A Generic Operation Transformation Scheme for Consistency Maintenance in Real-time Cooperative Editing Systems

Chengzheng Sun
School of Computing & Information Technology
Griffith University
Brisbane, Qld 4111, Australia
scz@cit.gu.edu.au

Xiahua Jia
Department of Computer Science
City University of Hong Kong
Kowloon, Hong Kong
jia@cs.cityu.edu.hk

Yanchun Zhang
Department of Mathematics & Computing
University of Southern Queensland
Toowoomba, Qld 4350, Australia
yan@usq.edu.au

Yun Yang
School of Computing & Mathematics
Deakin University
Geelong, Vic 3217, Australia
yun@deakin.edu.au

Abstract – *In real-time cooperative editing systems, independent operations on any part of the shared document may be generated from multiple cooperating sites. It is very important and technically challenging to ensure that the effect of executing an operation at remote sites, in the presence of concurrent execution of independent operations, achieves the same effect as executing this operation at the local site at the time of its generation, thus preserving its intention and maintaining system consistency. In this paper, we investigate the technical issues involved in preserving intentions of concurrent operations, explain the reasons why traditional serialization-based concurrency control strategies and existing operational transformation strategies failed to solve these problems, and propose a generic operation transformation scheme for intention preservation and consistency maintenance in real-time cooperative editing systems. The proposed scheme has been implemented in an Internet-based prototype REDUCE (REal-time Distributed Unconstrained Cooperative Editing) system.*

Keywords: intention preservation, consistency maintenance, distributed computing, cooperative editing, CSCW.

Appeared in the Proceedings of International ACM SIGGROUP Conference on Supporting Group Work, pp.425 -434, Phoenix, Arizona, USA, Nov. 16-19, 1997.

I. Introduction

Cooperative editing systems allows physically dispersed people to view and edit shared textual/graphical/multimedia documents at the same time [2, 8, 9]. They are very useful facilities in the rapidly expanding area of CSCW (Computer-Supported Cooperative Work) applications [3]. The goal of our research is to investigate, design and implement cooperative editing systems with the following characteristics: (1) *Real-time* requirements – the response to local user actions is quick (idealy as quick as a single-user editor) and the latency for remote user actions is low (determined by external communication latency only). (2) *Distributed* environments – cooperating users may reside on different machines connected by different communication networks. (3) *Unconstrained collaboration* – multiple users may concurrently and freely edit any part of the document at any time, as advocated in [2, 8, 9].

To achieve good responsiveness and unconstrained collaboration, we have adopted a replicated architecture regarding the storage for shared documents: the shared documents are replicated at the local storage of each participating site, so editing operations (updates only) are first performed at local sites and then propagated to remote sites. One of the most significant challenges in designing and implementing real-time cooperative editing systems with a replicated architecture is concurrency control to maintain consistency of the replicated documents under the constraints of a short response time, a short notification time, and unconstrained collaboration in a distributed environment. To illustrate, consider a scenario in a cooperative editing system with three cooperating sites, as shown in Fig. 1. Suppose that an opera-

tion is executed on the local replicate 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.

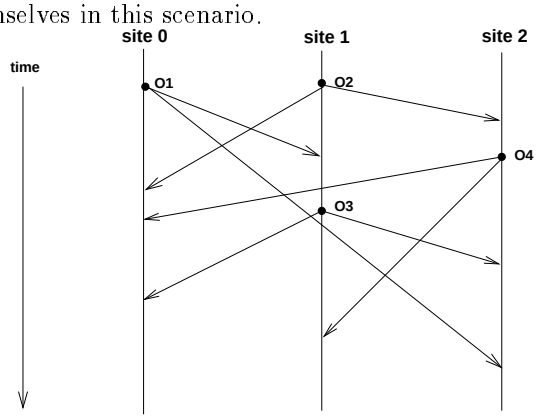

Fig. 1. A scenario of a real-time cooperative editing session.

