Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems

Chengzheng Sun David Chen
School of Computing & Information Technology
Griffith University
Brisbane, Qld 4111, Australia
{C.Sun,D.Chen}@cit.gu.edu.au

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

Abstract. *Operation transformation has been recognized as a promising approach to intention preservation and consistency maintenance in cooperative editing systems. To deal with the complications caused by the fact that independent operations may come from different document states, we propose a pair of mutually reversible inclusion and exclusion transformation functions, which can be used to effectively include/exclude the impact of one operation into/from another operation so that the pre-/post-conditions of transformation functions can be satisfied and correct transformation results can be achieved. The technical issues and strategies in the design of inclusion and exclusion transformation functions for string-wise operations in cooperative text editing systems are discussed in detail in this paper* [1]

Keywords: intention preservation, cooperative editing, CSCW.

1 Introduction

Cooperative editing systems allow physically dispersed people to view and edit shared textual/graphical/multimedia documents at the same time[1, 2, 3, 4]. They are very useful facilities in the rapidly expanding area of CSCW (Computer-Supported Cooperative Work) applications. 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 (ideally 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 parts of the document at any time, as advocated in [1, 3, 4].

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

[1] To appear in *Proc. of The 21st Australasian Computer Science Conference*, 1998.

then propagated to remote sites. Three inconsistency problems have been identified in a real-time cooperative system with a replicated architecture in the absence of proper concurrency control [1, 4, 6]. First, operations may arrive and be executed at different sites in different orders, resulting in *divergent final results*. This inconsistency problem can be solved by any serialization protocol, which ensures the final result is the same as if all operations were executed in the same total order at all sites [4]. Secondly, operations may be executed out of their natural *cause-effect* order, which could cause confusing to the system users. 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 [1, 3, 4]. 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. The *intended effect* or *intention* of an operation is defined as the editing effect which can be achieved by executing this operation on the document state from which it was generated [4, 6].

To illustrate the intention-violation problem, consider the following cooperative text editing scenario: the initial document contains a string: "ABCDE", and the users at site 0 and site 1 concurrently issue two operations: $O_1 = Insert[\text{"12"}, 1]$, 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". Each operation is executed immediately at the local site after its generation and then propagated to the remote site. After the execution of these two operations, the *intention-preserved* result (at both sites) should be: "A12BE". However, the actual result at site 0, obtained by executing O_1 followed by 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 serialization-based protocol was used to ensure that both 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 generally be fixed by any serialization protocol if operations were always executed in their original forms. Pioneered by the GROVE system [1], operational transformation has been used to adjust the parameters of one operation according to the effects of other executed independent operations so that the execution of the transformed operation on the new document state can achieve the same effect as executing the original operation on the original document state. For instance, in the previous example, when O_2 arrives at site 0, it can be transformed into $O_2' = Delete[2, 4]$. The execution of O_2' at site 0 will result in the intention-preserved document state: "A12BE". This transformation strategy is called *inclusion* transformation [6], since it 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. The correctness of this inclusion transformation relies on the precondition that both O_a and O_b are generated from the same document state, so their position parameters are comparable and can be used to derive a proper adjustment to O_a. The GROVE system used a distributed operational transformation (dOPT) algorithm to apply a kind of inclusion transformation function on each operation against all executed independent operations in the history buffer in the order from the oldest to most recent.

Unfortunately, the dOPT algorithm did not work if independent operations were generated from different document states [6]. To achieve correct results in transformation-based cooperative editing systems, we propose an additional *exclusion transformation*, which transforms an operation O_a against another operation O_b in such a way that the impact of O_b is effectively excluded from O_a. Based on the inclusion and exclusion transformation strategies, we proposed a *Generic Operation Transformation* (GOT) control algorithm in [6], which is responsible for determining which operations should be applied with inclusion/exclusion transformation against which others and for ensuring the preconditions for transformation functions are always satisfied. In this paper, we will focus on the design of inclusion and exclusion transformation functions for text editing systems. For the GOT control algorithm, the reader is refereed to [6].

