution in (2), and the verification criteria.

For example, to apply the inclusion transformation to an *Insert* operation O_a against a *Delete* operation O_b in $IT\_ID(O_a, O_b)$, if $P(O_a) \leq P(O_b)$, then O_a must refer to a position which is to the *left* of or at the position referred to by O_b, so the assumed execution of O_b should not have any impact on the intended position of O_a. Therefore, no adjustment needs to be made to O_a. However, if $P(O_a) > (P(O_b) + L(O_b))$, which means that the position of O_a goes beyond the rightmost position in the deleting range of O_b, the intended position of O_a would have been shifted by $L(O_b)$ characters to the left if the impact of executing O_b was taken into account. Therefore, the position parameter of O_a is decremented by $L(O_b)$. Other-

```
FUNCTION 3. $IT\_II(O_a, O_b)$
{
  if $P(O_a) < P(O_b)$   $O'_a := O_a$;
  else $O'_a := Insert[S(O_a), P(O_a) + L(O_b)]$;
  return $O'_a$;
}
```

```
FUNCTION 4. $IT\_ID(O_a, O_b)$
{
  if $P(O_a) \leq P(O_b)$   $O'_a := O_a$;
  else if $P(O_a) > (P(O_b) + L(O_b))$   $O'_a := Insert[S(O_a), P(O_a) - L(O_b)]$;
  else $O'_a := Insert[S(O_a), P(O_b)]$;   $Save\_LI(O'_a, O_a, O_b)$;
  return $O'_a$;
}
```

```
FUNCTION 5. $IT\_DI(O_a, O_b)$
{
  if $P(O_b) \geq (P(O_a) + L(O_a))$   $O'_a := O_a$;
  else if  $P(O_a) \geq P(O_b)$   $O'_a := Delete[L(O_a), P(O_a) + L(O_b)]$;
  else  $O'_a := Delete[P(O_b) - P(O_a), P(O_a)] \oplus$
                $Delete[L(O_a) - (P(O_b) - P(O_a)), P(O_b) + L(O_b)]$;
  return $O'_a$;
}
```

```
FUNCTION 6. $IT\_DD(O_a, O_b)$
{
  if $P(O_b) \geq (P(O_a) + L(O_a))$   $O'_a := O_a$;
  else if $P(O_a) \geq (P(O_b) + L(O_b))$   $O'_a := Delete[L(O_a), P(O_a) - L(O_b)]$;
  else
    if $P(O_b) \leq P(O_a)$ and   $(P(O_a) + L(O_a)) \leq (P(O_b) + L(O_b))$
          $O'_a := Delete[0, P(O_a)]$;
    else if $P(O_b) \leq P(O_a)$ and $(P(O_a) + L(O_a)) > (P(O_b) + L(O_b))$
          $O'_a := Delete[P(O_a) + L(O_a) - (P(O_b) + L(O_b)), P(O_b)]$;
    else if $P(O_b) > P(O_a)$ and $(P(O_b) + L(O_b)) \geq (P(O_a) + L(O_a))$
          $O'_a := Delete[P(O_b) - P(O_a), P(O_a)]$;
    else  $O'_a := Delete[L(O_a) - L(O_b), P(O_a)]$;
    $Save\_LI(O'_a, O_a, O_b)$;
  return $O'_a$;
}
```

Fig. 4. Inclusion transformation functions.

wise, it must be that the intended position of O_a falls in the deleting range of O_b. In this case, O_b should not delete any characters to be inserted by O_a, and the new inserting position should be $P(O_b)$, according to the verification criteria.

9.2.2 *Operation Splitting for Split Segments.* Normally, an inclusion transformation produces a single transformed operation. However, to apply the inclusion transformation to a *Delete* operation O_a against an *Insert* operation O_b in $IT\_DI(O_a, O_b)$, when the inserting position of O_b falls into the deleting range of O_a, O_a should not delete any characters inserted by O_b, according to the verification criteria. Therefore, the deleting range will be split into two segments on the assumption that O_b has been executed, and the outcome of this transformation will be an operation O'_a being split into two suboperations, expressed as $O'_{a1} \oplus O'_{a2}$. The context-based relationship among O'_{a1}, O'_{a2}, and O_b is $O_b \mapsto O'_{a1}$, $O_b \mapsto O'_{a2}$, and $O'_{a1} \sqcup O'_{a2}$.

9.2.3 *Lost Information Saving for Reversibility.* According to the reversibility requirement, if $IT(O_a, O_b)$ produces O'_a, *then* $ET(O'_a, O_b)$ can be applied to get O_a back. Normally, adequate information is available in the parameters of O'_a (together with O_b) to recover O_a. However, when (1) O_b is a *Delete* operation and (2) O_a inserts a string or deletes some characters within the deleting range of O_b, then some information in O_a may get lost when being transformed into O'_a, so that O_a cannot be recovered by only using the information in the parameters of O'_a and O_b.

For example, to apply the inclusion transformation to an *Insert* operation O_a against a *Delete* operation O_b in $IT\_ID(O_a, O_b)$, when the inserting position of O_a falls into the deleting range of O_b (including the boundary case that $P(O_a) = P(O_b) + L(O_b)$), the position parameter of O'_a has to be $P(O_b)$, according to the verification criteria. In this case, the information about the offset from $P(O_b)$ to $P(O_a)$ is lost, so there will be no way to recover the original $P(O_a)$ by using only $P(O_b)$ in later exclusion transformations.

As another example, to apply the inclusion transformation to a *Delete* operation O_a against another *Delete* operation O_b in $IT\_DD(O_a, O_b)$, when the two delete operations' deleting ranges overlap, the transformed operation O'_a will have a deleting range shorter than that of O_a, and the position parameter of O'_a may have to be $P(O_b)$ if the position parameter of O_a falls into the deleting range of O_b, according to the verification criteria. In this case, the information about the original length and position of O_a is lost, and there will be no way to recover O_a by just using the parameters of O_b and O'_a.

To ensure reversibility, a utility routine $Save\_LI(O'_a, O_a, O_b)$ is used to save the lost information (e.g., the parameters of O_a before this transformation, the reference to O_b, etc.) into an internal data structure associated with O'_a, which will be used by the exclusion transformation function to recover O_a (see Section 9.3).

9.2.4 *An Example.* Consider two context-equivalent operations O_a = *Insert*["*aa*", 2], and O_b = *Delete*[3, 1], independently generated from the following document state: *ABCDE*. After applying the inclusion transformation to O_a against O_b, O_a will become O'_a = *Insert*["*aa*", 1]. If O_b was executed on the original document state, the new document state would become *AE*. Then, the execution of O'_a on the new document state would produce *AaaE*.

Apparently, the execution of O'_a on the new document state achieves the same effect as the execution of O_a on the original document state since (1) $S(O_a) = S(O'_a)$, and the string "*aa*" indeed exists in the document after the execution, and (2) the character "*A*" (or "*E*") which was at the left (or right) of position $P(O_a)$ in the original document state remains at the left (or right) of position $P(O'_a)$ in the new document state. Generally, it can be verified that given two context-equivalent operations O_a and O_b, $IT(O_a, O_b)$ will produce O'_a, such that the postconditions in Specification 1 are satisfied according to the verification criteria.