First, operations may arrive and be executed at different sites in different orders, resulting in *divergent final results*. As shown in Fig. 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. This inconsistency problem can be solved by a convergence scheme [5, 9], or any serialization protocol [1, 6], which ensures the final result is the same as if all operations were executed in the same total order at all sites.

Secondly, due to the nondeterministic communication latency, operations may be executed out of their natural *cause-effect* order. As shown in Fig. 1, operation O_3 is generated after the arrival of O_1 at site 1, so O_3 may be *dependent* on O_1 (See Section II for a definition about *dependency*). However, since O_3 arrives (and is executed) before O_1 at site 2, the user at site 2 could be confused by observing the *effect* in O_3 before observing the *cause* in O_1. For 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 Tuesday in November ?" into the shared text document,[1] 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*. Out of causal order execution should be prohibited for applications where a synchronized interaction among multiple users is required. This *causality violation* problem can be solved by selectively[2] delaying the execution of some operations to enforce a causally ordered execution based on state vector timestamps [2, 9].

Thirdly, 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 Fig. 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 II for a 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 II for a definition about *intention*). For example, assume the shared document initially contains the following sequence of characters: "ABCDE". Suppose $O_1 = Insert[\text{"12"}, 1]$, 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 also violates the intention of O_2 since characters "CD", which were intended to be deleted, are still present in the final text. It should be pointed out that even if a convergence scheme or serialization-based protocol was used to ensure that all sites have an identical result "A1CDE", but this identical result is still inconsistent with the intentions of both O_1 and O_2. Unlike the divergent final results and causality violation problems, which are all related to the execution order of operations and could be fixed by properly re-scheduling the operations, the intention violation problem cannot be fixed by any serialization protocol if operations were always executed in their original forms.

It should be pointed out that the above three identified 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 divergent final results problem. Although the issue of consistency maintenance has attracted a great deal of attention in the areas of CSCW and distributed computing [2, 4, 5, 8], the nature and the complications of the intention violation problem have not been well understood, and the issue of preserving intentions of operations has often been mixed with the issue of ensuring convergence of the final results, which we believe

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

is one of the major reasons why some existing systems failed to *correctly* solve these problems (see Section VI for discussions of related work). In this paper, a novel and integrated approach to correctly solving the intention violation problem (in combination with the problems of divergent final results and causality violation) is proposed.

Apart from the *syntactic* inconsistency problems identified above, there are still other *semantic* inconsistency problems. For example, suppose a shared document contains the text:

"There will be student here."

In this text there is an English grammar 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 as presented in this paper). 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 consistent (since all sites have the same document contents and the syntactic editing effects of all operations are achieved), but semantically inconsistent (since there is still a grammar error in it). In other words, the system is able to ensure the plain *strings* be inserted 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 concurrency control mechanisms without the intervention of the people in collaboration. To the best of our knowledge, none of existing cooperative (and even single-user) editing systems has attempted to maintain semantical consistency automatically. The consistency maintenance algorithms to be discussed this paper will not address the semantic consistency maintenance problem either.

The organization of this paper is as follows. First some previous results on a consistency model and concurrency control schemes will be briefly described in Section II. Next, the basic issues and complications involved in resolving the intention violation problem will be analyzed in Section III. Then, a novel intention-preserving scheme will be presented in Section IV, and it will be integrated with the convergence scheme to provide a solution to both the intention-violation and the divergent final results problems in Section V. Our work will be compared with other related approaches in Section VI. The major contributions and future work of our research will be summarized in Section VII.

II. Previous work

In this section, some relevant results obtained in previous work are briefly introduced. For more detailed discussion of them, the reader is referred to [9, 10].

A. A consistency model

Following Lamport [6], 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 "$\rightarrow$"
Given two operations O_a and O_b, generated at sites i and j, then $O_a \rightarrow O_b$, *iff*:

1. $i = j$ and the generation of O_a *happened before* the generation of O_b, or
2. $i \neq j$ and the execution of O_a at site j *happened before* the generation of O_b, or
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 *iff* $O_a \rightarrow O_b$.
2. O_a and O_b are said to be *independent* (or *concurrent*) *iff* 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/independency 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 $O_2 \rightarrow O_4$.