In contrast to the *character-wise* text editing transformation algorithms proposed in [1, 3], our inclusion and exclusion transformation functions are *string-wise*. The extension from character-wise transformation to string-wise transformation is important and worthwhile since string-wise transformation is more general and can greatly reduce the number of transformations and communications. This extension is also nontrivial since string-wise transformation involves quite a few complications not occurring in character-wise transformation, as will be shown in this paper. To our knowledge, there has been no published work on string-wise transformation algorithms for the same purpose. We strongly believe that an in-depth and detailed investigation of transformation algorithms deserve a serious attention because this level of investigation enables us to better understand the intrinsic interactions (in the form of pre-/post-conditions) between transformation algorithms and higher level control algorithms. Without a deep understanding of the interactions between algorithms at the lower and higher levels, it would not have been possible for us to detect the hidden flaws in existing systems and to devise correct solutions at both the lower and higher levels [6].

The rest of the paper is organized as follows: First, the pre-/post-conditions for the inclusion and exclusion transformation are specified in Section 2. Next, the application environment in which the inclusion and exclusion transformation algorithms are designed are described in Section 3. Then, the definitions and strategies in the design of the inclusion and exclusion transformation functions for two primitive editing operations – *Insert* and *Delete* – are discussed in Sections 4 and 5, respectively. Finally, the major contributions reported in this paper are summarized in Section 6.

2 Specification of pre-/post-conditions

To support the operation transformation, each cooperating site maintains a linear *history buffer (HB)* for saving executed operations at each site. The document state at any instance of time can be determined by sequentially executing operations in HB on the initial document state. Conceptually, an operation O is associated with a *context*, denoted as CT_O, which is the list of operations executed before O is generated. 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 context of an operation can be changed by explicitly applying the following primitive transformation functions. For specifying pre-/post-conditions of transformation functions, two context-based relations among operations are defined below.

Definition 1 *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* $CT_{O_a} = CT_{O_b}$.

Definition 2 *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., $O_a \mapsto O_b$, *iff* $CT_{O_b} = CT_{O_a} + [O_a]$ (where operator "+"is the concatenation of two lists).

To include/exclude operation O_b into/from the context of O_a, the *inclusion/exclusion* transformation function $IT(O_a, O_b)/ET(O_a, O_b)$ 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 environment of $HB = CT_{O'_a}$ is the same as O_a's execution effect in the environment of $HB = CT_{O_a}$.

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 environment of $HB = CT_{O'_a}$ is the same as O_a's execution effect in the environment of $HB = CT_{O_a}$.

In addition to the postconditions, the two primitive transformation functions are required to be *reversible* in the following sense: (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 $O_a = IT(ET(O_a, O_b), O_b)$.

Preconditions are required by the 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 GOT control algorithm [6]. 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 definitions of the concrete inclusion and exclusion transformation functions.

3 Application environment

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 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 [7].

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.

It has been shown that practical text editing systems, such as *vi* and *Emacs*, can be implemented on top of these two primitives [2, 3, 7].

Criteria for verifying intention-preserved effects 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 environment of $HB = CT_{O'}$ achieves the same effect as the execution of O in the environment of $HB = CT_O$ 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 entirety appears in the document after the execution of 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 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 $R_O = [C_P, C_{P+1}, ..., C_{P+N}]$ be the list of characters within the deleting range of O in the document state determined by CT_O, and $R_{O'} = [C_{P'}, C_{P'+1}, ..., C_{P'+N'}]$ 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'. (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 sense that: (1) An *Insert* operation may never insert a string into the middle of another string inserted by an independent operation. (2) A *Delete* operation may never delete characters inserted by independent operations.