9.3 Exclusion Transformation

To apply the exclusion transformation to O_a against O_b, $ET(O_a, O_b)$ is invoked, which in turn, according to the operation types of O_a and O_b, calls one of the four subfunctions to do the real transformation. The definitions of the four subfunctions are given in Figure 5.

9.3.1 *Basic Transformation Strategy.* A precondition for O_a and O_b, i.e., $O_b \mapsto O_a$, is required to ensure that the relation of operation ranges of O_a and O_b can be *correctly* determined by simply comparing their parameters, so that the following basic exclusion transformation strategy can be applied: (1) compare the parameters of O_a and O_b to determine the relation of their operation ranges, (2) assume that O_b has been undone to find O_a's operation range in the new document state, and (3) adjust O_a's parameters to make O'_a, according to the comparison result in (1), the impact of the assumed undone in (2), and the verification criteria.

For example, to apply the exclusion transformation to an *Insert* operation O_a against another *Insert* operation O_b in $ET\_II(O_a, O_b)$, if $P(O_a) \leq P(O_b)$, then O_a must refer to a position which is to the *left* of the position referred to by O_b, so the assumed undoing of O_b should not have any impact on the intended position of O_a. Therefore, no adjustment needs to be made to O_a. Otherwise, if $P(O_a) \geq (P(O_b) + L(O_b))$, which means that the position of O_a goes beyond the rightmost position in the inserting range of O_b, the intended position of O_a would have been shifted by $L(O_b)$ characters to the left if the impact of undoing O_b was taken into account. Therefore, the position parameter of O_a is decremented by $L(O_b)$. Otherwise, it must be that the intended position of O_a falls in the middle of the string inserted by O_b. In this case, the basic strategy for the exclusion transformation is not applicable anymore, since undoing O_b will result in O_a's operation range undefined.

9.3.2 *Relative Addressing for Undefined Ranges.* Generally, when $O_b \rightarrow O_a$, and (1) O_b is an insert operation and (2) O_a inserts a string or deletes some characters within the string inserted by O_b, undoing O_b will result in O_a's operation range being undefined.

The technique for dealing with the *undefined range* problem is called *relative addressing*: the outcome of $ET(O_a, O_b)$, i.e., O'_a, is relatively addressed in the sense that its position parameter is relative to the position

```
FUNCTION 7. $ET\_II(O_a, O_b)$
{
  if $P(O_a) \leq P(O_b)$   $O'_a := O_a$;
  else if $P(O_a) \geq (P(O_b) + L(O_b))$   $O'_a := Insert[S(O_a), P(O_a) - L(O_b)]$;
  else $O'_a := Insert[S(O_a), P(O_a) - P(O_b)]$;   $Save\_RA(O'_a, O_b)$;
  return $O'_a$;
}

FUNCTION 8. $ET\_ID(O_a, O_b)$
{
  if $Check\_LI(O_a, O_b)$   $O'_a := Recover\_LI(O_a)$;
  else if $P(O_a) \leq P(O_b)$   $O'_a := O_a$;
  else $O'_a := Insert[S(O_a), P(O_a) + L(O_b)]$;
  return $O'_a$;
}

FUNCTION 9. $ET\_DI(O_a, O_b)$
{
  if $(P(O_a) + L(O_a)) \leq P(O_b)$   $O'_a := O_a$;
  else if $P(O_a) \geq (P(O_b) + L(O_b))$   $O'_a := Delete[L(O_a), P(O_a) - L(O_b)]$;
  else
    if $P(O_b) \leq P(O_a)$ and $(P(O_a) + L(O_a)) \leq (P(O_b) + L(O_b))$
        $O'_a := Delete[L(O_a), P(O_a) - P(O_b)]$;
    else if $P(O_b) \leq P(O_a)$ and $(P(O_a) + L(O_a)) > (P(O_b) + L(O_b))$
        $O'_a := Delete[P(O_b) + L(O_b) - P(O_a), (P(O_a) - P(O_b))] \oplus$
                $Delete[(P(O_a) + L(O_a)) - (P(O_b) + L(O_b)), P(O_b)]$;
    else if $P(O_a) < P(O_b)$ and $(P(O_b) + L(O_b)) \leq ((P(O_a) + L(O_a))$
        $O'_a := Delete[L(O_b), 0] \oplus Delete[L(O_a) - L(O_b), P(O_a)]$;
    else $O'_a := Delete[P(O_a) + L(O_a) - P(O_b), 0] \oplus Delete[P(O_b) - P(O_a), P(O_a)]$;
    $Save\_RA(O'_a, O_b)$;
  return $O'_a$;
}

FUNCTION 10. $ET\_DD(O_a, O_b)$
{
  if $Check\_LI(O_a, O_b)$   $O'_a := Recover\_LI(O_a)$;
  else if $P(O_b) \geq (P(O_a) + L(O_a))$   $O'_a := O_a$;
  else if $P(O_a) \geq P(O_b)$   $O'_a := Delete[L(O_a), P(O_a) + L(O_b)]$;
  else   $O'_a := Delete[P(O_b) - P(O_a), P(O_a)] \oplus$
                $Delete[L(O_a) - (P(O_b) - P(O_a)), P(O_b) + L(O_b)]$;
  return $O'_a$;
}
```

Fig. 5. Exclusion transformation functions.

parameter of the *base operation* O_b, instead of being relative to the beginning of the document (or absolutely addressed). In $ET\_DI(O_a, O_b)$ and $ET\_II(O_a, O_b)$, a utility routine $Save\_RA(O'_a, O_b)$ is used to save in the internal data structure associated with O'_a the fact that O'_a is relatively addressed with respect to the base operation O_b.

Relatively addressed operations will be converted into absolutely addressed operations (before giving it as an input parameter to a subsequent inclusion transformation function) by the higher-level list transformation functions (see Section 9.4). This strategy works because the outcome of the exclusion transformation is just an intermediate result of the top-level GOT control algorithm and will only be used for subsequent inclusion transformations, but never used for updating the document state.

9.3.3 *Lost Information Recovery.* To reverse the effect of an inclusion transformation in $ET\_ID(O_a, O_b)$ and $ET\_DD(O_a, O_b)$, a utility routine $Check\_LI(O_a, O_b)$ is used to check whether O_b was involved in an information-losing inclusion transformation which resulted in O_a. If yes, another utility routine $Recover\_LI(O_a)$ is used to recover O'_a from the information saved in O_a. Otherwise, the basic exclusion transformation strategy is applied to construct O'_a by using the information in the parameters of O_a and O_b.

9.3.4 *Operation Splitting for Split Segments.* Operation splitting may also occur in the exclusion transformation under two circumstances. Firstly, in $ET\_DD(O_a, O_b)$, when the position parameter of O_b falls into the deleting range of O_a, the deleting range will be split into two segments if O_b was undone, so the outcome of this transformation will be two *Delete* operations, O'_{a1} and O'_{a2}, corresponding to the two split deleting segments. The context-based relationship among O'_{a1}, O'_{a2}, and O_b is $O_b \sqcup O'_{a1} \sqcup O'_{a2}$.