In [9], the notion of intention of operations was first introduced and a consistency model was defined as follows.

Definition 3: Intention of an operation
Given an operation O, the intention of O is the execution effect which could be achieved by applying O on the document state from which O was 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 all sites have executed the same set of operations, the copies of the shared document at all sites 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,
   (a) both the local and remote execution effects of O equal to the intention of O, and
   (b) if there exists an operation O_x such that $O_x \parallel O$, then the execution effect of O_x does not interfere

with the execution effect of O, and vice versa. □

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 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, regardless the concurrent execution of independent operations and the nondeterministic latency of propagating local operations to remote sites. The consistency model effectively specifies what assurance a cooperative editing system gives to its users and what properties the underlying concurrency control schemes must support.

B. Achieving causality-preservation and convergence

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 [2, 9]. Let N be the number of cooperating sites in the system. Assume that sites are identified by integers $0, ..., N-1$. Each site maintains an SV with N components. Initially, $SV[i] := 0$, for all $i \in \{0, ..., N-1\}$. 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 $i \in \{0, 1, ..., N-1\}$ 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 the 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.

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.[4] To achieve convergence in the presence of different execution order of independent operations, we first define a total ordering relation among operations as follows.

Definition 6: Total ordering relation "⇒"

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 $O_a \Rightarrow O_b$, *iff*:

1. $sum(SV_{O_a}) < sum(SV_{O_b})$, or
2. $i < j$ when $sum(SV_{O_a}) = sum(SV_{O_b})$,

where $sum(SV) = \sum_{i=0}^{N-1} SV[i]$. □

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 [9].

Algorithm 1: The undo/do/redo scheme

When a new operation O_{new} is causally-ready, the following steps are executed:

1. **Undo** all operations in *HB* which totally *follow* O_{new} to restore the document to the state before their execution.
2. **Do** O_{new} and save it in *HB*.
3. **Redo** all operations that were undone from *HB*. □

Obviously, when all sites have executed the same set of operations under the undo/do/redo scheme, the editing effect will be the same as if all operations were executed in the total order "⇒" at all sites, thus ensuring the convergence property.

III. Basic issues and complications in achieving intention-preservation

Achieving intention-preservation is much harder than achieving convergence and causality-preservation. This is because that the intention violation problem cannot be resolved by just re-scheduling 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.

To transform an operation against another operation, an *inclusion* transformation strategy (see Section IV-A) 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 system 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”, $O_1 = Insert[\text{“12”}, 1]$, and $O_2 = Insert[\text{“23”}, 0]$, After the execution of O_2 at site 2, the document state becomes “23ABCDE”. Suppose $O_4 = Insert[\text{“45”}, 2]$, 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 $O_4' = Insert[\text{“45”}, 4]$. 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 IV-A) 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.

Life would have been 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 scheme 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. This paper will focus on the generic intention-preserving scheme, and the design of the concrete inclusion/exclusion transformation algorithms for a text editing system can be found in another paper [12].

IV. 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 $HB = [EO_1, EO_2, ..., EO_n]$. 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 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 $L_1 + L_2 = [EO_1, EO_2, EO_3]$.

A. Pre-/post-conditions for transformation functions

Conceptually, an operation O is associated with a *context*, denoted as CT_O, which is the list of operations that have affected the document state from which O is generated. The significance of context is that the meaning 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. The original context of an operation does not change unless it is explicitly transformed. For specifying pre-/post-conditions for 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., $O_a \sqcup O_b$, *iff* $CT_{O_a} = CT_{O_b}$. □

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 Fig. 1, $O_2 \parallel O_1$, and $O_1 \parallel O_4$, but $O_2 \rightarrow O_4$.