Notations To facilitate the definition of transformation functions, the following notations are introduced: (1) $T(O)$: the type of operation O, i.e., *Insert/Delete*. (2) $P(O)$: the position parameter of O. (3) $L(O)$: the length of O. For *Insert*, it is the length of the string to be inserted. For *Delete*, it is the number of characters to be deleted. (4) $S(O)$: the string of the *Insert* operation O.

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.

4 Inclusion transformation

The function $IT(O_a, O_b)$ is defined to perform the inclusion transformation on O_a against O_b. After checking some special cases (to be explained in Section 5), one of the four sub-functions is called to do the real transformation, according to the operation types (*Insert/Delete*) of O_a and O_b. The definitions of $IT(O_a, O_b)$ and its four sub-functions are given in Figure 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 right-most 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 $L(O_b)$. Otherwise, it must be that the intended position of O_a falls in the

Function 1 *IT(O_a, O_b)*
{
  if $Check\_RA(O_a)$
    if $Check\_BO(O_a, O_b)$ $O'_a := Convert\_AA(O_a, O_b)$;
    else $O'_a := O_a$;
  else if ($T(O_a) = Insert$ **and** $T(O_b) = Insert$) $O'_a := IT\_II(O_a, O_b)$;
  else if ($T(O_a) = Insert$ **and** $T(O_b) = Delete$) $O'_a := IT\_ID(O_a, O_b)$;
  else if ($T(O_a) = Delete$ **and** $T(O_b) = Insert$) $O'_a := IT\_DI(O_a, O_b)$;
  else /* ($T(O_a) = Delete$ **and** $T(O_b) = Delete$) */ $O'_a := IT\_DD(O_a, O_b)$;
  return O'_a;
}

Function 2 *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 3 *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 4 *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 5 *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. 1. Inclusion transformation functions.

deleting range of O_b. In this case, the new inserting position should be $P(O_b)$, according to the verification criteria.

It should be pointed out that when two independent operations insert two strings at the same position, the two strings will appear in the document state as if they were inserted in some total order, which is enforced by a high level control scheme(i.e., the undo/do/redo scheme in [6]). In case that one string, e.g., "AB", is a prefix of the other string, e.g., "ABCD", the combined inserting effects will be "ABABCD" (provided that "ABCD" was inserted before "AB" is inserted), according to the definition of Function 2. This non-merged strategy is the same as the strategy used in [3], but different from that in [1].

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 sub-operations, 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 $O'_{a1} \sqcup O'_{a2}$.

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 the 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, 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 5).

It is worth pointing out that according to the definition of $IT\_DD(O_a, O_b)$, if O_a and O_b have overlapped deleting ranges, the combined deleting range will be the union of the two individual deleting ranges – an effect which could not be achieved by serializing independent operations in any order.

5 Exclusion transformation

The function $ET(O_a, O_b)$ is defined to perform the exclusion transformation on O_a against O_b. After checking some special cases (to be explained in Section 5), one of the four sub-functions is called to do the real transformation, according to the operation types (*Insert/Delete*) of O_a and O_b. The definitions of $ET(O_a, O_b)$ and its four sub-functions are given in Figure 2.

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 the 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, and (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. Else if $P(O_a) \geq (P(O_b) + L(O_b))$, which means that the position of O_a goes beyond the right-most 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 any more since undoing O_b will result in O_a's operation range undefined.

Relative addressing for undefined ranges Generally, when O_b is executed before the generation of O_a (i.e., O_b is causally preceding O_a[6]), 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, undoing O_b will result in O_a's operation range undefined.

The technique used to deal 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 parameter of the

Function 6 *ET(O_a, O_b)*
{
  if $Check\_RA(O_a)$ $O'_a := O_a$;
  else if ($T(O_a) = Insert$ **and** $T(O_b) = Insert$) $O'_a := ET\_II(O_a, O_b)$;
  else if ($T(O_a) = Insert$ **and** $T(O_b) = Delete$) $O'_a := ET\_ID(O_a, O_b)$;
  else if ($T(O_a) = Delete$ **and** $T(O_b) = Insert$) $O'_a := ET\_DI(O_a, O_b)$;
  else /*($T(O_a) = Delete$ **and** $T(O_b) = Delete$)*/ $O'_a := ET\_DD(O_a, O_b)$;
  return O'_a;
}

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. 2. Exclusion transformation functions.

