Computer Supported Cooperative Work (2010) 19:1–43
DOI 10.1007/s10606-009-9103-1
© Springer 2009

An Admissibility-Based Operational Transformation Framework for Collaborative Editing Systems

Du Li[1] & Rui Li[2]
[1]*Nokia Research Center, 955 Page Mill Road, Palo Alto, CA 94304, USA (E-mail: lidu008@gmail.com);*
[2]*Google, Inc., Building 43 - 171B, 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA (E-mail: ruili73@hotmail.com)*

Abstract. Operational transformation (OT) as a consistency control method has been well accepted in group editors. With OT, the users can edit any part of a shared document at any time and local responsiveness is not sensitive to communication latencies. However, established theoretical frameworks for developing OT algorithms either require transformation functions to work in all possible cases, which complicates the design of transformation functions, or include an under-formalized condition of intention preservation, which results in algorithms that cannot be formally proved and must be fixed over time to address newly discovered counterexamples. To address those limitations, this paper proposes an alternative framework, called admissibility-based transformation (ABT), that is theoretically based on formalized, provable correctness criteria and practically no longer requires transformation functions to work under all conditions. Compared to previous approaches, ABT simplifies the design and proofs of OT algorithms.

Key words: CSCW, collaboration, consistency control, group editor, operational transformation

1. Introduction

Group editors allow a group of human users to view and modify shared documents over a computer network. Example documents include software programs and documentation, web pages, online encyclopedia, and music scores (Bellini et al. 2002). To increase the productivity of human users, it is well understood that multi-user editors must be as responsive as single-user counterparts, and the users must be allowed to make contributions freely in parallel without blocking each other (Begole et al. 1999; Ellis et al. 1991; Hymes and Olson 1992; Sun et al. 1998).

Over the past decade, operational transformation (OT) (Ellis and Gibbs 1989) has been well established for concurrency control in group editors (Sun and Ellis 1998). The technique has been implemented in many products including CoWord (Sun et al. 2006), ACE, Gobby, SubEthaEdit, and most recently Google Wave. In an OT-based group editor, conceptually, the shared document is replicated at every site; local editing operations are allowed to execute as soon as they are generated; remote operations are transformed before execution to repair inconsistencies. As a result, local response time is not sensitive to communication

latencies and the users can modify any part of the shared data without being blocked for the sake of concurrency control. The use of OT improves effectiveness and productivity of collaboration in certain applications (Begole et al. 1999; Bellini et al. 2002).

Most existing OT algorithms are developed under two theoretical frameworks: (Ressel et al. 1996) and (Sun et al. 1998). The first framework (Ressel et al. 1996) requires that OT algorithms preserve causality and achieve convergence. In particular, transformation functions must satisfy two strong properties for achieving convergence, which together require that transforming any operation with any two sequences (of the same set of concurrent operations in different execution orders) always yield the same result. Unfortunately, this has turned out very difficult to achieve, as confirmed in the (Ressel et al. 1996; Sun et al. 1998 Li and Li 2008a; Oster et al. 2005a). The second framework (Sun et al. 1998) futher includes a new condition termed "intention preservation", in addition to causality preservation and convergence. However, to our best knowledge, the condition of intention preservation has not been formalized rigorously. Consequently, algorithms following this framework are generally not formally proved with regard to intention preservation and often need to be patched to address newly discovered counterexamples.

In this paper, we propose an alternative framework called admissibility-based transformation (ABT). Theoretically, it requires only two correctness conditions, causality preservation and admissibility, that are formalized and provable. The new admissibility condition requires that the execution of every operation be "admissible", i.e., not violating object relations that have been established by earlier admissible executions. Because convergence is implied by these two conditions, our consistency model does not include an explicit condition of convergence. Practically, it establishes a principled design methodology in which sufficient conditions of transformation functions are first identified and a suitable control procedure is then found to satisfy those sufficient conditions. This way, the control procedure and transformation functions are not separated as in previous works (Suleiman et al. 1998; Sun and Ellis 1998)—instead, they work synergistically in ensuring correctness; correctness of the algorithm can be easily proved without requiring the transformation functions to work in all possible cases. Due to the above properties of ABT, it is easier to develop OT algorithms and prove their correctness.

At the algorithm level, the framework is elaborated on two primitive characterwise insert and delete operations in the context of linear documents. Roughly, the history buffer at each site is represented by an insertion sequence concatenated by a deletion sequence. Before an operation is propagated to remote sites, the effects of the deletions are excluded from it. When it is integrated at a remote site, it is always transformed with insertions before deletions. That is, when two concurrent insertions are transformed, the intermediary characters between them (if any) are alway present. As a result, the notorious "false-tie" problem that

has been under heated debate over the past decade (e.g., Sun et al. 1998; Sun and Sun 2006) will not occur. This is an important result in its own right.

The remainder of this paper is organized as follows. Section 2 gives a background of OT and motivates this research. Section 3 formalizes our new consistency model and correctness conditions. Sections 4 and 5 elaborate how to develop and prove an OT algorithm that satisfies the proposed conditions. Section 6 compares related works. Section 7 summarizes contributions and future directions.

2. Background and motivation

In a typical OT-based group editor, the shared document is replicated at all sites. Editing operations are first executed locally and then propagated to remote sites for synchronization. As a result, high local responsiveness and high concurrency are achieved. However, consistency control becomes a challenging issue because a remote operation often has to be executed in a new context that may be different from the context in which it is generated. Apparently it may not be safe to directly apply the operation in the new context without taking extra measures. In the following subsections, we formulate the problem and overview related approaches.

2.1. Problem abstraction

The basic idea of OT can be illustrated by the simple example as shown in Figure 1. Suppose two sites start editing the same document “NOT” and they collaboratively change it to “LOT”. Site 1 generates operation o_1=ins(1,‘L’) to insert character ‘L’ at position 1 (or between ‘N’ and ‘O’), while concurrently site 2 generates operation o_2=del(0,‘N’) to delete character ‘N’ at position 0. When o_1 is received at site 2, if it is executed as it is in the current state “OT”, the wrong

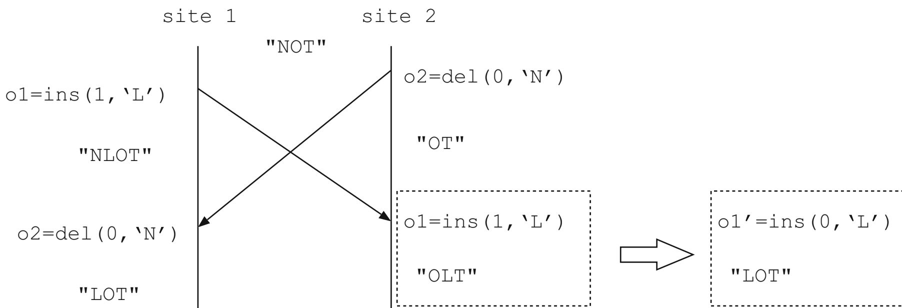

Figure 1. OT allows operations to be executed in different orders at different sites.

state “OLT” will result. The intuition of the seminal work in (Ellis and Gibbs 1989) is to transform o_1 with o_2, the concurrent operation that has been executed, to incorporate its effect such that o_1 will be executed at site 2 in the form of $o'_1 = \text{ins}\,(0, `L')$, which yields the correct state “LOT”. As a result, the final states at both sites converge and the intentions of both operations are preserved (Sun et al. 1998).

The problem of OT-based consistency control can be abstracted as follows: Suppose that an operation o is generated in (document) state s at site i and is first executed in s. Then o is propagated to another site j by message passing, where $j \neq i$, and is to be executed in some state s' at site j. The problem is how site j executes operation o in state s' or, alternatively, how to compute a “correct” form of o, denoted as o', that can be executed safely in the new state s'.

Assume that all sites in the system start from the same initial state s_0 and, as a convention, each site maintains a log of operations which are stored in their order of execution at that site (Ellis and Gibbs 1989). Usually the “context” of an operation, namely, the state in which it is generated or executed, can be represented by the operation log (or history buffer) (Sun et al. 1998). To solve the problem, we must answer two questions: First, what outcome of o' is considered “correct”. Second, how to develop an OT algorithm that computes the “correct” o' and, moreover, how to prove that the solution is “correct”.

2.2. Correctness criteria

It has been understood that in group editors data replicas at all sites must converge in the same state and the execution of all operations must preserve their natural cause-effect order (Ellis and Gibbs 1989; Ressel et al. 1996). Additionally, the data replicas must not only converge but the converged content must be further constrained (Sun et al. 1998). In particular, a group text editor must ensure not only that all sites have the same set of characters but also that the characters are in specific logical order for them to make sense together (Li and Li 2004). For example, a document containing word “not” may become something else if the characters are misordered as “ton”. So far we have seen two different approaches to formulating the extra constraints in (textual) group editors, as follows.

2.2.1. *Intention preservation*

In the first approach, group editors are required to preserve operation intentions (Sun et al. 1998). The condition of intention preservation is that executing an operation o at a remote site j must achieve the same effect as the intention of o when it is generated at site i. However, the constraint of “operation intention” has not been rigorously defined to our best understanding. On one hand, this conveys a good intuition and leaves a free space for interpreting it in a variety of application domains, such as texts (Sun et al. 1998; Sun 2002), structured documents (Davis et al. 2002), graphics (Sun and Chen 2002), and music scores

(Bellini et al. 2002). On the other hand, however, the ambiguity causes problems in verifying whether or not an algorithm really preserves operation intentions.

Example 1 Consider the following well-known scenario (Sun et al. 1998). Three sites concurrently generate three operations in the same state of three characters *s= "abc":* o_1 at site *1* inserts character '*x*' between '*b*' and '*c*', o_2 at site *2* deletes character '*b*', and o_3 at site *3* inserts character '*y*' between '*a*' and '*b*'. At site *2*, suppose o_2 and o_1 are executed first and its state becomes *s′= "axc"*. The question is what the intention of o_3 is with regard to s'.