Definition 8: Context immediate 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 immediate preceding* O_b, i.e., $O_a \mapsto O_b$, *iff* $CT_{O_b} = CT_{O_a} + [O_a]$. □

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 immediate preceding relation "$\mapsto$" is different from the dependence (i.e. the causal ordering) relation "$\rightarrow$": the dependence relation is transitive whereas the context immediate preceding relation is not transitive by definition.

The current list of operations in HB at any site determine the current document state at that site and define an *environment* for executing new operations. Obviously, the execution environment 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 environment at the destination site, it can be executed as it is since the document state determined by the current environment is the same as the document state determined by the operation's original context and the operation's intention is automatically guaranteed. However, when a causally-ready operation has its original context being different from the current environment (due to preceding executions of independent operations), this operation needs to be transformed before its execution in the current environment 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. Postconditions for output: (1) $O_b \mapsto O'_a$, and (2) the execution of O'_a in the environment of $HB = CT_{O'_a}$ achieves the same editing effect as the execution of O_a in the environment of $HB = CT_{O_a}$. □

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: $O_b \sqcup O'_a$. □

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 $O_a \sqcup O_b$ and $O'_a = IT(O_a, O_b)$, then, it must be that $O_a = ET(O'_a, O_b)$.
2. If $O_b \mapsto O_a$ and $O'_a = ET(O_a, O_b)$, then, it must be that $O_a = IT(O'_a, O_b)$. □

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 ensure 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 IV-B. 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 and ET 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 string-wise primitive operations – *Insert* and *Delete*, and the corresponding LIT and LET functions capable of handling exceptional cases, such as lost information saving/recovery for ensuring reversibility, and relatively addressing for undefined ranges[2], can be found in [12].

B. The generic operation transformation control scheme

Based on the concept of context and the specifications of inclusion and exclusion transformation functions, the *Generic Operation Transformation* (GOT) control scheme is proposed in this section.

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

1. $EO_1 \Rightarrow EO_2 \Rightarrow ... \Rightarrow EO_m \Rightarrow O_{new}$, and
2. $EO_1 \mapsto EO_2 \mapsto ... \mapsto EO_m$.

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

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

Let us start with 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 operations in HB which are independent 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 into the context of O_{new}.

Suppose $HB = [EO_1, ..., EO_k, ..., EO_m]$, where EO_k is the oldest operation in HB which is independent of O_{new}, so that all operations in the range of $HB[1, k-1]$ are causally preceding 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}, i.e., $CT_{O_{new}} = HB[1, k-1] + EOL'$, where EOL' is a list of operations executed after the execution of EO_{k-1} but before the generation of O_{new}.

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

The complication comes when there is a mixture of independent and dependent operations in the range of $HB[k, m]$. Let $EOL = [EO_{c_1}, ..., EO_{c_r}]$, which is the list of operations in the range of $HB[k+1, m]$ and they are causally preceding O_{new}. Under this circumstance, EOL' must be equal to $[EO'_{c_1}, ..., EO'_{c_r}]$, where EO'_{c_i} is the corresponding form of EO_{c_i} at the time of

[2] Generally, when O_b is causally preceding O_a, and (1) O_b is an insert operation, and (2) O_a inserts a string or delete some characters within the string inserted by O_b, the application of exclusion transformation on O_a against O_b will result in O_a's operation range undefined.

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

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 non-empty list of 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 $EO_{c_i} \mapsto EO_{c_{i+1}}$, 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. $TO^4 := LET(EO_{c_2}, HB[k,c_2-1]^{-1})$;
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. $EO'_{c_i} := LIT(TO, [EO'_{c_1}, ..., EO'_{c_{i-1}}])$.

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. $EO_{new} := LIT(O'_{new}, HB[k,m])$.

[4] TO stands for Temporary Operation – a notation used to represent an intermediate result during transformation.

Based on the above discussion and reasoning, the GOT control scheme is derived below.