Secondly, in $ET\_DI(O_a, O_b)$, when the deleting range of O_a covers some characters inserted by O_b and some characters outside the string inserted by O_b, the outcome will consist of two *Delete* operations: O'_{a1} with a relative address for deleting these characters inserted by O_b, and O'_{a2} with an absolute address for deleting the characters outside the string inserted by O_b. The relationship among O'_{a1}, O'_{a2}, and O_b is that O_b is the base operation of O'_{a1}, and $O_b \sqcup O'_{a2}$. The higher-level list transformation functions will handle these two delete operations differently (see Section 9.4).

9.3.5 *An Example.* Suppose the initial document state is *ABCDE*, and two operations O_b and O_a are generated as follows. First, operation O_b = *Insert*["*aa*", 1] was generated from the above document state. After the execution of O_b, the document state becomes *AaaBCDE*. Then, operation O_a = *Delete*[2, 4] was generated from the above document state. Apparently, $O_b \mapsto O_a$. If O_a is applied with the exclusion transformation against O_b, the transformed operation will be O'_a = *Delete*[2, 2]. If O'_a was executed in the context of CT_{O_b} (i.e., the initial document state), the editing result would be *ABE*, which achieves the same editing effect as that when O_a is executed in the context of $CT_{O_b} + O_b$ (i.e., the document state after the execution of O_b): *AaaBE*, since (1) the two characters "*CD*" which were in the deleting range of O_a indeed disappear from the document after the execution of O'_a and (2) no characters which were inserted by independent operation O_b have been deleted by O'_a.

Generally, it can be verified that given two operations O_a and O_b, where $O_b \mapsto O_a$, $ET(O_a, O_b)$ will produce O'_a, which satisfies the postconditions in Specification 2, according to the verification criteria.

9.4 Revised List Transformation

In Section 6.1, two list transformation functions are defined based on the specification of the inclusion and exclusion functions. These two list transformation functions have served as the interface between the low-level application-dependent transformation functions and the high-level generic GOT control algorithm. In this section, the two list transformation functions are revised in order to cope with split operations and relatively addressed operations produced by the primitive inclusion and exclusion transformation functions.

9.4.1 *Revised LIT( ).* To cope with the split operations, the two input parameters to the revised list inclusion transformation function, shown in Functions 11 and 12, become two lists of operations: $OL1$ and $OL2$; the outcome becomes a list of transformed operations as well.

```
FUNCTION 11. $LIT(OL1, OL2)$
{
  if $OL1 = [\ ]$ $RL := [\ ]$;
  else
    $TL1 := LIT(OL1[1], OL2)$;
    $TL2 := LIT(Tail(OL1), OL2 + TL1)$;
    $RL := TL1 + TL2$;
  return $RL$;
}
FUNCTION 12 $LIT1(O, OL)$
{
  if $OL = [\ ]$ $RL := [O]$;
  else if $Check\_RA(O)$ and not ($Check\_BO(O, OL[1])$)
    $RL := LIT1(O, Tail(OL))$
  else if $Check\_RA(O)$ and $Check\_BO(O, OL[1])$
    $O' := Convert\_AA(O, OL[1])$;
    $RL := LIT1(O', Tail(OL))$;
  else
    $RL := LIT(IT(O, OL[1]), Tail(OL))$;
  return $RL$;
}
```

The preconditions for input parameters of $LIT(OL1, OL2)$ are (1) for any operation O in $OL1$, it must be that either $O \sqcup OL2[1]$ or O is relatively addressed, and O's base operation is in $OL2$; and (2) for any two consecutive operations $OL2[i]$ and $OL2[i + 1]$ in $OL2$, $OL2[i] \mapsto OL2[i + 1]$.

The postconditions for the output of this function, denoted as $OL1'$, are (1) $OL2[l2] \mapsto OL1'[1] \mapsto OL1'[2] \mapsto \ldots \mapsto OL1'[l1]$, where $l2 = |OL2|$ and $l1 = |OL1'|$; and (2) the sequential execution of operations in $OL1'$ in the context of $CT_{OL1'[1]}$ preserves the intentions of all operations in $OL1$.

To apply the inclusion transformation to the list of operations in $OL1$ against the list of operations in $OL2$, $OL1[1]$ is first applied with the inclusion transformation against all operations in $OL2$ by calling $LIT1(OL1[1], OL2)$ to produce $TL1$. Then, $Tail(OL1)$ and $OL2 + TL1$ are used as input parameters for recursively calling $LIT(\ )$ to produce $TL2$. Finally, a list of transformed operations obtained by concatenating $TL1$ and $TL2$, i.e., $TL1 + TL2$, is returned. It should be noted that the outcome of $LIT1(OL1[1], OL2)$ (i.e., $TL1$) must be appended to $OL2$ (i.e., $OL2 + TL1$) in the recursive calling of $LIT(\ )$, so that the postcondition for $LIT(\ )$ can be satisfied.

The main functionality of $LIT1(O, OL)$ is to handle the special cases when the input operation O is relatively addressed. The basic strategy here is that if O is a relatively addressed operation, it will be first converted into an absolutely addressed operation based on the information available in its base operation in OL; and then the absolutely addressed operation will be applied with the inclusion transformation against only those operations which are after the base operation in OL. As shown in Function 12, a utility routine $Check\_RA(O)$ is used to check whether O is relatively addressed. If yes, another utility routine $Check\_BO(O, OL[1])$ is used to check whether $OL[1]$ is O's base operation. If $OL[1]$ is not O's base operation, then $OL[1]$ is skipped, and $LIT1(O, Tail(OL))$ is recursively invoked. If, however, $OL[1]$ is indeed O's base operation, then a utility routine $Convert\_AA(O, OL[1])$ is used to convert O into an absolutely addressed operation O', and $LIT1(O', Tail(OL))$ is recursively invoked. In case that O is not a relatively addressed operation, $LIT(IT(O, OL[1]), Tail(OL))$ is recursively invoked.

9.4.2 *Revised LET( ).* To cope with the split operations, the two input parameters to the revised list exclusion transformation function, shown in Functions 13 and 14, become two lists of operations: $OL1$ and $OL2$; and the outcome becomes a list of transformed operations as well.

```
FUNCTION 13. $LET(OL1, OL2)$
{
  if $OL1 = [\ ]$ $RL := [\ ]$;
  else
    $TL1 := LET1(OL1[1], OL2)$;
    $TL2 := LET(Tail(OL1), OL2)$;
    $RL := TL1 + TL2$;
  return $RL$;
}
FUNCTION 14. $LET1(O, OL)$
```

```
{
  if $Check\_RA(O)$ or $OL = [\ ]$ $RL := [O]$;
  else $RL := LET(ET(O, OL[1]), Tail(OL))$;
  return $RL$;
}
```

The preconditions for input parameters of $LET(OL1, OL2)$ are (1) for any operation O in $OL1$, it must be that either $OL2[1] \mapsto O$ or O is relatively addressed; and (2) for any two consecutive operations $OL2[i]$ and $OL2[i + 1]$ in $OL2$, $OL2[i + 1] \mapsto OL2[i]$. The postcondition for the output of this function, denoted as $OL1'$, is that for any operation O in $OL1'$, it must be that either $O \sqcup OL2[len]$, where $len = |OL2|$, or O is relatively addressed.