The intention of o_3 in its generation state s is to insert 'y' between 'a' and 'b'. However, s' no longer contains 'b'. Hence the intention of o_3 is preserved as long as 'y' is inserted between 'a' and 'c'. According to (Sun et al. 1998), both "ayxc" and "axyc" are equally intention preserving and the algorithm only needs to choose either one to converge into. Observe, however, that the intention of o_1 is to insert 'x' after 'b' while the intention of o_3 is to insert 'y' before 'b'. Hence, due to the intermediary character 'b', character 'y' must precede 'x' in any state as long as they appear together, implying that the final state must be unambiguously "ayxc". This example reveals that the notion of operation intention as it has been interpreted (Sun et al. 1998; Sun 2002) fails to constrain the interplay of the intentions of multiple concurrent operations (Li and Li 2004).

2.2.2. *Operation effects relation*

To address the limitation of intention preservation, our early work introduced the notion of "operation effects relation" $\prec$ to model the logical order between operation effects (Li and Li 2004). For any operation o, its effect character c(o) is the character to be inserted or deleted by o. In Example 1, when o_1 is generated, the characters are ordered as 'a'$\prec$'b' $\prec$ 'x' $\prec$ 'c', and when o_3 is generated, the order is 'a' $\prec$ 'y' $\prec$ 'b' $\prec$ 'c'. By transitivity, the order between all characters is 'a' $\prec$ 'y' $\prec$ 'b' $\prec$ 'x' $\prec$ 'c'. Conceptually, the effects relation between the three concurrent operations o_1, o_2, and o_3 is $c(o_1) \prec c(o_2) \prec c(o_3)$ and the order between 'y' and 'x' must be 'y'$\prec$'x' without ambiguity.

Based on this concept, we formalize three consistency conditions as an alternative to (Sun et al. 1998): causality preservation, single-operation effects preservation, and multi-operation effects relation preservation (CSM) (Li and Li 2004). In more recent work (Li and Li 2007), we rigorously define the effects relation and rephrased CSM in two conditions: causality preservation and operation effects relation preservation (CR). However, in CSM and CR, the operation effects relation $\prec$ is a total order (over all characters that ever appear in the system) that has to be defined in the consistency model. Although the relation is natural and obvious in simple scenarios as in Example 1, it turns out a nontrivial task to develop a complete and rigorous definition, which is illustrated by the following scenario.

Example 2 Consider a scenario in which three sites start from the same state s= *"ab"*. Site *1* generates o_1 to insert '*x*' between '*a*' and '*b*', yielding *"axb"*, and site *2* concurrently generates o_2 to insert '*y*' between '*a*' and '*b*', yielding *"ayb"*. After executing o_1, site *3* generates o_3 to insert '*z*' between '*a*' and '*x*', yielding s'= *"azxb"*. The question is what the relation is between the three effect characters '*x*', '*y*' and '*z*' such that '*y*' can be correctly inserted into s'.

In this scenario, essentially there is no natural order between characters 'x' and 'y' because they are inserted at the "same" position without knowledge of each other. To have a consistent view, we can mandate an order such that the one inserted by a site with a smaller id precedes the other, yielding 'x' $\prec$ 'y'. When the same rule is applied directly when deciding the order between 'y' and 'z', we get 'y' $\prec$ 'z' because they are inserted independently of each other and the site id of 'y' is smaller. However, this result causes a problem because, due to transitivity, we have 'x' $\prec$ 'y' $\prec$ 'z', which contradicts the character order in state s' in which 'z' $\prec$ 'x'. Therefore, the order between 'y' and 'z' cannot be determined without considering the fact that o_1 happened before o_3. That is, we have to consider not only the site ids of operations that insert these two characters but also the order in which these operations are executed (Li and Li 2007).

In addition, consider the following scenario: A user first deletes character 'b' from state "abc", yielding "ac", and then inserts 'x' between 'a' and 'c', yielding "axc". To define the effects relation would require deciding an order between 'b' and 'x'. Note that there is not a natural order between these two characters unless they appear in the same state, e.g., by recovering 'b' via selective undo. However, again, in this and similar situations, an application or algorithm specific policy must be used to break the position tie, e.g., by mandating that the newly inserted character 'x' precedes the deleted character 'b' in the total order.

The concept of effects relation overcomes the problems of operation intention by formalizing the effects of single operations and introducing a new constraint on the interplay between the effects of multiple concurrent operations. It is an important step because finally we have a set of verifiable correctness criteria. However, to define a total order, we have to consider algorithm-specific tie-breaking policies, types of primitive operations, and execution order of operations. These limitations render the correctness criteria tightly coupled with specific algorithms and appear "artificial" to other algorithms, especially those that use different tie-breaking policies, as revealed in the review process of (Li and Li 2008a).

2.3. OT-based algorithms

Given the problem abstraction in Section 2.1 and the correctness criteria in Section 2.2, now we consider how existing OT-based approaches work. In an abstract sense, computing o' in state s' for a given operation o that is generated in

state s includes two steps: (1) choose or build a proper transformation path P from s to s', and (2) transform o against P. Conceptually a path is a sequence of operations, the execution of which in one state s leads to another state s'. There generally exist multiple paths from s to s': Some paths are unidirectional such that we only need to transform o with some path P_u, while some other paths are bidirectional such that we transform o first against a backward path P_b from s to a significant intermediate state s'' and then against a forward path P_f from s'' to s'. Existing OT algorithms naturally fall into three categories by the transformation paths they use, as reviewed in the following three subsections.

2.3.1. *Unidirectional transformation paths*

Many existing OT algorithms, such as adOPTed (Ressel et al. 1996), SOCT2 (Suleiman et al. 1997), GOTO (Sun and Ellis 1998), SDT (Li and Li 2004; Li and Li 2008a; Li and Li 2008b), and TTF (Oster et al. 2006b) adopt unidirectional transformation paths.

To integrate a remote operation o, their paths are only composed of operations that are concurrent with o. Typically, when operation o generated in state s at site i is ready to be executed in state s' at site j, the current history buffer H of site j is transposed into a concatenation of two sequences H_h and H_c, where H_h includes all operations in H that happened before o and H_c includes operations in H that are concurrent with o. Due to causality preservation, state s is but the state that is obtained by executing sequence H_h in the initial state s_0. Hence we just need to inclusively transform o with H_c to incorporate its effects such that the result, o', can be safely executed in state s'.

The advantage of this approach is that the transformation path is simple to construct by established methods (Suleiman et al. 1997; Sun and Ellis 1998). However, the transformation function is generally difficult to design because operations in H_c may be ordered arbitrarily at different sites. To maintain consistency, it is formally established that the (inclusion) transformation function must verify two transformation properties, TP1 and TP2 (Knister and Prakash 1994; Ressel et al. 1996), which ensure that transforming any operation o along arbitrary path H_c yields the same result. This is an underpinning assumption in a large body of work, e.g., (Ressel et al. 1996; Suleiman et al. 1997; Sun and Ellis 1998; Sun 2002).

Nevertheless, we observe two gaps in this approach: First, there is a *theoretical gap*. In (Ressel et al. 1996), it is only proved that satisfying TP1 and TP2 implies convergence. However, to our best knowledge, no sufficient conditions have been formally established for achieving intention preservation. As will be shown in the counterexamples of Section 4.2, the context equivalence condition given in (Sun et al. 1998) is not sufficient for preserving operation intentions as they interpreted.

Second, there is a *practical gap*. It turns out very difficult to design and prove transformation functions that verify TP2, as has been confirmed repeatedly in the

literature (Ressel et al. 1996; Suleiman et al. 1997; Sun et al. 1998; Imine et al. 2003; Li and Li 2008a; Oster et al. 2005a). Due to the need to consider complicated case coverage, formal proofs are very complicated and error-prone, even for OT algorithms that only treat two characterwise primitives (insert and delete) (Li and Li 2008a). The work by Molli and colleagues (Molli et al. 2003; Oster et al. 2005a; Imine et al. 2006; Oster et al. 2006b) resorts to theorem provers and tries to automatically prove TP1 and TP2. According to (Ressel et al. 1996), TP1 and TP2 are sufficient conditions for convergence. That is, even if TP1 and TP2 are proved, we can only conclude that an algorithm achieves convergence but cannot draw any conclusion about intention preservation.

2.3.2. *Unique transformation paths*

Realizing the difficulties in satisfying TP2, several approaches (Sun et al. 1998; Vidot et al. 2000; Shen and Sun 2002; Li et al. 2004) were proposed to free TP2 so that "ordinary" transformation functions can be used, which only use basic operation parameters such as position, type, and site id. In general, they achieve so by maintaining only one transformation path at all sites every time an operation o is transformed. Although using the same path achieves convergence, those approaches do not address how to achieve intention preservation. Counter-examples in (Li and Li 2008a) show that they violate the natural ordering of operation effects even though tie-breaking policies are not involved.

2.3.3. *Bidirectional transformation paths*

LBT (Li and Li 2007) is the first OT work to our knowledge that proposes to build bidirectional transformation paths. Based on the operation effects relation discussed above, LBT takes two main steps: (1) it first formally establishes sufficient conditions that transformation functions can work correctly, i.e., maintain the predefined total order, and (2) it then constructs a special bidirectional transformation path from the history buffer that ensure those sufficient conditions and hence any operation can be integrated correctly. In this new approach, theoretically it no longer needs to verify TP2 and intention preservation as in previous work.

2.4. Improvements over previous works

In hindsight, there are success lessons as well as spaces for improvement. Theoretically, using a total order as the basis for formalizing correctness criteria can be beneficial for developing new OT algorithms and proving their correctness. However, the total order is necessarily algorithm-specific and hence cannot be used directly for verifying other OT algorithms. Practically, our experience with SDT (Li and Li 2008a) and LBT (Li and Li 2007) reveals that building special transformation paths is significantly easier than designing

TP2-satisfying transformation functions. As shown in LBT, by building special transformation paths, we no longer need to design complicated transformation functions that work in all possible cases. However, due to the needs to explicitly derive and maintain the total order in transformation functions, transformation paths in LBT are very complicated.