Algorithm 2: The GOT control scheme

Given a new causally-ready operation O_{new}, and $HB = [EO_1, EO_2, ..., EO_m]$. The following steps are executed to obtain EO_{new}:

1. Scanning the 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 $EO_{new} := O_{new}$.
2. Otherwise, search the range of $HB[k+1,m]$ to find all operations which are causally preceding O_{new}, and let EOL denote these operations. If $EOL = []$, then $EO_{new} := LIT(O_{new}, HB[k,m])$.
3. Otherwise, suppose $EOL = [EO_{c_1}, ..., EO_{c_r}]$, the following steps are executed:
   (a) Get $EOL' = [EO'_{c_1}, ..., EO'_{c_r}]$ as follows:
      i. $EO'_{c_1} := LET(EO_{c_1}, HB[k,c_1-1]^{-1})$.
      ii. For $2 \leq i \leq r$,
         $TO := LET(EO_{c_i}, HB[k,c_i-1]^{-1})$;
         $EO'_{c_i} := LIT(TO, [EO'_{c_1}, ..., EO'_{c_{i-1}}])$.
   (b) $O'_{new} := LET(O_{new}, EOL'^{-1})$.
   (c) $EO_{new} := LIT(O'_{new}, HB[k,m])$. □

According to the above analysis and description of the GOT control scheme, we know that the preconditions required by the transformation functions are always guaranteed by the GOT control scheme. Therefore, if the postconditions are always ensured by the transformation functions, then EO_{new} obtained by the GOT control scheme will have the following property: the execution of EO_{new} in the environment of $HB = CT_{EO_{new}}$ will achieve the same effect as the execution of O_{new} in the environment of $HB = CT_{O_{new}}$, thus preserving the intention of O_{new}. Finally, it is worth pointing out that the GOT control scheme works solely on a linear HB, and no additional data structures need to be maintained for deriving context-based relationship among operations.

V. Integrating the GOT control scheme with the undo/do/redo scheme

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

Let O_{new} be a new causally-ready operation, and $HB = [EO_1, ..., EO_m, ..., EO_n]$, 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 executions. Then, O_{new} is transformed into EO_{new} by the GOT control

scheme such that $CT_{EO_{new}} = HB[1,m]$, and executed. Finally, all undone operations are transformed and redone one by one to take into account of 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 also $CT_{EO_{new}} = HB[1,m]$, 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. $TO := LET(EO_{m+i}, HB[m+1, m+i-1]^{-1})$;
2. $EO'_{m+i} := LIT(TO, [EO_{new}, EO'_{m+1}, ..., EO'_{m+i-1}])$.

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 environment of $HB = CT_{EO'_{m+i}}$ will achieve the same effect as the execution of EO_{m+i} in the environment of $HB = CT_{EO_{m+i}}$.

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

Algorithm 3: The undo/transform-do/transform-redo scheme

Given a new causally-ready operation O_{new}, and $HB = [EO_1, ..., EO_m, ..., EO_n]$, 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 scheme. Then, **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) $EO'_{m+i} := LIT(TO, [EO_{new}, EO'_{m+1}, ..., EO'_{m+i-1}])$.

   Then, **redo** $EO'_{m+1}, EO'_{m+2}, ..., EO'_n$, sequentially.

After the execution of the above steps, the contents of the history buffer becomes: $HB = [EO_1, ..., EO_m, EO_{new}, EO'_{m+1}, ..., EO'_n]$. □

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, resulting in the same document state, hence convergence is ensured. Moreover, each executed operation in *HB* is obtained by either the GOT control scheme or by step 3 of the undo/transform-do/transform-redo scheme, hence intention-preservation is ensured.

An example – To illustrate how the undo/transform-do/transform-redo scheme works, consider the scenario shown in Fig 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.