To apply the exclusion transformation to the list of operations in $OL1$ against the list of operations in $OL2$, $OL1[1]$ is first applied with the exclusion transformation against all operations in $OL2$ by calling $LET1(OL1[1], OL2)$ to produce $TL1$. Then, $Tail(OL1)$ and $OL2$ are used as input parameters for recursively calling $LET(\ )$ to produce $TL2$. Finally, a list of transformed operations obtained by concatenating $TL1$ and $TL2$, i.e., $TL1 + TL2$, is returned. In contrast to $LIT(\ )$, the outcome of $LET1(OL1[1], OL2)$ (i.e., $TL1$) is not appended to $OL2$ in the recursive calling of $LET(\ )$ so that the postcondition for $LET(\ )$ can be satisfied.

As for $LIT1(O, OL)$, the main functionality of $LET1(O, OL)$ is to handle the special cases when the input operation O is relatively addressed. However, the strategy here is relatively simple: if operation O is relatively addressed, then (1) the operations in OL will be skipped and (2) the function returns O. Otherwise, $LET(ET(O, OL[1]), Tail(OL))$ is recursively invoked. It should be noted that the strategy used here in $LET1(O, OL)$ to skip the exclusion transformation against all operations in OL in case that O is relatively addressed is corresponding to the strategy used in $LIT1(O, OL)$ to apply the inclusion transformation to the relatively addressed operation O against only those operations in OL which are after the base operation.

With the two list transformation functions being revised, the top-level control structure of the GOT control algorithm and the undo/transform-do/transform-redo scheme remain unchanged.

10. AN INTEGRATED EXAMPLE

In this section, we use one example to illustrate how the whole system works when the generic part and the application-dependent part are integrated.

The scenario in Figure 2 will be used once more, with the initial document state being the following contents:

“*ABCDEFGH*”

The four editing operations in the scenario are instantiated as follows:

(1) $O_1 = \mathit{Delete}[3, 2]$ (intention: to delete "*CDE*")

(2) $O_2 = \mathit{Insert}[\text{"}\mathit{abcd}\text{"}, 4]$ (intention: to insert "*abcd*" between "*ABCD*" and "*EFGH*")

(3) $O_3 = \mathit{Delete}[4, 5]$ (intention: to delete "*dFGH*")

(4) $O_4 = \mathit{Delete}[2, 6]$ (intention: to delete "*cd*")

Moreover, let DS_j^i denote the document state at site i after executing j operations, and let $(DS_j^i)\ EO_x o EO_y$ denote the application of operations EO_x and EO_y in sequence on DS_j^i. The execution form of each operation and the document state after its execution at different sites are illustrated below. (Example 1 in Section 7 should be referred to in order to understand why the execution form of an operation is obtained in the way illustrated here.)

—*Site 0*: $DS_0^0 :=$ "*ABCDEFGH*"

(1) Execute O_1:

$EO_1 := O_1$

$DS_1^0 := (DS_0^0)EO_1 =$ "*ABFGH*"

(2) Execute O_2:

$EO_2 := IT(O_2, EO_1) = \mathit{Insert}[\text{"}\mathit{abcd}\text{"}, 2]_{LI}$[7]

$DS_2^0 := (DS_1^0)EO_2 =$ "*ABabcdFGH*"

(3) Execute O_4:

$O_2 := ET(EO_2, EO_1) = \mathit{Insert}[\text{"}\mathit{abcd}\text{"}, 4]$[8]

$O'_4 := ET(O_4, O_2) = \mathit{Delete}[2, 2]_{RA}$[9]

$EO_4 := LIT(O'_4, [EO_1, EO_2]) = \mathit{Delete}[2, 4]$[10]

$DS_3^0 := (DS_2^0)\ EO_4 =$ "*ABabFGH*"

(4) Execute O_3:

$EO_3 := IT(O_3, EO_4) = \mathit{Delete}[3, 4]_{LI}$[11]

$DS_4^0 := (DS_3^0)EO_3 =$ "*ABab*"

—*Site 1*: $DS_0^1 :=$ "*ABCDEFGH*"

(1) Execute O_2:

$EO_2 := O_2$

[7] The inserting position of O_2 falls in the deleting range of EO_1, so $P(EO_2) = P(EO_1)$. This is an information-losing transformation.

[8] Since EO_2 is the outcome of an information-losing transformation, $\mathit{Recover\_LI}(EO_2, EO_1)$ is called to obtain O_2 in this exclusion transformation.

[9] Since the deleting range of O_4 is within the string inserted by O_2, O'_4 has to be relatively addressed, with O_2 being the base operation.

[10] In this list transformation, EO_1 will be skipped (since EO_4 is relatively addressed, but EO_1 is not its base operation), and $P(EO_4) = P(O'_4) + P(EO_2)$.

[11] The position parameter of O_3 falls in the deleting ranges of EO_4, so EO_3's deleting range is shorter than the deleting range of O_3, and its position parameter is equal to that of EO_4. This is also an information-losing transformation.

$DS_1^1 := (DS_0^1)EO_2 =$ "*ABCDabcdEFGH*"

(2) Execute O_1:

Undo EO_2 (i.e., do $\overline{EO_2} = Delete[4, 4]$):

$(DS_1^1)\overline{EO_2} =$ "*ABCDEFGH*"

Do O_1:

$EO_1 := O_1$

$(DS_1^1)\overline{EO_2} o EO_1 =$ "*ABFGH*"

Transform-redo EO_2:

$EO_2' := IT(EO_2, EO_1) = Insert[$"*abcd*"$, 2]_{LI}$

$DS_2^1 := (DS_1^1)\overline{EO_2} o EO_1 o EO_2' =$ "*ABabcdFGH*"

(3) Execute O_3:

$EO_3 := O_3$

$DS_3^1 := (DS_2^1)EO_3 =$ "*ABabc*"

(4) Execute O_4:

Undo EO_3 (i.e., do $\overline{EO_3} = Insert[$"*dFGH*"$, 5]$):

$(DS_3^1)\overline{EO_3} =$ "*ABabcdFGH*"

Transform-do O_4 (the same as stage (3) at site 0):

$EO_4 := Delete[2, 4]$

$(DS_3^1)\overline{EO_3} o EO_4 =$ "*ABabFGH*"

Transform-redo $EO_3 (= O_3)$:

$EO_3' := IT(EO_3, EO_4) = Delete[3, 4]_{LI}$

$DS_4^1 := (DS_3^1)\overline{EO_3} o EO_4 o EO_3' =$ "*ABab*"

—*Site 2*: $DS_0^2 :=$ "*ABCDEFGH*"

(1) Execute O_2:

$EO_2 := O_2$

$DS_1^2 := (DS_0^2)EO_2 =$ "*ABCDabcdEFGH*"

(2) Execute O_4:

$EO_4 := O_4$

$DS_2^2 := (DS_1^2)EO_4 =$ "*ABCDabEFGH*"