This paper proposes a systematic scheme to address limitations in previous work and improve our early results. Theoretically, it no longer requires a predefined total order of characters. A new graph-based analysis tool will be provided as a basis to formalize a more "natural" partial order between characters that does not need algorithm-specific policies. Practically, based on the new theory, the design and proof of OT algorithms are significantly simplified. There is no need to store and retrieve an explicit effects relation and the transformation paths are more straightforward. As an evidence, in Sections 4 and 5, we will show how a simple and efficient algorithm is designed and proved. In Section 6, we will present more indepth comparisons between this work and related previous works.

3. A new consistency model

Observe that there are certain logical relations between characters in a document. A "correct" state must have the right set of characters as well as the right logical relations. For example, in Figure 1, state "LOT" is correct while state "OLT" is not. The intuition of our approach is to examine how characters are ordered in the initial state and how the order evolves as characters are inserted and deleted, based on which we study what is correct and how to achieve correctness. The rest of this section answers the question of what correctness means in group editors. Section 3.1 gives a model of group editor. Section 3.2 introduces a graph-based analysis tool. Section 3.3 defines some related concepts based on the graph. Section 3.4 formalizes correctness criteria.

3.1. Group editor

As a convention in group editors (Ellis and Gibbs 1989), we model the shared document as a linear string of characters (objects). Let the position of the first character in a string be zero. Notation $s[i]$ is the character at position i in string s and $s[c]$ is the position of character c in s, where $0 \leq i < |s|$ and $|s|$ is the number of characters in s. We assume that every appearance of any character is unique. With OT, we can uniquely identify characters by their positions in a linear address space, without the costs of maintaining global object ids.

For availability and responsiveness reasons, the shared document is replicated so that users can edit their local replicas. Suppose that all sites start from the same initial document state s_0. Synchronization among cooperating sites is achieved by

exchanging messages that piggyback editing operations. In addition to an internal primitive (null operation ϕ), we define two primitives for use in applications:

- ins(p,c): insert character c at position p.
- del(p,c): delete character c at position p.

For any operation o, attribute $o.t$ is its type (ins/del), $o.c$ is the effect character to be inserted or deleted, $o.p$ is its position parameter, and $o.id$ is the id of the site that generates o. Apparently $o.p$ is relative to some *definition state* s, denoted as s=dst(o). Note that only $o.p$ is relative to dst(o), whereas the other attributes will never be changed after o is generated. The definition state s is the *generation state* of o if o is generated in s, or the *execution state* of o if o is to be executed in s (Sun et al. 1998). The fact that the execution of operation o in state s yields state s' is denoted as s'=exec(s,o). Following the long established conventions (Ellis and Gibbs 1989), given any two operations o_1 and o_2, we say $o_1 \rightarrow o_2$ if o_1 happened before o_2, and $o_1 \parallel o_2$ if o_1 is concurrent with o_2.

3.2. Effects relation graph

The effects relation graph (or "graph" for brevity) is a global data structure for studying the behavior of group editors. Its purpose is to visualize effects of all operations in one unified view so as to signify abnormal executions. Note that the graph is only for theoretical analysis and need not be implemented in actual systems. It is constructed incrementally as operations are generated and executed at cooperating sites as if by an external observer of the system.

A graph G is a tuple $<V,E>$, where V is a set of nodes and $E \subseteq V \times V$ is a set of directed edges. Every node corresponds to a character that ever appears in the system and every edge represents the relation between two characters. Each node $n \in V$ has three attributes: *n.char* is the character it corresponds to; *n.counter* traces how the character is inserted or deleted; and *n.color* indicates the status, *white* for normal and *grey* for abnormal. An edge $<n_b, n_a> \in E$ means that n_b.*char* appears on the immediate left of n_a.*char* in some state.

The algorithm in Figure 2 shows how G is maintained. Suppose that V and E are initially empty and N is the number of sites. We first use the initial state s_0 to initialize G: If s_0 is not empty, for each character $c \in s_0$ add a new node n to V such that *n.char*=c, *n.counter*=N, and *n.color*=white. Then for any two nodes n_b, $n_a \in V$ add an edge $<n_b, n_a>$ to E, if n_b.*char* appears on the immediate left of n_a.*char* in s_0, or $s_0[n_b.char]+1=s_0[n_a.char]$.

We assume that every operation is executed locally first before it is propagated to remote sites. Every operation o is invoked once when it is generated (and executed locally) and once when it is executed at every remote site. Every invocation updates G once. Hence G is eventually updated N times by every operation o. The order of invocations must maintain their natural cause-effect

(a) Initialize:
(a.1) $\forall c \in s_0$ add a new node n to V such that
(a.2) $n.char \leftarrow c$; $n.counter \leftarrow N$; $n.color \leftarrow$white;
(a.3) $\forall n_b, n_a \in V$ if $s_0[n_b.char] + 1 = s_0[n_a.char]$
(a.4) add edge $< n_b, n_a >$ to E;

(b) Invoke an insertion o in state s:
(b.1) if $\exists n \in V : n.char = o.c$
(b.2) $n.counter \leftarrow n.counter + 1$;
(b.3) else add a new node n to V such that
(b.4) $n.char \leftarrow c$; $n.counter \leftarrow 1$; $n.color \leftarrow$white;
(b.5) if $\exists n_b \in V : n_b.char = s[o.p - 1]$
(b.6) add $< n_b, n >$ to E if $< n_b, n > \notin E$;
(b.7) if $\exists n_a \in V : n_a.char = s[o.p]$
(b.8) add $< n, n_a >$ to E if $< n, n_a > \notin E$;

(c) Invoke a deletion o in state s:
(c.1) find $n \in V$ such that $n.char = o.c$;
(c.2) if $s[o.p] \neq o.c$
(c.3) $n.color \leftarrow$ grey;
(c.4) elseif $n.color \neq grey$
(c.5) $n.counter \leftarrow n.counter - 1$;
(c.6) $n.color \leftarrow$ grey if $n.counter = -1$;

Figure 2. Initializing and maintaining a global graph G.

relation: (1) given any operation o, its local invocation must be earlier than any remote invocation, and (2) given any o_1 and o_2, if $o_1 \rightarrow o_2$, invocation of o_1 must be earlier than invocation of o_2 at every site.

Every invocation of any o is always relative to some state s at some site. The local invocation of an insertion o always leads to the creation of a new node n in G with $n.char = o.c$, $n.counter = 1$, and $n.color$=white. After that, every remote invocation of insertion o increments $n.counter$ by one. The invocation of insertion o in s will cause a new character $o.c$ to be inserted between two neighboring characters $s[o.p-1]$ and $s[o.p]$. Hence if these two characters exist, we also add the two edges $<n_b, n>$ and $<n, n_a>$ into G, where $n_b.char = s[o.p-1]$ and $n_a.char = s[o.p]$. As an insertion o is executed in different states at different sites, it is possible that different nodes are connected with node n in this way.

When invoking a deletion o on G, it is similarly possible that $o.p$ may point to different characters in different execution states. Since $o.c$ is a constant once o is generated and we never delete nodes in G, we can always find the node $n \in V$ that contains character $o.c$. If $s[o.p]$ fails to point to the right character $o.c$, we turn the color of n to grey, which signals an abnormal status. Otherwise we decrement $n.counter$ by one. If $n.counter$ has been decremented this way more than N times, we also turn the color of that node to grey.

As shown in Figure 3, we use an example to illustrate how a graph is constructed. Suppose that three sites start from initial state s_0="abc" and concurrently generate three operations: o_1=ins(2,'x') yields state "abxc" at site 1, o_2=del(1,'b') yields state "ac" at site 2, and o_3=ins(1,'y') yields state "aybc" at

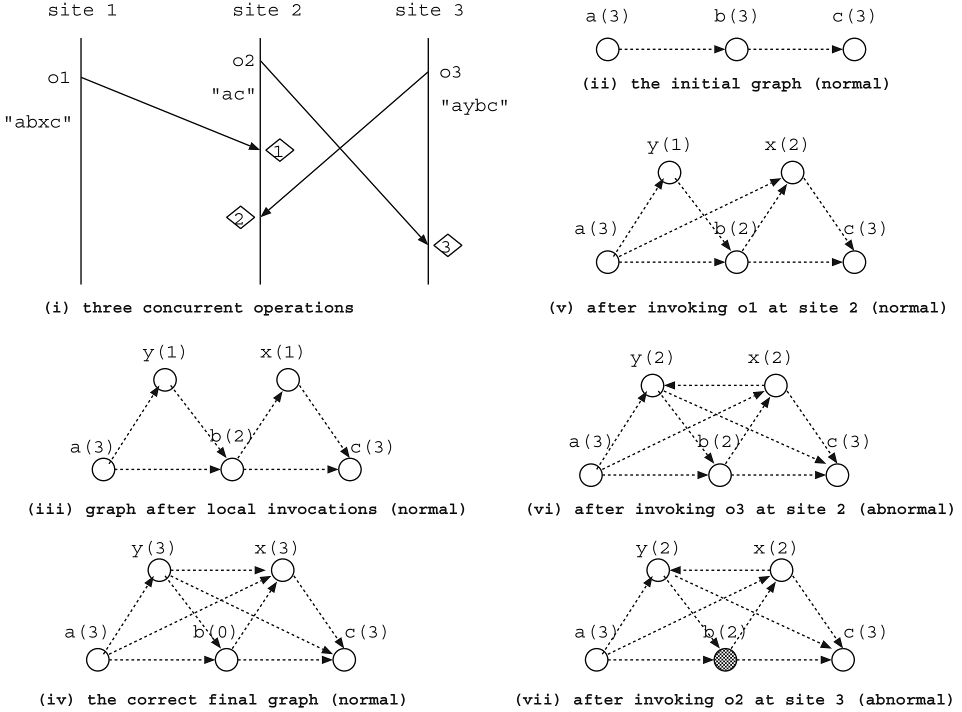

Figure 3. An example of normal and abnormal effects relation graphs.

site 3. The graph as initialized with s_0 contains three white nodes each with counter value 3. After the three operations are executed locally, regardless of the order of invocation on the graph, the counter of node ‘b’ is decremented to 2; a new node ‘x’ is created with counter value 1 and with a new edge from node ‘b’ and a new edge to node ‘c’; and a new node ‘y’ is created with counter value 1 and with a new edge from node ‘a’ and a new edge to node ‘b’.