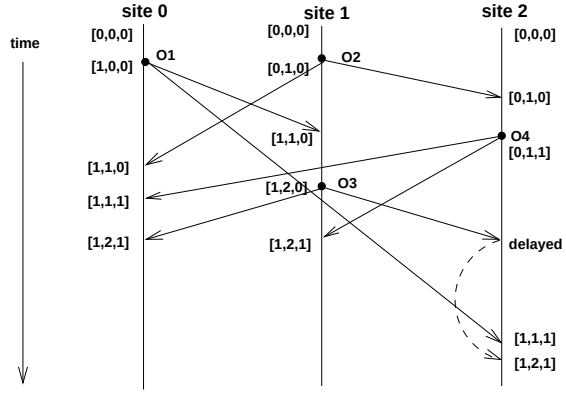

Fig. 2. A scenario of a real-time cooperative editing session augmented with both the causality-preserving scheme and the undo/transform-do/transform-redo scheme.

The sequence of events happening at each site are 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 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.
   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 become: $EO_2 := IT(O_2, EO_1)$, 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 to be 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.

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

VI. Discussions of related work

Traditional serialization-based concurrency control strategies have found successful application in database management systems [1], and they can also be applied to achieve convergence in real-time cooperative editing systems, such as the undo/do/redo scheme in our approach. However, without changing the operations' original forms, intentions of concurrent operations in real-time cooperative editing systems cannot be achieved by serializing concurrent operations in any order.

Using operational transformation to maintain consistency in real-time cooperative editing systems was pioneered by the GROVE system [2], to which our work is most closely related. The GROVE system used the distributed operational transformation (dOPT) algorithm for transforming independent operations. Essentially, the dOPT algorithm adopted an inclusion transformation strategy. Their basic idea was to perform the inclusion transformation on each causally-ready operation against all precedingly executed independent operations in the Log (i.e., the history buffer in our terminology) in the order from the oldest to the most recent. 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 were 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 scheme 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 scheme 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 proposed a different approach, called the adOPTed algorithm, to solve the problem [8]. The adOPTed algorithm added to the original dOPT algorithm a multi-dimensional in-

teraction graph, which keeps track of all valid paths of transforming operations, and a double recursive function (similar in functionality to our GOT control algorithm), which determines which operations should be applied the *L-Transformation* (similar to our *Inclusion Transformation*) against which others. If the *L-Transformation* functions could always satisfy the properties specified in [8], 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 paper. 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 forms) 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). Moreover, the adOPTed algorithm uses a serial number k, in addition to the state vector, to identify each operation, but our algorithm does not need this k.

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[6]. 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 nature of transformation functions, it is difficult to verify whether a given *L-Transformation* function satisfies the properties required in [8]. 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: $O_1 = Insert[\text{“1”}, 2]$, $O_2 = Insert[\text{“2”}, 1]$ and $O_3 = Delete[1, 1]$, generated from the same initial document state “ABC” at site 1, 2, and 3 respectively. According to the adOPTed algorithm and the *L-Transformation* functions for text editing given in [8], we constructed the interaction graph for these operations, shown in Fig. 3. 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 ! One may be tempted to fix this problem by reversing the following rule used in [8]: 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.

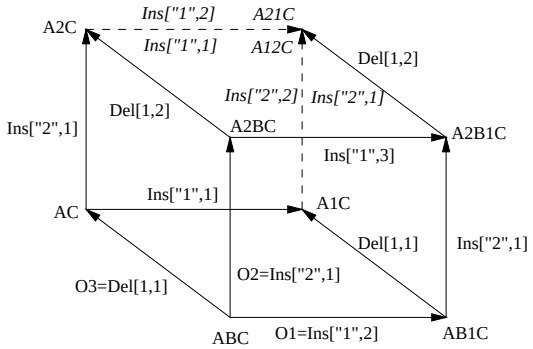

Fig. 3. An interaction graph with ambiguously labeled edges and vertices.

Finally, the adOPTed algorithm has been applied to support user-initiated *undo* operations, similar to that in [7]. We have also investigated this issue and found the inclusion and exclusion transformation functions for consistency maintenance can be directly used to support

[6] In [5], another undo/redo-based scheme was proposed to ensure convergence of cooperative graphical editing systems. However, the issues of causality-preservation and intention-preservation were not addressed by that scheme.

collaborative undo in unconstrained cooperative environments, in which no lock is used. Our results on this topic will be reported in a forthcoming paper.