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.

Once an operation has become relatively addressed, it will skip all subsequent transformations except the one against its base operation. As shown in Function 1, a utility routine $Check\_RA(O_a)$ is used to check whether O_a is relatively addressed. If yes, another utility routine $Check\_BO(O_a, O_b)$ is used to check whether O_b is O_a's base operation. If yes, then another utility routine $Convert\_AA(O_a, O_b)$ is used to convert O_a into an absolutely addressed operation O'_a. If, however, O_b is not O_a's base operation, then O_a is returned as O'_a without transformation. Also, as shown in Function 6, if O_a is relatively addressed, it is returned as O'_a without transformation. This strategy works because a relatively addressed operation is just an intermediate result of the top level GOT control scheme [6], and it will only be used for subsequent exclusion/inclusion transformations but never used for updating the document state.

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.

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 also some characters outside the string inserted by O_b, the outcome will consists 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: O_b is the base operation of O'_{a1}, and $O_b \sqcup O'_{a2}$.

6 Conclusions

In this paper, we have proposed and discussed in detail a pair of reversible inclusion and exclusion transformation algorithms for string-wise operations in cooperative text editing systems. Based on the specifications of the pre- and post-conditions for general transformation functions, we defined criteria for verifying intention-preserved editing effects and concrete transformation functions

for string-wise *Insert/Delete* operations. In addition to the basic strategies for normal transformation based on operations' parameters, we devised strategies to cope with special technical issues, such as lost information saving/recovery for ensuring reversibility, relatively addressing for handling undefined ranges, and operation splitting for handling split segments. The defined inclusion and exclusion transformation functions in this paper have been implemented in an Internet-based prototype REDUCE (REal-time Distributed Unconstrained Cooperative Editing) system using programming language Java [5]. Our work continues on various aspects of the REDUCE system, including the application of the inclusion/exclusion transformation functions for supporting user-initiated *undo* operations.

Although the transformation algorithms were designed in the context of cooperative text editing, they are actually quite general and potentially applicable in other cooperative editing environments, which allow concurrent inserting/deleting any sequence of objects with a linearly ordered relationship, such as a sequence of pages in a document/book, a sequence of slides in a seminar/lecture, a sequence of frames in a movie/video, etc.

Acknowledgments The work reported in this paper has been conducted in the context of the REDUCE project, which 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. The authors wish to thank Yun Yang and Yanchun Zhang, the other two members of our search team, for their contributions to the REDUCE project. Finally, the authors are very grateful to the anonymous referees for their valuable comments, which helped improve the final version of this paper.

References

1. 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.
2. M. Knister and A. Prakash: "Issues in the design of a toolkit for supporting multiple group editors," *Journal of the Usenix Association,* Vol.6, No.2, pp. 135-166, 1993.
3. 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 CSCW,* pp 288-297, 1996.
4. 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.
5. C. Sun, X. Jia, Y. Yang, and Y. Zhang: "REDUCE: a prototypical cooperative editing system," *Proceedings of the 7th International Conference on Human-Computer Interaction,* pp.89-92, San Francisco, Aug. 1997.
6. C. Sun, X. Jia, Y. Zhang, and Y. Yang: "A generic operation transformation scheme for consistency maintenance in real-time cooperative editing systems," To appear in *Proc. of International Conference on Supporting Group Work (GROUP'97)*, ACM Press, Phoenix, Arizona USA, Nov. 1997.
7. Ray Valdes: "Text editors: algorithms and architectures, not much theory, but a lot of practice," *Dr. Dobb's Journal,* pp. 38- 43, April 1993.