We assume that every operation is executed locally once it is generated, without any interference from concurrent operations. It is obvious that the current ordering between all characters in the graph, after all operations are invoked locally, is “natural” and reflects the operation intentions. Ideally, all remote invocations should be consistent with this natural order obtained by all local invocations. The correct final graph is shown in Figure 3(iv). In particular, the order between ‘x’ and ‘y’ must be that ‘y’ precedes ‘x’ wherever they appear together.

When o_1 is received at site 2, suppose it is executed as o'_1=ins(1,‘x’), which yields state “axc”. Then invocation of o'_1 on the graph increments the counter of node ‘x’ to 2 and adds a new edge from node ‘a’ to node ‘x’. When o_3 arrives at site 2, suppose it is executed as o'_3 = ins(2,‘y’), which yields state “axyc”. Apparently this result violates the natural order between ‘x’ and ‘y’. The invocation of o'_3 increments the counter of node ‘y’ to 2 and adds a new edge

from node ‘x’ to node ‘y’. As a result, there is a cycle between ‘x’, ‘y’ and ‘b’. Intuitively, o'_3 is not a correct invocation.

When o_2 is received at site 3, if it is executed as-is in the current state “aybc”, the wrong character ‘y’ is deleted. As a result, the color of node ‘b’ is turned grey. Intuitively, o_2 is not a correct invocation.

3.3. Some key concepts

As an analysis tool rather than a specific concurrency control algorithm, the graph maintenance algorithm in Figure 2 itself does not calculate the execution form of any operation in its execution state. Instead, the operation parameters (esp. position) are determined by the group editor or, more specifically, its concurrency control algorithm. The graph detects possible inconsistent behavior of the group editor. In the following, based on the effects relation graph, we first define what a consistent graph is like (Definitions 1–4), then derive the effects relation from a consistent graph (Definition 5), and finally derive correctness conditions (Definitions 6–12).

Definition 1 Given two nodes, n_1 and n_2, in graph G, the order between them with regard to G, denoted as $order_G(n_1,n_2)$, is defined as (1) $n_1 \rightsquigarrow n_2$ if there is a path from n_1 to n_2, or $n_2 \rightsquigarrow n_1$ if there is a path from n_2 to n_1; (2) **cyclic** if $n_1 \rightsquigarrow n_2$ and $n_2 \rightsquigarrow n_1$; (3) **unorderable** if neither $n_1 \rightsquigarrow n_2$ nor $n_2 \rightsquigarrow n_1$.

Definition 2 Given two graphs, G_1 and G_2, we say that G_1 is similar to G_2, denoted by $G_1 \approx G_2$, if (1) G_1 and G_2 have the same set of nodes with identical attributes, and (2) for any nodes n_1 and n_2, $order_{G_1}(n_1, n_2) = order_{G_2}(n_1, n_2)$.

Definition 3 Given two graphs, G_1 and G_2, G_1 is a subgraph of G_2, denoted by $G_1 \subseteq G_2$, if (1) any node n in G_1 is also in G_2, and (2) for any two nodes n_1 and n_2 in G_1 (and G_2), $order_{G_1}(n_1, n_2) = order_{G_2}(n_1, n_2)$.

By these definitions, any graph G is a subgraph of itself and is similar to itself.

Definition 4 [Consistent Graph] An effect relation graph G is consistent if G is acyclic and without grey nodes.

Lemma 1 *Given two graphs, G and G', if G is consistent and $G' \subseteq G$, G' is also consistent.*

Suppose that the final graph resulted after all generated operations have been invoked at all sites be consistent and let it be G_f. Due to the algorithm in Figure 2, no node is ever deleted from the graph. Hence every character that ever appears

in the system has a corresponding node in G_f. Let C_t be the set of characters that ever appear.

Definition 5 [Effect Relation $\prec$] For any $c_1, c_2 \in C_t$, we say $c_1 \prec c_2$ if there exists a path from node n_1 to n_2 in G_f, where $n_1.char = c_1$ and $n_2.char = c_2$.

Note that we define relation $\prec$ only if the final graph is consistent. Apparently relation $\prec$ is transitive. That is, $c_1 \prec c_2$ and $c_2 \prec c_3$ implies $c_1 \prec c_3$. However, relation $\prec$ is only a partial order. For example, if 'x' is deleted from state "axb" and then 'y' is inserted to yield "ayb", it does not order 'x' and 'y' directly unless 'x' appears in the same state as 'y'. In general, for any two characters $c_1, c_2 \in C_t$, if they ever appear in the same state of a group editor, their relation must have been determined by the group editor, or more specifically, its concurrency control algorithm. There must be two nodes that contain c_1 and c_2, respectively, in the graph and at least one path exists between them.

Definition 6 A state s corresponds to a graph G if (1) for every $c \in s$, there is one and only one node $n \in G$ such that $n.char = c$, and (2) for every node n in G, there is one and only one character $c \in s$ such that $c = n.char$.

Definition 7 [Reachable State] A given state s is reachable if s corresponds to a subgraph of G_f and for any two characters $c_1, c_2 \in s$, if $s[c_1] < s[c_2]$ then $c_1 \prec c_2$ holds.

Corollary 1 *Given any reachable state s, its corresponding graph is a subgraph of G_f and is consistent.*

Definition 8 [Admissible Operation] Any operation o is admissible in its definition state $s = dst(o)$, if s is reachable (or its corresponding graph G is consistent) and the invocation of o results in a consistent graph.

Axiom 1 The initial state s_0 is reachable.
Axiom 2 The graph constructed from s_0 is consistent.

Assumption 1 If state s is reachable, any operation o generated in s is admissible in s.

The following theorem states an important implication of an operation being admissible in its definition state. We omit its proof because the theorem is self-explanatory.

Theorem 1 *Given an operation o admissible in state s, if $o.t = del$ and s is not empty, then $o.c = s[o.p]$ must hold; if $o.t = ins$, then o.c must be inserted between two characters, $s[o.p-1]$ and $s[o.p]$, and $s[o.p-1] \prec o.c \prec s[o.p]$, if these two characters exist.*

Definition 9 [Operation Sequence] An operation sequence sq is an ordered list of operations such that $sq[i+1]$ is defined in the state resulted from executing $sq[i]$, or $dst(sq[i+1]) = exec(dst(sq[i]), sq[i])$, where $0 \leq i < |sq|$. In particular, state $dst(sq[0])$ is called the definition state of sq, or $dst(sq) = dst(sq[0])$.

Let s=dst(sq). Then sq is executed in s as follows: first execute $sq[0]$ in s, then execute $sq[1]$ in state exec(s,$sq[0]$), and so forth. We extend the notion of operation execution such that $s' = exec(s,sq) = exec(exec(s,sq[0]), sq[1,n-1])$, where $n = |sq|$ and $sq[1,n-1]$ is the subsequence of sq ranging from its second to last operation.

Definition 10 [Admissible Operation Sequence] Any operation sequence sq is admissible in $dst(sq)$ if every operation o in sq is admissible in $dst(o)$.

Lemma 2 *Given an operation sequence sq, let s= dst(sq) and s'= exec(s, sq). If s is reachable and sq is admissible, then s' is also reachable.*

Definition 11 [Effects Equivalence] Given two sequences, sq_1 and sq_2, that are defined and admissible in state s, we say that they are effects equivalent, denoted by $sq_1 \approx sq_2$, if executed in s they yield the same final state.

3.4. Correctness of group editors

Definition 12 [Correctness Criteria] Assume that all sites in a group editor start from the same initial state s_0. The group editor is correct if the following two criteria always hold.

(1) Causality preservation: For any two operations, o_1 and o_2, if $o_1 \rightarrow o_2$, then o_1 is invoked before o_2 at any site.
(2) Admissibility Preservation: The invocation of every operation is admissible in its execution state.

By the initials of these two conditions, we call this model “CA”. Given a reachable initial state s_0, the graph remains consistent because no admissible invocation introduces a new cycle or a grey node. As a result, in the quiescent state, in which all generated operations have been executed at all sites, the final graph G_f is consistent and relation $\prec$ is well-defined. That is, the invocation of every operation *eventually* preserves the effect relation $\prec$, which is a partial order.

This model can be used to verify any OT-based group editors, even if they do not explicitly have the notion of effects relation. Due to the causality-preserving process of graph construction, the order between characters is essentially established by the initial state and invocations of local operations in their generation states. A group editor is correct if the execution of any remote

operation does not contradict the character order established earlier by itself, i.e., not introducing cycles and grey nodes. The existence of a cycle between any two nodes (say n_1 and n_2) in the graph means that $n_1.char$ precedes $n_2.char$ in some state while $n_2.char$ precedes $n_1.char$ in some other state. The existence of a grey node n in the graph means that the character $n.char$ should not have been deleted. Either case may lead to divergence of states at different sites or the violation of the established character order even if the final states converge.

In Definition 12, we do not include an explicit convergence condition as in previous work (Ressel et al. 1996; Sun et al. 1998) because convergence is implied by the two given conditions. Intuitively, in a correct group editor, all inserts and deletes are executed correctly at every site. As a result, in the quiescent state, all sites have the same set of characters that are in the same order. That is, all sites converge in the same state. Assuming that each site maintains a history of locally executed operations, Theorem 2 asserts that all replicas of the shared data converge in the quiescent state. Not to be tedious, here we omit the formal proof.