VII. Conclusions

In this paper, we have proposed and discussed in detail a generic operation transformation scheme for intention preservation and consistency maintenance in real-time cooperative editing systems. Unique contributions made by this work include: the specification of the pre-/post-conditions for a pair of reversible inclusion and exclusion transformation functions, the design of the GOT control scheme, and the integration of the GOT control scheme with the undo/do/redo scheme to achieve both intention-preservation and convergence. The algorithms presented in this paper have been implemented in an Internet-based prototype REDUCE (REal-time Distributed Unconstrained Cooperative Editing) system using programming language Java [11]. The current prototype system has been developed mainly to test the feasibility of our approach and to experiment with alternative strategies and algorithms. Efforts are being directed towards building a more useful and robust system, which will be used by external users in real application contexts to evaluate the research results from end-users' perspective.

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 formalizing the identified problems and the proposed solutions has played a crucial role in enabling us to understand the nature and complexity of the intention-preservation problem, to design and verify the algorithms to solve the problem, and to efficiently implement the algorithms. In fact, our prototype implementation has directly followed the algorithms formally described in this paper. We will continue to apply this experimental and theoretical combined approach in our ongoing work on the models and techniques for supporting user-initiated undo operations and group-awareness in highly concurrent and unconstrained cooperative environments.

Acknowledgements

The work reported in this paper has been partially supported by an *NCGSS Grant* from Griffith University and a *Strategic Research Grant* (Ref No:7000641) from City University of Hong Kong.

References

[1] P. Bernstein, N. Goodman, and V. Hadzilacos: *Concurrency control and recovery in database systems,* Addison-Wesley, 1987.

[2] C. A. Ellis and S. J. Gibbs: "Concurrency control in groupware systems," In *Proc. of ACM SIGMOD Conference on Management of Data*, pp.399-407, 1989.

[3] C. A. Ellis, S. J. Gibbs, and G. L. Rein: "Groupware: some issues and experiences," *CACM 34(1)*, pp.39-58, Jan. 1991.

[4] S. Greenberg and D. Marwood:"Real time groupware as a distributed system: concurrency control and its effect on the interface," In *Proc. of ACM Conference on Computer Supported Cooperative Work,* pp. 207-217, Nov. 1994.

[5] A. Karsenty and M. Beaudouin-Lafon: "An algorithm for distributed groupware applications," In *Proc. of 13th International Conference on Distributed Computing Systems,* pp. 195-202, May 1993.

[6] L. Lamport: "Time, clocks, and the ordering of events in a distributed system," *CACM 21(7)*, pp.558-565, 1978.

[7] A. Prakash and M. Knister: "A framework for undoing actions in collaborative systems," *ACM Trans. on Computer-Human Interaction*, 4(1), pp.295-330, 1994.

[8] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenbauser:"An integrating, transformation-oriented approach to concurrency control and undo in group editors," In *Proc. of ACM Conference on Computer Supported Cooperative Work,* pp 288-297, 1996.

[9] C. Sun, Y.Yang, Y. Zhang, and D. Chen: "A consistency model and supporting schemes for real-time cooperative editing systems," In *Proc. of the 19th Australian Computer Science Conference,* pp. 582-591, Melbourne, Jan. 1996.

[10] C. Sun, Y.Yang, Y. Zhang, and D. Chen:"Distributed concurrency control in real-time cooperative editing systems," *Proc. of the 1996 Asian Computing Science Conference,* Lecture Notes in Computer Science, #1179, Springer-Verlag, Singapore, pp.84-95, Dec. 1996.

[11] C. Sun, X. Jia, Y. Yang, and Y. Zhang:"REDUCE: a prototypical cooperative editing system," In *Proceedings of the 7th International Conference on Human-Computer Interaction,* San Francisco, pp.89-92, Aug. 1997

[12] C. Sun, et al:"Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems," CIT-97-07. School of Computing & Information Technology, Griffith University, 1997.