(3) Execute O_1 (O_3 is suspended):

Undo EO_4 and EO_2 (i.e., do $\overline{EO_4} = Insert[$"*cd*"$, 6]$ and $\overline{EO_2} = Delete[4, 4]$):

$(DS_2^2)\overline{EO_4} o \overline{EO_2} =$ "*ABCDEFGH*"

Do O_1:

$EO_1 := O_1$

$(DS_2^2)\overline{EO_4} o \overline{EO_2} o EO_1 =$ "*ABFGH*"

Transform-redo EO_2 and EO_4:

$EO_2' := IT(EO_2, EO_1) = Insert[$"*abcd*"$, 2]_{LI}$

$TO := ET(EO_4, EO_2) = Delete[2, 2]_{RA}$

$EO_4' := LIT(TO, [EO_1, EO_2']) = Delete[2, 4]$

$DS_3^2 := (DS_2^2)\overline{EO_4}o\overline{EO_2}oEO_1oEO'_2oEO'_4$ = "*ABabFGH*"

(4) Execute O_3:

$EO_3 := IT(O_3, EO'_4) =$ *Delete*[3, 4]

$DS_4^2 := (DS_3^2)EO_3$ = "*ABab*"

From the above illustration, it can be observed that the document states of different sites at the same stages (i.e., after executing the same number of operations) have the following relationship:

$$Stage\ 1.\therefore\ DS_1^0 \neq DS_1^1 = DS_1^2$$

$$Stage\ 2.\therefore\ DS_2^0 = DS_2^1 \neq DS_2^2$$

$$Stage\ 3.\therefore\ DS_3^0 = DS_3^1 \neq DS_3^2$$

$$Stage\ 4.\therefore\ DS_4^0 = DS_4^1 = DS_4^2$$

The above document states relationship highlights the following aspects of the convergence property of our consistency model: (1) different sites are allowed to execute different independent operations and hence have divergent document states at any intermediate stages of a session, e.g., at stage 1, $DS_1^0 \neq DS_1^1$, due to executing the two independent operations O_1 and O_2 at site 0 and 1, respectively; (2) when two sites have executed the same set of operations, their document states must be the same (e.g., $DS_2^0 = DS_2^1$); in particular, when all sites have executed the same set of operations at the end of a session, document states at all sites are ensured to be identical (i.e., $DS_4^0 = DS_4^1 = DS_4^2$).

In addition, this example also demonstrates the following properties of the intention-preserving scheme: (1) the characters inserted by an *Insert* operation (e.g., O_2) may not be deleted by another independent *Delete* operation (e.g., O_1), but may be deleted only by other dependent *Delete* operations (e.g., O_3 and O_4); (2) if the deleting ranges of two or more independent *Delete* operations (e.g., O_3 and O_4) partially or completely overlap, the effective deleting range of these (transformed) independent *Delete* operations will be the union of their separate deleting ranges.

Finally, one interface-related issue is worth a brief mentioning: although all editing operations in the example are expressed as *atomic* stringwise operations, they could have actually been generated from the user interface (e.g., a keyboard) character-by-character. To achieve good responsiveness, local operations on every single character should be immediately executed and reflected on the local user interface. However, characterwise local operations need not be immediately propagated to remote sites. Instead, they can be accumulated and then converted into stringwise operations for communication, transformation, and remote execution.

11. AN INTERNET-BASED PROTOTYPE IMPLEMENTATION

All algorithms presented in the previous sections have been implemented in an Internet-based prototype REDUCE (REal-time Distributed Unconstrained Cooperative Editing) system [Sun et al. 1997a] using the programming language Java. The prototype REDUCE system consists of multiple cooperating REDUCE sites, typically a PC machine or a workstation, connected by Internet. The REDUCE process running on each site is called *Site Server* (SS). In addition, there is a centralized REDUCE *Session Manager* (SM) which manages cooperating membership and sessions, but which is not involved in handling cooperative editing operations. SM may run on a special server machine or on one of the REDUCE sites. Only SS is relevant to the issues discussed in this article.

The central component in an SS process is the REDUCE *Engine* (RE), which implements all algorithms for achieving convergence, causality preservation, and intention preservation. There are multiple concurrent threads inside an SS process, with one dedicated thread handling requests propagated from every remote site, and one special thread handling operations generated from the input part of the local user interface (i.e., the keyboard and mouse). RE is implemented as a Java *monitor* object to provide the multiple threads with a synchronized point of access to the output part of the local graphical user interface (i.e., the screen). Each thread repeatedly executes the following two steps: waiting for requests (carrying timestamped operations) from the corresponding remote/local site; then invoking RE to handle the incoming request. When a thread invokes RE, this thread may be blocked (outside RE) until there is not any active thread inside RE. Then, RE first checks to see whether the incoming operation is causally ready for execution by invoking the causality-preserving algorithm. If not, this thread is blocked (inside RE) until it is waken up later by another thread leaving RE. If the operation is causally ready, RE invokes the undo/transform-do/transform-redo scheme to do the job. After that, the garbage collection algorithm is invoked to delete useless operations in the history buffer. Before leaving RE, all threads being blocked inside RE are notified to continue (and to check the causally ready conditions again). Local requests handling has higher priority than remote requests handling and may never be blocked inside RE.

The correctness of the causality-preserving scheme and the garbage collection scheme relies on a reliable and order-preserving communication channel between any pair of cooperative sites. This communication channel is naturally implemented by a Java socket based on the Internet TCP/IP protocol. Except some complications caused by operations splitting in stringwise inclusion/exclusion transformations, the implementation of the consistency maintenance algorithms presented in this article is straightforward. Our experience with using the prototype system has shown that the overhead involved in the undo/transform-do/transform-redo scheme has been very small and has rarely had any visible impact on the responsiveness of the user interface. This is mainly because the number of operations involved in each invocation of the undo/transform-do/transform-redo scheme is very small (often less than 3) due to the facts that (1) the number of cooperating sites involved in each testing session is small (3 to 5 sites), (2) operations are normally arriving in their correct causal and total orders, and (3) the garbage collection scheme is effective in reducing the number of operations in the history buffer. Therefore, the worst-case theoretical complexity of the algorithm (which is still to be analyzed precisely and to be improved in our future work), under the circumstances of a large number of cooperating sites and a large number of operations in the history buffer, has not concerned us yet in practice.

The current prototype system has been developed mainly to test the feasibility of our approach and to explore system design and implementation issues. Efforts are being directed toward building a more robust and useful system, which will be used by external users in real application contexts to evaluate the research results from end-users' perspective.

12. DISCUSSION OF ALTERNATIVE AND RELATED WORK

In this section, some alternative and related approaches to consistency maintenance in real-time cooperative editing systems will be examined, with special regard to their capability in achieving convergence, causality preservation, and intention preservation under the constraints of a short response time and support for unconstrained collaboration in distributed environments.