Theorem 2 *In a correct group editor, after all generated operations are executed at all sites, any two histories in the system are effects equivalent.*

Therefore, the problem of OT-based concurrency control is how to ensure that every remote operation is admissible in its execution state. We elaborate how to achieve so in the remainder of this paper. It is well understood that a typical OT algorithm consists of two layers: transformation functions determine how to transform two operations, and a control procedure determines how to call transformation functions to integrate remote operations (Sun and Ellis 1998). Accordingly, Section 4 first examines conditions under which an admissible operation remains admissible after being transformed with another admissible operation. Section 5 then shows how to ensure the admissibility of every operation executed in the system.

4. Transformation functions and conditions

In Section 4.1 we define three basic transformation functions. However, counterexamples show that these functions do not work correctly in some cases. We examine those cases in Section 4.2 with regard to the proposed consistency model. After that, we study their sufficient conditions (which ensure the correctness of transformation functions if satisfied) in Sections 4.3 and 4.4. Section 4.5 generalizes the conditions to operation sequences.

4.1. Transformation functions

Following the notations in (Sun and Ellis 1998), we say that any two operations, o_1 and o_2, are contextually equivalent, denoted as $o_1 \sqcup o_2$, if $\mathrm{dst}(o_1)=\mathrm{dst}(o_2)$; they

are contextually serialized, denoted as $o_1 \mapsto o_2$, if dst(o_2)=exec(s, o_1) and s=dst (o_1). In the following, we define three binary transformation functions, IT(o_1,o_2), ET(o_1,o_2) and SWAP(o_1,o_2), where o_1 and o_2 are assumed to be admissible in their definition states. Ideally, we would like the results returned from these transformation functions to be also admissible.

Given two operations, o_1 and o_2, where $o_1 \sqcup o_2$ and s=dst(o_1)=dst(o_2), the purpose of o_1'=IT(o_1,o_2) is to include the effect of o_2 into o_1 such that $o_2 \mapsto o_1'$ and o_1' is defined in state s'=exec(s,o_2). Function 1 defines IT(o_1, o_2). If $o_2.p$>$o_1.p$, meaning that o_2 inserts or deletes a character on the right of the target position of o_1, we return o_1 as it is because the effect of o_2 does not affect $o_1.p$ in s'.

FUNCTION 1. *IT(o_1, o_2): o_1'*

```
1  o1' ← o1;
2  if o2.p < o1.p
3      if o2.t = ins
4          o1'.p ← o1'.p + 1;
5      else //if o2.t = del
6          o1'.p ← o1'.p − 1;
7  else if o2.p = o1.p
8      if o2.t = ins ∧ o1.t = del
9          o1'.p ← o1'.p + 1;
10     else if o2.t = o1.t = ins ∧ o2.id < o1.id
11         o1'.p ← o1'.p + 1;
12     else if o2.t = o1.t = del
13         o1' ← φ;
14 return o1';
```

If $o_2.p$<$o_1.p$ and $o_2.t$=ins, meaning that o_2 inserts a character on the left of the target position of o_1, we increment $o_1.p$ by one because the original position of o_1 has been shifted by the insertion of $o_2.c$ in s' (lines 3–4). If $o_2.p$<$o_1.p$ and $o_2.t$= del, meaning that o_2 deletes a character on the left of the target position of o_1, we decrement $o_1.p$ by one (lines 5–6).

When $o_2.p$=$o_1.p$, we need to consider four combinations of the types of o_1 and o_2. If $o_2.t$=ins and $o_1.t$=del, meaning that o_2 inserts character $o_2.c$ right before character $o_1.c$=s[$o_1.p$], we increment $o_1.p$ by one because, due to the semantics of insertion and deletion, the original character $o_1.c$ has been shifted by the insertion of $o_2.c$ in s' (lines 8–9). If $o_2.t$=del and $o_1.t$=ins, we return o_1 as-is because the deletion of $o_2.c$ does not affect $o_1.p$ in s'. For the two cases in lines 10 and 12, we define the following two policies:

(P1) If o_1 and o_2 insert at the same position in s, we compare their site ids to order $o_1.c$ and $o_2.c$ such that the one with a smaller site id precedes the other (lines 10–11). That is, if o_2.id<$o_1.id$, we increment $o_1.p$.

(P2) If o_1 and o_2 attempt to delete the same character in s, the one to be executed later (o_1) is transformed into an identity operation ϕ so that the same character will not be deleted more than once (lines 12–13).

FUNCTION 2. $ET(o_1, o_2)$: o_1'

```
1   o1' ← o1;
2   if o2.p < o1.p
3       if o2.t = ins
4           o1'.p ← o1'.p − 1;
5       else //if o2.t = del
6           o1'.p ← o1'.p + 1;
7   else if o2.p = o1.p
8       if o2.t = o1.t = del
9           o1'.p ← o1'.p + 1;
10      else if o1.t = del ∧ o2.t = ins
11          return undefined; //o1 depends on o2
12  return o1';
```

FUNCTION 3. $SWAP(o_1, o_2)$: $< o_1', o_2' >$

```
1   if o1.p > o2.p
2       if o2.t = ins
3           o1.p ← o1.p − 1;
4       else //o2.t = del
5           o1.p ← o1.p + 1;
6   else if o1.p = o2.p
7       if o1.t = o2.t = del
8           o1.p ← o1.p + 1;
9       else if o1.t = del ∧ o2.t = ins
10          return undefined; //o1 depends on o2
11      else if o1.t = o2.t = ins
12          o2.p ← o2.p + 1;
13      else //o1.t = ins ∧ o2.t = del
14          o2.p ← o2.p + 1;
15  else //o1.p < o2.p
16      if o1.t = ins
17          o2.p ← o2.p + 1;
18      else //o1.t = del
19          o2.p ← o2.p − 1;
20  return < o1, o2 >;
```

Given two operations, o_1 and o_2, where $o_2 \mapsto o_1$, s=dst(o_2), and s'=dst(o_1)=exec(s,o_2), the purpose of o_1'=ET(o_1,o_2) is to exclude the effect of o_2 from o_1 as if o_2 had not happened such that $o_1' \sqcup o_2$ and o_1' is defined in s. Function 2 defines ET(o_1,o_2). If $o_2.p > o_1.p$, meaning that o_2 inserted or deleted a character on the right of $o_1.c$, we return o_1 as it is because the effect of o_2 did not affect o_1. If $o_2.p < o_1.p$, meaning that o_2 inserted (deleted) a character on the left of $o_1.c$, we decrement (increment) $o_1.p$ by one to exclude the effect of o_2, as in lines 2–6.

When $o_1.p=o_2.p$, we need to consider four combinations of the operation types. (1) If $o_1.t=o_2.t=$del, meaning that o_2 and o_1 delete two neighboring characters in s in tandem, the one deleted first must precede the one deleted next, In this case, we increment $o_1.p$ by one had $o_2.c$ not been deleted earlier (lines 8–9). (2) If $o_2.t=$ins and $o_1.t=$del, meaning that o_1 deletes the character inserted by o_2, we say that o_1 depends on o_2 and o_1' is not defined (lines 10–11). In this case, it does not make sense logically to exclude the effect of o_2 from o_1. (3) If $o_1.t=o_2.t=$ ins, meaning that they insert two characters in s in tandem, the one inserted later must precede the one inserted earlier, In this case, we return o_1 as it is because the effect of o_2 would not affect $o_1.p$ even if o_2 had not been executed. (4) The following tie-breaking policy is hidden:

(P3) If $o_1.p=o_2.p$, $o_1.t=$ins and $o_2.t=$del, meaning that o_2 deleted character $o_2.c=s[o_2.p]$ and o_1 inserts character $o_1.c$ at the same position ($o_1.p$ in s' and $o_1.p=o_2.p$), then o_1 is returned as-is.

The third transformation function SWAP(o_1,o_2) is defined in Function 3. It transposes two operations, o_2 and o_1, where $o_2 \mapsto o_1$, into o_1' and o_2' such that $o_1' \mapsto o_2'$ and $[o_2,o_1] \approx [o_1', o_2']$. Conceptually, this amounts to first processing $o_1'=$ET(o_1, o_2) to get $o_1' \sqcup o_2$ and then $o_2'=$IT(o_2, o_1') to get $o_1' \sqcup o_2'$. As will be discussed later in Sections 4.3 and 4.4, IT and ET have different sufficient conditions. This way the correctness of SWAP would depend on both IT and ET. Observe, however, that the relation between $o_1.c$ and $o_2.c$ is known after processing ET(o_1, o_2). We merge the rules of ET and IT in the definition of SWAP so that the sufficient condition of SWAP is the same as that of ET.

4.2. The need for tighter IT/ET preconditions

Intuitive as they are, IT and ET as defined above cannot guarantee that the transformed operations are admissible. More specifically, the contextual equivalence condition $o_1 \sqcup o_2$ is not sufficient for ensuring the correctness of IT(o_1,o_2) and the contexual serialization condition $o_2 \mapsto o_1$ is not sufficient for ET(o_1,o_2). In the following, we discuss several counterexamples and analyze them based on our consistency model.

4.2.1. *Counterexamples*

Based on our consistency model, Example 3 first explains a classic counterexample (Suleiman et al. 1997; Sun et al. 1998) to the contextual equivalence condition of IT.