12.1 Alternative Approaches

12.1.1 *Turn-Taking.* Only one user at a time has the "token/floor" to edit the shared document [Ellis et al. 1991]. Access to the token may be controlled by internal technical protocols (implemented by software) or through external social protocols (followed by cooperating users to mediate their actions [Greenberg and Marwood 1994]). Since there is only a single active user at any instant of time, none of the three inconsistency problems occurs in such systems. Therefore, convergence, causality preservation, and intention preservation are ensured at the price of not supporting concurrent editing. Consequently, this approach is limited to just situations where a single active user fits the need of collaborative working and is ill suited to application environments where the nature of collaboration is characterized by concurrent streams of activities from multiple users [Dourish 1995].

12.1.2 *Locking.* An object (e.g., a word, a line, or a section, etc.) is first locked before it is updated, so only one user at a time is able to update an object [Greenberg and Marwood 1994; Knister and Prakash 1993; McGuffin and Olson 1992]. Concurrent editing is allowed under the condition that multiple users are locking and editing different objects. Locking can prevent multiple users from generating conflicting editing operations over the same object, since only one user can edit one object at any instant of time. However, unless the granularity of locking is the whole document (thus resorting to the turn-taking protocol), none of the three inconsistency problems can be resolved by locking, since the occurrence of these inconsistency problems is independent of whether or not editing operations refer to the same object (at least in the domain of text editing). Besides, when concurrent locking is mixed with concurrent editing, the three inconsistency problems have to be resolved for locking operations as well as for normal editing operations. Moreover, locking increases response time due to the overhead of requesting and releasing locks in distributed environments.

12.1.3 *Serialization.* Operations may be generated concurrently, but the execution effects will be the same as if all operations were executed in the same total order at all sites. This can be achieved either by pessimistically delaying the execution of an operation until all totally preceding operations have been executed [Lamport 1978; Sun and Maheshwari 1996] or by optimistically executing operations upon their arrival or generation without regard to their ordering, but using undo/redo to repair the out-of-order execution effect [Greenberg and Marwood 1994; Karsenty and Beaudouin-Lafon 1993]. Apparently, serialization can solve the problem of divergence. However, serialization-based protocols have the following problems. First, the intention violation problem cannot be solved by serializing concurrent operations in any order (at least in the text-editing domain). Second, the causality violation problem remains unsolved if operations are executed upon their arrival in the optimistic case or if the total ordering is inconsistent with the causal ordering among operations in the pessimistic case. Lastly, responsiveness may be lost if operations are not executed immediately after their generation, but after some delay in the pessimistic case.

12.1.4 *Causal Ordering.* Operations may be generated and executed concurrently, but their execution order is constrained by their natural causal order. This can be achieved by using the well-known *vector logical clock* (i.e., the state vector) technique [Fidge 1988; Raynal and Singhal 1996]. In this approach, local operations can be executed immediately after their generation, so responsiveness is good; but some remote operations may be delayed until all causally preceding operations have been executed [Ellis and Gibbs 1989; Sun et al. 1996a]. This approach can achieve causality preservation only, but it does not address the problems of divergence and intention violation. It is worth pointing out that the use of state vectors to enforce the causal ordering among group operations/messages has been a "classic" technique exploited in many distributed computing systems, such as the ISIS distributed group communication tools [Birman et al. 1991] and the Amoeba distributed operating system [Mullender et al. 1990]. However, the generic group communication services provided by these systems are often too heavyweighted for real-time groupware applications: a reliable connection-oriented point-to-point communication service is often what is needed (as in the REDUCE system). Moreover, the consistency maintenance algorithms in some real-time groupware systems (such as the one presented in this article) often need explicit knowledge of causal relationship among messages/operations in order to work properly, but some general-purpose distributed systems try to make this knowledge *transparent* to applications. That is why real-time groupware systems often have to bypass these general-purpose distributed systems and to handle causality and state vectors explicitly.

12.1.5 *Transformation.* Operations can be generated and executed concurrently, but they may be transformed before their execution so that the execution of the same set of properly transformed operations in different orders could produce identical document states [Ellis and Gibbs 1989]. Transformation is often combined with the casual-ordering approach to achieve both convergence and causality preservation. In this approach, local operations can be executed immediately after their generation, so responsiveness is good. Moreover, transformation is especially suitable to achieve intention preservation, as discussed in this article. Transformation has been found a promising approach not only to maintaining document consistency, but also to supporting user-initiated *undo* in collaborative systems [Prakash and Knister 1994; Ressel et al. 1996].

12.2 Comparison to Related Work

Using operational transformation to maintain consistency in real-time cooperative editing systems was pioneered by Ellis and Gibbs [1989] in the GROVE system, to which our work is most closely related. Therefore, a more detailed comparison of our approach to existing transformation-based approaches is in order.

The GROVE system used the distributed operational transformations (dOPT) algorithm for transforming independent operations. Essentially, the dOPT algorithm adopted an inclusion transformation strategy. The basic idea was to perform the inclusion transformation on each causally ready operation against all executed independent operations in the Log (i.e., the history buffer) in the order from the oldest to the most recent. The GROVE operational transformation approach has been a major inspiration to our work in this area. In the initial stage of our research, we tried to incorporate the dOPT algorithm into our early experimental prototype system, but we found the dOPT algorithm did not always produce identical and desired (i.e., intention-preserved) results at all sites. In fact, it took us a long time to find the root of the problem: the relationship among independent operations is diverse in an unconstrained cooperative environment, but the inclusion transformation algorithm works correctly only if the pair of input operations was generated (either originally or by transformation) from the same document state. In recognizing this problem, we introduced an additional exclusion transformation, specified the pre- and postconditions for the inclusion and exclusion transformations and devised the GOT control algorithm which determines when and how inclusion/exclusion transformation is applied to which operations and ensures the preconditions are always met. As long as the application-dependent transformation functions satisfy the specified postconditions, the GOT control algorithm is able to preserve the intentions of all independent operations no matter whether they are generated from the same or different document states.

In parallel with our work, another research group also discovered that the dOPT algorithm did not work if one user issues and executes more than one operation concurrently with an operation of another user, and they proposed a different approach, called the adOPTed algorithm, to solve the problem [Ressel et al. 1996]. The adOPTed algorithm added to the original dOPT algorithm a multidimensional interaction graph, which keeps track of all valid paths of transforming operations, and a double recursive function (similar in functionality to our GOT control algorithm) to determine which operations should apply the *L-Transformation* (similar to our *Inclusion Transformation*) against which others. If the *L-Transformation* functions could always satisfy the properties specified by Ressel et al. [1996], the adOPTed approach would be equivalent to our approach in the sense that the execution of the same set of operations on the same initial document by the two algorithms will produce the same outcome document. The proof (or disproof) of the equivalence between the two approaches is an interesting topic, but beyond the scope of this article. In the following, we will discuss the important differences between our approach and the adOPTed approach.

Firstly, our algorithm works on a linear history buffer containing operations in their executed forms, whereas the adOPTed algorithm works on an N-dimensional (where N is the number of cooperating sites) interaction graph containing all operations in various possible forms (i.e., the original, intermediate, and executed) in addition to a linear Log (the same as our history buffer) with operations in their original forms. The interaction graph provides a very useful model for visualizing the transformation relationship among original and transformed operations, but maintaining and searching a dynamically growing and potentially large N-dimensional graph at run time is inefficient and unnecessary (as proved by our approach).