Example 3 As shown in Figure 4, the initial state is $s_0=$ *"abc"*. At site 2, after $o_2=$ *del(1,'b')* is invoked locally, the state becomes $s_1=$ *"ac"*. Then two concurrent

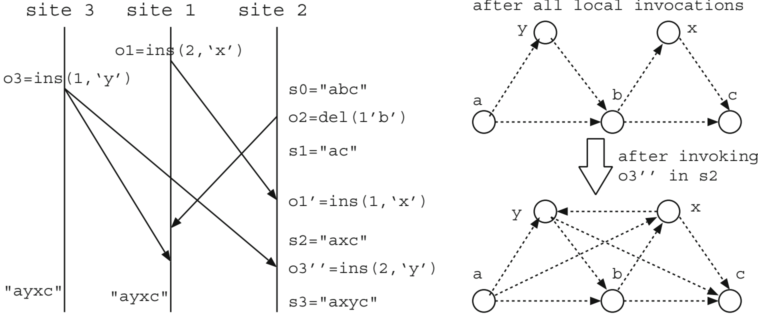

Figure 4. Inclusively transforming two concurrent insertions: o_3''=IT(IT(o_3, o_2), IT(o_1, o_2)) is not admissible in s_2.

operations, o_1= *ins*(2,'*x*') and o_3= *ins*(1,'*y*'), are integrated in tandem. First we get o_1'= *IT*(o_1, o_2)= *ins*(*1,*'*x*') and s_2= *exec*(s_1, o_1')= *"axc"*. Invoking o_1' *in* s_1 adds a new edge from node '*a'* to '*x*'. Now we compute o_3' that can be executed in s_2. From s_0= *dst*(o_3) to s_2= *dst*(o_3'') there are a sequence of two operations [o_2, o_1'], both operations being concurrent with o_3. We first get o_3' = *IT*(o_3, o_2)= *ins*(*1,*'*y*') and then compute o_3'' = *IT*(o_3', o_1'). Since the positions of o_1' and o_3' tie, by the IT definition in Function 1, we compare their site ids to break tie and get o_3''= *ins* (*2,*'*y*'). The execution of o_3'' in state s_2 yields s_3= *"axyc"*. Invoking o_1' in s_2 adds an edge from node '*x*' to '*y*', which yields a cycle. That is, o_3'' is not admissible in s_2.

Example 4 As shown in Figure 5, suppose the initial state is s_0= *"ab"*. The generation of o_1= *ins*(*1,*'*x*') results in state s_1= *"axb"*. After that, the generation of o_2= *ins*(*2,*'*y*') yields s_2= *"axyb"*. It is easy to see that o_2' = *ET*(o_2,o_1)= *ins*(*1,*'*y*') is admissible in state s_0. Then o_2'' = *IT*(o_2' ,o_1) is defined in state s_1. However, by the *IT* definition in Function 1, o_2''= *ins*(*1,*'*y*') is obviously not admissible in state s_1 because its invocation creates a cycle between '*x*' and '*y*'.

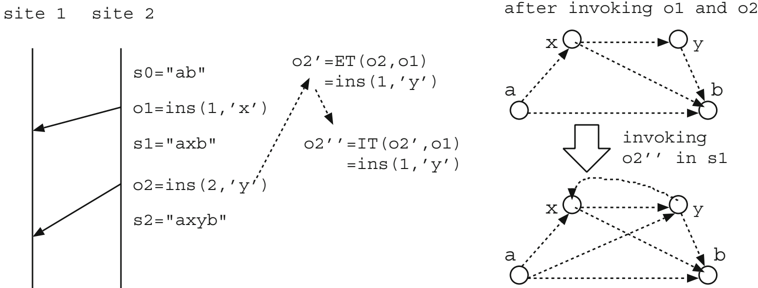

Figure 5. Inclusively transforming an insertion with another insertion that happened before it. o_2'' =IT (ET(o_2, o_1), o_1) is not admissible in s_1.

Example 3 reveals that, although the contextual equivalence condition of IT is satisfied, the result of IT(o_x,o_y) may not be admissible when $o_x \parallel o_y$, $o_x.t$=$o_y.t$=ins and $o_x.p$=$o_y.p$. Example 4 shows that, although the contextual equivalence condition of IT is satisfied, the result of IT(o_x,o_y) may not be admissible when $o_x \longrightarrow o_y$, $o_x.t$=$o_y.t$=ins and $o_x.p$=$o_y.p$.

For counterexamples to the contextual serialization condition of ET, consider the following two scenarios.

Example 5 Consider the scenario as shown in Figure 6. The system initial state is s_0= *"abc"*. Site 1 generates o_1= *ins*(*2,'x'*) and concurrently site 2 generates o_2= *del*(*1,'b'*). At site 2, when o_1 is received, we compute o_1'= *IT*(o_1,o_2)= *ins(1,'x'*), which is admissible in its execution state s_1= *exec*(s_0,o_2)= *"ac"*. By the ET definition in Function 2, o_1''= *ET*(o_1' ,o_2)= *ins*(*1,'x'*) is defined in state s_0. However, o_1'' is not admissible because its invocation in s_0 results in a cycle between node '*x*' and '*b*'.

Example 6 As shown in Figure 7, three sites start from s_0="abc". Site 1 and site 2 concurrently generate two operations o_1=*del*(*1,'b'*) and o_2=*ins*(*2,'x'*), respectively. At site 3, the execution of o_1 yields s_1=*"ac"*. When o_2 is received, we get o_2' =IT(o_2,o_1)=*ins*(*1,'x'*). The execution of o_2' yields s_2= *exec*(s_1, o_2')=*"axc"*. After that, site 3 generates o_3=*ins*(*2,'y'*), yielding s_3= *exec*(s_2,o_3)= *"axyc"*. Now consider how to compute o_3'' that is defined in s_0. From s_2= *dst*(o_3) to s_0= *dst*(o_2') there are a sequence of two operations [o_1, o_2']. We first get o_3'=ET(o_3, o_2')=*ins*(*1,'y'*) and then o_3''=ET(o_3', o_1)= *ins*(*1,'y'*). As shown in Figure 7, o_3'' is not admissible in state s_0 because its invocation introduces a cycle.

Example 5 reveals that the result of ET(o_x, o_y) may not be admissible when $o_y \parallel o_x$, $o_y.t$=del, $o_x.t$=ins, and $o_y.p$=$o_x.p$. Example 6 reveals that the result of ET(o_x,o_y) may not be admissible when $o_y \longrightarrow o_x$, $o_y.t$=del, $o_x.t$=ins, and $o_y.p$=$o_x.p$. Note, however, that the contextual serialization condition of ET is satisfied in both scenarios.

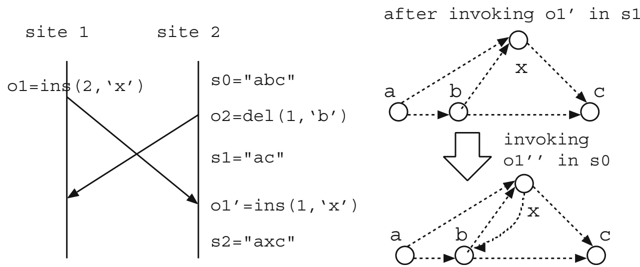

Figure 6. Exclusively transforming an insertion with a concurrent deletion: o_1''=ET(IT(o_1, o_2), o_2) is not admissible in s_0.

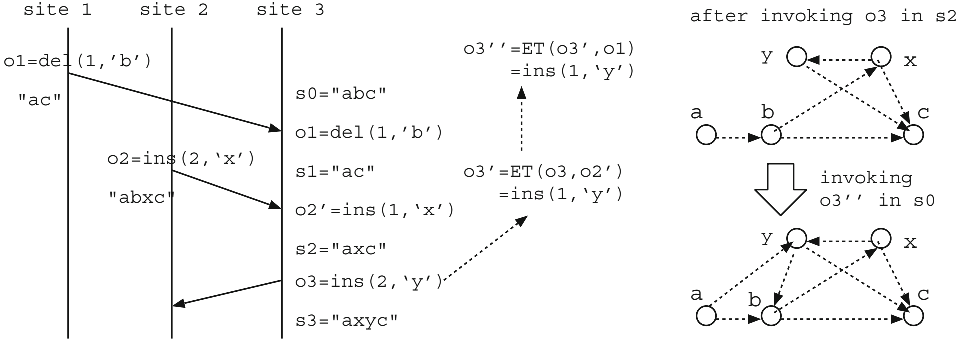

Figure 7. Exclusively transforming an insertion with a deletion that happened before it: o_3''=ET(ET(o_3,IT(o_2, o_1)), o_1) is not admissible in s_0.

4.2.2. *Further analyses*

The above four counterexamples show that care must be taken when the basic IT/ET functions as defined in Functions 1 and 2 are used. Here we first define an important concept "landmark characters" and analyze these scenarios further to draw intuitions of solutions.

Definition 13 [**Landmark Character**] For any three characters, c_1, c_2, $c_3 \in C_t$, where C_t is the set of characters that ever appear in the system, we say that c_3 is a landmark character between c_1 and c_2, if either $c_1 \prec c_3 \prec c_2$ or $c_2 \prec c_3 \prec c_1$.

In the IT scenario of Example 3, the tie between o_3' and o_1' is caused by the fact that landmark character 'b' between 'y' and 'x' ('y'≺'b' ≺ 'x') is not present in state s_2. Because o_1'=IT(o_1,o_2) and o_3' = IT(o_3,o_2) have been processed before we do IT(o_3', o_1'), in state s_2=dst(o_1')=dst(o_3'), the landmark character 'b' has been deleted by operation o_2. In other words, if another path (e.g., [o_1,o_2] at site 1 in Figure 4) is taken such that character 'b' is not deleted before inclusively transforming the two insertions, the problem will not occur. For example, when we do IT(o_3,o_1), since the landmark character 'b' is present in state s_0=dst(o_1)= dst(o_3), there is no tie at all. As a result, o_3'=IT(o_3,o_1)=ins(1,'y') is admissible in state exec(s_0,o_1)="abxc".

On the other hand, consider the following scenario. Suppose two sites have the same initial state s_0="ab". Site 1 generates o_1=ins(1,'x') and concurrently site 2 generates o_2=ins(1,'y'). In this case, due to the lack of landmark character between 'x' and 'y', there is no natural order between them and hence, in theory, either result "axyb" or "ayxb" is acceptable. In IT (Function 1), policy P1 is adopted to break the tie such that the result is unambiguously "axyb". As a result, both o_1'=IT(o_1,o_2)=ins(1,'x') and o_1' =IT(o_2,o_1)=ins(2,'y') are admissible.

In the ET scenario of Example 6, when o_3 is generated in state s_2="axc", it is clear from Figure 7 that the landmark character between 'y' and 'b' is 'x',

namely, ‘b’$\prec$‘x’ $\prec$ ‘y’. However, after processing o'_3=ET(o_3, o'_2), the definition state of o'_3 or dst(o'_3), in which ‘x’ is assumed to not have been inserted by o'_2, is s_1=“ac”. The tie in processing ET(o'_3, o$_1$) is essentially caused by the absence of landmark character ‘x’ in s_1. In other words, the problem will not arise had the landmark character ‘x’ been present in s_1. For example, suppose that o_2 is executed before o_1 at site 3 before o_3 is generated. Then the path will instead be [o_2, o'_1], where o'_1=del(1,‘b’). There will be no tie in computing o''_3, the form of o_3 relative to state s_0: We first get $o'_3 = \mathrm{ET}\left(o_3, o'_1\right)$=ins(3,‘y’) and then o''_3=ET(o'_3, o_2)=ins(2,‘y’). Obviously o''_3 is admissible in s_0.

On the other hand, consider the following scenario. Starting from s_0=“abc”, a user first generates o_1=del(1,‘b’) and then o_2=ins(1,‘x’). There is neither landmark character nor natural order between ‘b’ and ‘x’. In theory, either ‘b’$\prec$ ‘x’ or ‘x’ $\prec$ ‘b’ is acceptable. In ET (Function 2), policy *P3* is adopted such that the relation is mandated as ‘x’$\prec$‘b’ unambiguously. The result o'_2=ET(o_2, o_1)=ins (1,‘x’) is admissible in s_0.

The above analyses give us the intuition that, although the basic IT/ET functions are not guaranteed to yield admissible results on arbitrary transformation paths, it is possible to find some special transformation paths for them to work correctly. On those special paths, the landmark characters must be present, if they exist at all, when the results of IT/ET could be nondeterministic. The tie-breaking policies (P1 and P3) should be used only when landmark characters do not exist.

For any two characters c_1, $c_2 \in C_t$, we denote the set of landmark characters between c_1 and c_2 as $C_{ld}\,(c_1, c_2)=\{\, c_3 \in C_t \mid c_1 \prec c_3 \prec c_2 \vee c_2 \prec c_3 \prec c_1 \,\}$. For any two operations, o_1 and o_2, set $C_{ld}(o_1.c,\ o_2.c)$ can be used to determine the order between $o_1.c$ and $o_2.c$. However, we do not really need to compute the whole set C_{ld} ($o_1.c$, $o_2.c$). By transitivity of relation$\prec$, the knowledge that there exists (at least) one landmark character between them is good enough for concluding $o_1.c \prec$ $o_2.c$ or o_2.c $\prec$ $o_1.c$. It is often whether or not this set is empty (ϕ) that matters. In addition, when inclusively or exclusively transforming o_1 and o_2, their effects relation should only depend on characters inserted or deleted by operations that happened before either o_1 or o_2 or both, but not by those operations that are concurrent with or happened after both o_1 and o_2. By causality, only the subset of landmark characters that appear in the system earlier than either $o_1.c$ or $o_2.c$ are necessary for determining their “natural” order.

In Section 4.3, and 4.4, we will establish a set of sufficient conditions that amend contextual equivalence and contextual serialization. In particular, we only need to identify conditions under which these basic IT/ET functions produce admissible results. Then in Section 5 we show how to build special transformation paths that satisfy those conditions such that operations can be correctly integrated. We do not need to ensure the correctness of IT/ET in all possible cases.

4.3. Sufficient conditions of IT

Theorem 3 *Given two operations, o_1 and o_2, that are defined and admissible in a reachable state s, then $o_1'=IT(o_1, o_2)$ is admissible in $s'=exec(s, o_2)$, if one of the following conditions holds*:

(1) $o_1 \cdot p \neq o_2 \cdot p$
(2) $(o_1 \cdot p = o_2 \cdot p) \wedge ((o_1 \cdot t = ins \wedge o_2 \cdot t = del) \vee (o_1 \cdot t = del \wedge o_2 \cdot t = ins) \vee (o_1 \cdot t = o_2 \cdot t = del))$
(3) $(o_1 \cdot p = o_2 \cdot p) \wedge (o_1 \cdot t = o_2 \cdot t = ins) \wedge (o_1 \| o_2) \wedge (C_{ld}(o_1 \cdot c, o_2 \cdot c) \neq \emptyset)$

This theorem is proved by the following three lemmas.

Lemma 3 *o_1' is admissible in s' if condition* (1) *holds.*

Proof Without loss of generality, assume $o_2.p<o_1.p$. We need to consider four cases: (1) $o_1.t=o_2.t=ins$, (2) $o_1.t=del \wedge o_2.t=ins$, (3) $o_1.t = ins \wedge o_2.t=del$, and (4) $o_1.t=o_2.t=del$. Not to be tedious, here we only prove case (1). Proofs of other cases are similar.

Let G be the consistent graph corresponding to s. Due to $o_2.p<o_1.p$, we have $o_2.p-1<o_2.p\leq o_1.p-1<o_1.p$. Consider the following four characters $c_a=s[o_2.p-1]$, $c_b=s[o_2.p]$, $c_d=s[o_1.p-1]$, and $c_e=s[o_1.p]$. Their relation is either $c_a\prec c_b\prec c_d\prec c_e$ or $c_a\prec c_b=c_d\prec c_e$.

Since o_2 is admissible in s, the invocation of o_2 on G produces a consistent graph G' and s' is also reachable. It introduces a new node containing $o_2.c$ and two new edges if they are not in G yet: one from c_a to $o_2.c$ and the other from $o_2.c$ to c_b. This does not change the order between c_a, c_b, c_d, and c_e. Relative to s', however, because of the insertion of $o_2.c$, we have $c_a=s'[o_2.p\text{-}1]$, $o_2.c=s'[o_2.p]$, $c_b=s'[o_2.p+1]$, $c_d=s'[o_1.p]$, and $c_e=s'[o_1.p+1]$. Their relation is either $c_a\prec o_2.c\prec c_b\prec c_d\prec c_e$ or $c_a\prec o_2.c\prec c_b=c_d\prec c_e$.

By IT defined in Function 1, we have $o_1'.p=o_1.p+1$. Hence $c_d=s'[o_1'.p-1]$, and $c_e=s'[o_1'.p]$. The invocation of o_1' on G' produces graph G'', which introduces a new node containing $o_1.c$ and two new edges (if they are not in G' yet): one from c_d to $o_1.c$ and the other from $o_1.c$ to c_e. Their relation is either $c_a\prec o_2.c\prec c_b\prec c_d\prec o_1.c\prec c_e$ or $c_a\prec o_2.c\prec c_b=c_d\prec o_1.c\prec c_e$. No new cycle is introduced by o_1'. Hence G'' is also consistent, which means that o_1' is admissible in s'. □

Lemma 4 *o_1' is admissible in s' if condition* (2) *holds.*

Proof Because one operation, say o_2, is a delete, by Theorem 1, $o_2.c$ is already in state s. Then we only need to determine the relation between $o_2.c$ and the other character $o_1.c$. The proof is easy and omitted. □

Lemma 5 *o_1' is admissible in s' if condition (3) holds.*

Proof Since the assertion must hold in spite of specific execution order of concurrent operations, we consider a simple case in which the effects of all operations concurrent with both o_1 and o_2 are not in s.

Let G be the consistent graph corresponding to state s, c_a=$s[o_2.p-1]$, and c_b= $s[o_2.p]$. The invocation of o_2 on G yields a consistent graph G', which introduces a new node containing $o_2.c$ and two new edges (if they are not in G yet): one from c_a to $o_2.c$ and the other from $o_2.c$ to c_b. Due to $o_1.p$=$o_2.p$, the invocation of o_1 in G would similarly yield a consistent graph with a new node containing $o_1.c$ and two new edges: one from c_a to $o_1.c$ and the other from $o_1.c$ to c_b. Now invoke o_1 on G' and add node $o_1.c$ and edges $<c_a, o_1.c>$ and $<o_1.c, c_b>$ first without considering the relation between $o_1.c$ and $o_2.c$. Let the resulting graph be G''.

Due to the analyses of Example 3 in Section 4.2, we know that the landmark characters in the graph are not always present in the current execution state s. However, if we can somehow find a transformation path such that state s includes all possible landmark characters, namely, $C_{ld}(o_1.c,o_2.c)\subseteq s$, deciding the relation between $o_1.c$ and $o_2.c$ becomes straightforward. Specifically, if $C_{ld}(o_1.c,o_2.c)\neq\emptyset$, there must be at least one landmark character between $o_1.c$ and $o_2.c$ in s and G. This implies $o_1.p\neq o_2$.p, which is the case of condition (1). On the other hand, if $C_{ld}(o_1.c, o_2.c)=\emptyset$, the node of $o_1.c$ and the node of $o_2.c$ are unorderable in G''. Then only adding an edge between node $o_1.c$ and node $o_2.c$ introduce no new cycle in G''. According to the IT definition in Function 1, if $o_1.id<o_2.id$, an edge $<o_1.c, o_2.c>$ is added; or otherwise $<o_2.c, o_1.c>$ is added. □

By the above proofs, we draw the following important corollary, in which condition $C_{ld}(o_1.c,o_2.c)\subseteq s$ means that the landmark characters between $o_1.c$ and $o_2.c$, if any, must be present in s. Note that in $C_{ld}(o_1.c,o_2.c)$ we only need to consider effect characters of operations that happened before either o_1 or o_2 or both.

Corollary 2 *Given two insert operations, o_1 and o_2, that are defined and admissible in a reachable state s, if $o_1 \parallel o_2$ and $C_{ld}(o_1.c,o_2.c)\subseteq s$, then o_1'=*IT(o_1,o_2) *is admissible in s'=*exec(s,o_2).