Secondly, the way of ensuring convergence is different in the two approaches. Our approach distinguishes the convergence issue from the intention preservation issue and ensures convergence by the higher-level undo/transform-do/transform-redo scheme. In essence, undo/redo is also a kind of transformation, which is performed directly on the document states rather than on the operations and which is generic rather than being application dependent. The correctness of our convergence scheme is established by the fact that the final document states at all sites will be the same as if all operations were executed in the same total order. In contrast, the adOPTed approach achieves both convergence and intention preservation at the application-dependent transformation algorithm level. The correctness of the adOPTed approach can be ensured by requiring *L-Transformation* functions to guarantee the uniqueness of the labeling of vertices (for document states) and edges (for original/transformed operations) of the interaction graph. However, due to the mixed complications in both convergence and intention preservation and the application-dependent

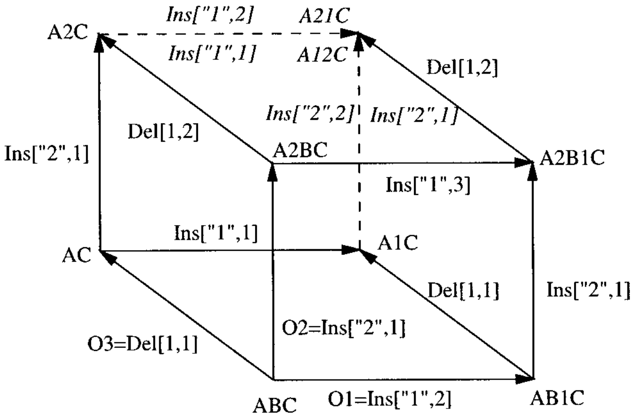


Fig. 6. An interaction graph with ambiguously labeled edges and vertices.

nature of transformation functions, it is difficult to verify whether a given *L-Transformation* function satisfies the properties required in Ressel et al. [1996]. In fact, it is fairly easy to propose seemingly correct *L-Transformation* functions which do not really guarantee the uniqueness of the labeling of vertices and edges of the interaction graph.

To illustrate, consider three operations: O_1 = *Insert*[“1”, 2], O_2 = *Insert*[“2”, 1], and O_3 = *Delete*[1, 1], generated from the same initial document state “ABC” at sites 1, 2, and 3 respectively. According to the adOPTed algorithm and the *L-Transformation* functions for text editing given in Ressel et al. [1996], we constructed the interaction graph for these operations, shown in Figure 6. As illustrated, the graph contains two ambiguously labeled edges (denoted as dashed arrows, one is labeled by *Insert[“1”, 2]* and *Insert[“1”, 1]*, and the other is labeled by *Insert[“2”, 2]* and *Insert[“2”, 1]*), and one ambiguously labeled vertices (labeled by *“A12C”* and *“A21C”*), which means that transforming and executing the three operations along different paths may result in different final states![12] One may be tempted to fix this problem by reversing the following rule used in Ressel et al. [1996]: when two insert operations have the same position parameter, the position of the operation with a *larger* site identifier will be shifted. Unfortunately, this quick fix works only in this case, but it fails in another rather similar scenario obtained by simply reversing the site identifiers of O_1 and O_2: the root of the problem is deeper than this and requires a more sophisticated solution than just using the site identifier. From our experience, the distinction between convergence and intention preservation has greatly helped us to understand the nature of both issues and to verify the correctness of the solutions to both issues.

[12] It should be noted that the two divergent final results—*“A12C”* and *“A21C”*—are both intention-preserved results according to the general definition of intention of operations (Definition 3) and the concrete criteria given in Section 9. This is one example that divergent final results do not necessarily lead to intention violation.

Finally, in contrast to the *characterwise* text-editing transformation algorithms proposed in Ellis and Gibbs [1989] and Ressel et al. [1996], our inclusion and exclusion transformation functions are string-wise. The extension from characterwise transformation to stringwise transformation is important and worthwhile, since stringwise transformation is more general and can greatly reduce the number of transformations and communications. This extension is also nontrivial, since stringwise transformation involves quite some complications not occurring in characterwise transformation, as shown in Section 9. To the best of our knowledge, there has been no published work on stringwise transformation algorithms for the same purpose. We strongly believe that an in-depth and detailed investigation of transformation algorithms deserve serious attention because this level of investigation enables us to better understand the intrinsic interactions (in the form of pre- and/or postconditions) between transformation algorithms and higher-level control algorithms and to devise correct solutions at both lower and higher levels.

13. CONCLUSIONS AND FUTURE WORK

In this article, we started from identifying three kinds of inconsistency problems—divergence, causality violation, and intention violation—in real-time cooperative editing systems with a replicated architecture. Particularly, we distinguished the nonserializable intention violation problem from the serializable divergence problem. Then, a consistency model with three properties—convergence, causality preservation, and intention preservation—has been proposed as the framework for resolving the identified three inconsistency problems. The convergence property ensures the replicated copies of the shared document at all sites be the same at the end of a session. The causality preservation property ensures the execution order of dependent operations be the same as their natural cause-effect order during the process of a session. The intention preservation property ensures (1) that the effect of executing an operation on any document states be the same as the effect of executing this operation on the document state from which it was generated and that (2) the effects of independent operations never interfere with each other. The consistency model imposes execution order constraint only on dependent operations, but not on independent operations as long as the convergence and intention preservation properties are always maintained.

Furthermore, an integrated set of schemes for supporting the consistency model have been devised and discussed in detail. Causality preservation has been achieved by a state-vector-based timestamping scheme. Convergence has been achieved by an optimistic serialization strategy using undo/do/redo. Achieving intention preservation has been the major technical challenge to our approach. Based on the recognition that independent operations may be generated from different document states in an unconstrained cooperative environment, we introduced the notion of operation *context* to capture the required relationship between operations for *correct* transformation. Then, the context-based pre- and postconditions were specified for a pair of generic inclusion and exclusion transformation functions. To cope with the complications involved, we divided the whole intention-preserving scheme into two parts: one generic part, which is based solely on the dependency and context relationships among operations, and the other application-dependent part, which relies on semantics of the operations involved. A novel and generic operational transformation (GOT) control algorithm was devised to determine when and how an inclusion/exclusion transformation be applied to which operations and to ensure the preconditions be always met. As long as the application-dependent transformation functions satisfy the specified postconditions, the GOT control algorithm is able to preserve the intentions of all independent operations no matter whether they are generated from the same or different document states. Furthermore, the GOT control algorithm has been integrated with the undo/do/redo scheme to form an undo/transform-do/transform-redo scheme for achieving both convergence and intention preservation. In addition, a distributed garbage collection scheme has been devised for managing the history buffer.

Apart from the generic operational transformation control algorithm, a pair of concrete inclusion and exclusion transformation algorithms has been devised for stringwise *Insert*/*Delete* operations in cooperative text-editing systems. The main issues addressed in this part include the criteria for verifying intention-preserved editing effects, the basic strategies for normal transformation based on operations' parameters, and the strategies for handling exceptions, such as lost information saving and/or recovery for ensuring reversibility, relatively addressing for undefined ranges, and operation splitting for split segments. Although the transformation algorithms were designed in the context of text editing, they are actually quite general and potentially applicable in other real-time groupware systems, which allow concurrent insertion/deletion of any sequence of objects with a linearly ordered relationship, such as a sequence of pages in a document/book [Moran et al. 1995], a sequence of slides in a seminar/lecture, a sequence of frames in a movie/video, etc.

Without the experimental effort to construct working prototype cooperative editing systems, we would not have learned some of the real challenges involved and would not have been motivated to devise new models and techniques to address the challenging problems. On the other hand, our theoretical effort in (partially) formalizing the identified problems and the proposed solutions has played a crucial role in enabling us to understand the nature and complexity of the inconsistency problems, to design correct algorithms, and to efficiently implement the algorithms. This combined experimental and theoretical approach will continue to be applied in our future research on various aspects of real-time cooperative editing systems.

We are currently applying the technical components used for consistency maintenance, including the state-vector timestamping, the history buffer, the GOT control algorithm, and the inclusion and exclusion transformation functions, to support user-initiated *collaborative undo* in a way similar to that in Prakash and Knister [1994], but without the use of any locks. We are also applying the consistency model and the generic supporting schemes to cooperative graphics editors to further validate our model and gain more insight in the design of these types of systems. Research on real-time cooperative editing systems, as a special class of distributed computing systems supporting *human-computer-human* interactions, has drawn inspiration from *traditional* distributed-computing techniques (e.g., causal/total ordering of events, state-vector timestamping, serialization, etc. [Bernstein et al. 1987; Birman et al. 1991; Fidge 1988; Lamport 1978; Mullender et al. 1990; Raynal and Singhal 1996; Sun and Maheshwari 1996]) and has invented *nontraditional* techniques to address its special issues (e.g., intention preservation and operational transformation, etc. [Ellis and Gibbs 1989; Ressel et al. 1996; Sun et al. 1996a; 1997b; 1998]). We believe that many of these novel models, concepts, and techniques invented in real-time cooperative editing systems research are fundamental, general, and potentially applicable to other areas of distributed computing. We hope the work presented in this article could be an inspiration to researchers looking for innovative solutions to challenging problems in areas of both CSCW and distributed computing in general.

ACKNOWLEDGMENTS

We wish to thank the anonymous referees and the Associated Editor, John Carroll, for their very valuable comments and suggestions which helped improve the final presentation of this article.

REFERENCES

BERNSTEIN, P., GOODMAN, N., AND HADZILACOS, V. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Mass.

BIRMAN, K., SCHIPER, A., AND STEPHENSON, P. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9,* 3 (Aug.), 272–314.

DOURISH, P. 1995. The parting of the ways: Divergence, data management and collaborative work. In *Proceedings of the 4th European Conference on Computer-Supported Cooperative Work* (Sept.). 215–230.

DOURISH, P. 1996. Consistency guarantees: Exploiting application semantics for consistency management in a collaboration toolkit. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (Nov.). 268–277.

ELLIS, C. A. AND GIBBS, S. J. 1989. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (May). 399–407.

ELLIS, C. A., GIBBS, S. J., AND REIN, G. L. 1991. Groupware: Some issues and experiences. *Commun. ACM 34,* 1 (Jan.), 39–58.

FIDGE, C. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference.* 56–66.

GREENBERG, S. AND MARWOOD, D. 1994. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work* (Nov.). 207–217.

HYMES, C. M. AND OLSON, G. M. 1992. Unblocking brainstorming through the use of a simple group editor. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (Nov.). 99–106.

KARSENTY, A. AND BEAUDOUIN-LAFON, M. 1993. An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems* (May). 195–202.

KNISTER, M. AND PRAKASH, A. 1993. Issues in the design of a toolkit for supporting multiple group editors. *J. Usenix Assoc. 6,* 2, 135–166.

LAMPORT, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21,* 7, 558–565.

MCGUFFIN, L. AND OLSON, G. 1992. ShrEdit: A shared electronic workspace. CSMIL Tech. Rep. 45, The University of Michigan.

MORAN, T., MCCALL, K., VAN MELLE, B., PEDERSEN, E., AND HALASZ, F. 1995. Some design principles for sharing in tivoli, a whiteboard meeting-support tool. In *Groupware for Real-Time Drawings: A Designer's Guide.* S. Greenberg, Ed. McGraw-Hill, 24–36. International (UK).

MULLENDER, S., VAN ROSSUM, G., TANENBAUM, A., VAN RENESSE, R., AND VAN STAVEREN, H. 1990. Amoeba—A distributed operating system for the 1990s. *IEEE Comput. 23,* 5 (May), 44–53.

OLSON, J., OLSON, G., STRORROSTEN, M., AND CARTER, M. 1992. How a group-editor changes the character of a design meeting as well as its outcome. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (Nov.). 91–98.

PRAKASH, A. AND KNISTER, M. 1994. A framework for undoing actions in collaborative systems. *ACM Trans. Comput. Hum. Interact. 4,* 1 (Dec.), 295–330.

RAYNAL, M. AND SINGHAL, M. 1996. Logical time: Capturing causality in distributed systems. *IEEE Comput. 29,* 2 (Feb.), 49–56.

RESSEL, M., NITSCHE-RUHLAND, D., AND GUNZENBAUSER, R. 1996. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (Nov.). 288–297.

SUN, C. AND MAHESHWARI, P. 1996. An efficient distributed single-phase protocol for total and causal ordering of group operations. In *Proceedings of the IEEE 3rd International Conference on High Performance Computing* (Dec.). IEEE Computer Society, 295–300.

SUN, C., CHEN, D., AND JIA, X. 1998. Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems. In *Proceedings of the 21st Australasian Computer Science Conference* (Perth, Australia, Feb.) 441–452.

SUN, C., JIA, X., YANG, Y., AND ZHANG, Y. 1997a. Reduce: A prototypical cooperative editing system. In *Proceedings of the 7th International Conference on Human-Computer Interaction* (San Francisco, Aug.). 89–92.

SUN, C., JIA, X., ZHANG, Y., AND YANG, Y. 1997b. A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems. In *Proceedings of the ACM International Conference on Supporting Group Work* (Phoenix, Ariz., Nov.). 425–434.

SUN, C., YANG, Y., ZHANG, Y., AND CHEN, D. 1996a. A consistency model and supporting schemes for real-time cooperative editing systems. In *Proceedings of the 19th Australian Computer Science Conference* (Melbourne, Jan.). 582–591.

SUN, C., YANG, Y., ZHANG, Y., AND CHEN, D. 1996b. Distributed concurrency control in real-time cooperative editing systems. In *Proceedings of the Asian Computing Science Conference* (Singapore, Dec.). Lecture Notes in Computer Science, vol. 1179. Springer-Verlag, 84–95.

VALDES, R. 1993. Text editors: Algorithms and architectures, not much theory, but a lot of practice. *Dr. Dobb's J.,* 38–43.

ZHANG, Y. AND YANG, Y. 1994. On operation synchronization in cooperative editing environments. *IFIP Trans. A-54 Bus. Process Reeng.,* 635–644.

Received August 1997; revised January 1998; accepted January 1998