4.4. Sufficient conditions of ET and SWAP

Theorem 4 *Given two operations, o_1 and o_2, where o_2 is defined and admissible in a reachable state s and o_1 is defined and admissible in s'= exec(s, o_2), then o_1'= ET(o_1, o_2) is admissible in s, if one of the following conditions holds:*

(1) $o_1 \cdot p \neq o_2 \cdot p$
(2) $(o_1 \cdot p = o_2 \cdot p) \wedge (o_1 \cdot t = o_2 \cdot t = ins)$
(3) $(o_1 \cdot p = o_2 \cdot p) \wedge (o_1 \cdot t = o_2 \cdot t = del)$
(4) $(o_1 \cdot p = o_2 \cdot p) \wedge (o_1 \cdot t = ins) \wedge (o_2 \cdot t = del) \wedge (o_2 \rightarrow o_1) \wedge (o_1 \textit{ is generated in } s')$

This theorem is proved by the following four lemmas. Not to be tedious, we only show how to prove Lemmas 7 and 9. Proofs of the other two are similar.

Lemma 6 o_1' *is admissible in s if condition* (1) *holds.*

Lemma 7 o_1' *is admissible in s if condition* (2) *holds.*

Proof Since o_2 is admissible in s, the graph G corresponding to s must be acyclic. Let $c_b=s[o_2.p-1]$ and $c_a=s[o_2.p]$, if they exist. The invocation of o_2 adds a new node containing $o_2.c$ and two edges $<c_b, o_2.c>$ and $<o_2.c, o_a>$, if they are not in G. Let the resulting graph be G', which corresponds to s'. Since o_1 is admissible in s', its invocation must generates an acyclic graph G'' which contains a new node $o_1.c$ and two edges $<c_b, o_1.c>$ and $<o_1.c, o_2.c>$. By the ET definition in Function 2, o_1'=ET(o_1, o_2) is defined in state s and o_1' $p=o_1.p = o_2.p$. That is, the execution of o_1' in s adds two edges $<c_b,o_1.c>$ and $<o_1.c, c_a>$ into G''. This invocation does not introduce a new cycle given that there is no cycle between c_b and c_a in G. Hence o_1' is admissible in s. □

Lemma 8 o_1' *is admissible in s if condition* (3) *holds.*

Lemma 9 o_1' *is admissible in s if condition* (4) *holds.*

Proof Since o_2 is admissible in s, the graph G corresponding to s must be consistent and $o_2.c=s[o_2.p]$ must hold. Let $c_b=s[o_2.p\text{-}1]$ and $c_a=s[o_2.p+1]$, if they exist. There must be two edges $<c_b, o_2.c>$ and $<o_2.c, c_a >$ in G. After invoking o_2, we have $c_a =s'[o_2.p]$. Since the invocation of o_2 only decrement the counter of node $o_2.c$, these two edges are also in graph G' that corresponds to s'. Since o_1 is admissible in s' and $o_1.p=o_2.p$, invocation of o_1 yields a new node $o_1.c$ and two edges $<c_b, o_1.c>$ and $<o_1.c, c_a >$ in the resulting acyclic graph G''.

By policy P_3 in the ET definition (Function 2), position parameter $o_1'.p =o_1.p$ is defined in s. Invocation of o_1' in s adds a new edge $<o_1. c, o_2.c>$ in G'' and results in a path from c_b to $o_2.c$. Since o_1 is generated in s', node $o_1.c$ is only connected from c_b and to c_a in G''. If adding edge $<o_1.c, o_2.c>$ introduces a new cycle, there must exist a path from $o_2.c$ to c_b in G'', which contradicts the fact that there is already an edge $<c_b, o_2.c>$ in G'' and G'' is acyclic. Therefore, o_1' is admissible in state s. □

Additionally, we can show that condition (4) actually implies that $C_{ld}(o_1.c, o_2.c)=\emptyset$, which justifies the use of policy P_3 in the above proof. Otherwise, $C_{ld}(o_1.c, o_2.c)\neq\emptyset$ and there must exist at least one landmark character $c \in s'$ that is directly related to the new character $o_1.c$ generated by o_1 in s' and c must be between the position of $o_2.c$ and $o_1.c$. That is, $o_1.p\neq o_2.p$, which contradicts $o_1.p=o_2.p$.

We have explained in Section 4.1 that ET and SWAP have the same sufficient conditions. SWAP(o_1, o_2) is conceptually equivalent to first doing o_1'=ET(o_1, o_2)

and then o'_2=IT(o_2, o'_1). That is, if o'_1=ET(o_1, o_2) is admissible, then the order between $o_1.c$ and $o_2.c$ is known and thus o'_2=IT(o_2, o'_1) is also admissible.

Corollary 3 *Suppose that two operations o_1 and o_2 are both admissible and $o_2 \mapsto o_1$*. Let <o'_1 ,o'_2>=SWAP(o_1, o_2). *Then $o'_1 \mapsto o'_2$ and both o'_1 and o'_2 are admissible under the same conditions (1–4) specified in Theorem 4.*

4.5. Conditions of reordering sequences

In this subsection, we study a few special operation sequences and their conditions. Those sequences will be used in Section 5 for constructing transformation paths.

Definition 14 [**CP Sequence**] A sequence *sq* is causality-preserving (CP) iff for any two operations *sq*[*i*] and *sq*[*j*], where $0 \le i, j < |sq|$, if $sq[i] \rightarrow sq[j]$, then $i<j$ must hold. Any sequence *sq* is trivially a CP sequence if $|sq|<2$.

In a causality-preserving sequence *sq*, for any $o \in sq$, all operations in *sq* that happened before *o* must precede *o*.

For any two operations, o_1 and o_2, we say o_1 depends on o_2, if o_1 deletes the character inserted by o_2, i.e., $o_2.t=ins$, $o_1.t=del$, $o_1.c=o_2.c$ and $o_2 \rightarrow o_1$.

Definition 15 [**DP Sequence**] A sequence sq is dependency-preserving (DP) iff for any two operations *sq[i]* and *sq[j]*, where $0 \le i, j < |sq|$, if *sq[j]* depends on *sq [i]*, then *i<j must hold. Any sequence sq is trivially a DP sequence if* $|sq| < 2$.

In a dependency-preserving sequence *sq*, any operation o_j that depends on any o_i must follow o_i in *sq*. However, note that o_j is not required to immediately follow o_i. The cause-effect relation is only preserved between pairs of operations that have dependencies. Hence a causality-preserving sequence must also be dependency preserving, but not vice versa.

Definition 16 [**ID Sequence**] A sequence *sq* is an insertion-deletion (ID) sequence, iff for any two operations *sq[i]* and *sq[j]*, where $0 \le i, j < |sq|$, we have $i<j$ if either (1) $sq[i].t=ins \wedge sq[j].t=del$, or (2) $sq[i].t=sq[j].t \wedge sq[i] \rightarrow sq[j]$. Any sequence *sq* is trivially an ID sequence if $|sq|<2$.

Let sq_1 and sq_2 be two operation sequences. Notation $sq_1 \bullet sq_2$, where $sq_1 \mapsto sq_2$, or dst(sq_2)=exec(dst(sq_1),sq_1), returns the concatenation of sq_1 and sq_2; notation $sq \bullet o$, where $sq \mapsto o$, or dst(o)=exec(dst(sq), sq), returns the concatenation of sequence *sq* and operation *o*. By definition, an *ID* sequence *sq* is the concatenation of an insertion subsequence sq_i and a deletion subsequence

sq_d, or $sq=sq_i \bullet sq_d$. Both sq_i and sq_d are causality-preserving. Sequence sq is dependency-preserving but not necessarily causality-preserving.

Definition 17 [**HC and IHC Sequences**] Given an operation o and a sequence sq, we say that sq is a happened-before-concurrent (HC) sequence with regard to o, iff for any two operations $sq[i]$ and $sq[j]$, where $0 \leq i, j < |sq|$, we have $i<j$ if either (1) $sq[i] \rightarrow o \wedge sq[j] \parallel o$, or (2) $sq[i] \rightarrow sq[j]$. Any sequence sq is trivially an HC sequence if $|sq| < 2$. If all operations in an HC sequence sq are insertions, we say that sq is an IHC sequence.

By definition, an *HC* sequence sq (with regard to o) is a concatenation of two subsequences $sq=sq_h \bullet sq_c$, where sq_h includes all operations in sq that happened before o and sq_c includes all operations in sq that are concurrent with o. Note that sq, sq_h and sq_c are all causality-preserving sequences.

In a group editor, every site maintains a history buffer H which records operations in their order of execution at that site. If every operation in H is admissible, then H is an admissible sequence. As a convention, s_0 denotes the initial state.

Theorem 5 *Suppose that H is an admissible history and sq is an ID sequence such that* $sq \approx H$. *Let o be an operation that is generated in state* $s=exec(s_0, H)$. *There must exist an admissible ID sequence sq' such that* $sq' \approx (sq \bullet o)$.

Proof The fact that o is generated in state s implies that all operations in H happened before o. If o is a deletion, we get sq' simply by appending o to sq, or $sq'=(sq \bullet o)$ and clearly sq' is an admissible ID sequence. If o is an insertion, we rewrite $sq=sq_i \bullet sq_d$, where sq_i includes all insertions in H and sq_d includes all deletions in H. We first swap o with every operation in sq_d from right to left, yielding sq'_d and o' as the result, and then get $sq'=((sq_i \bullet o') \bullet sq'_d)$. If the swappings are correct, then $sq' \approx (sq \bullet o)$.

Now we prove the swapping step is correct or, alternatively, the resulting o' and sq'_d are admissible. If $|sq_d|=0$, the assertion trivially holds. For $|sq_d|=n>0$, we swap o and $sq_d[n-1]$, $sq_d[n-2]$, ..., $sq_d[1]$, $sq_d[0]$ in turn. We prove the assertion by induction on k, the number of swaps that involve o, where $1<k<n=|sq_d|$.

k=1: consider $sq_d[n-1]$ and let it be d_1. Since o is generated in state dst(o), by Corollary 3, whether or not $o.p=d_1.p$, the results are admissible. Let the resulting form of o be $o^{(1)}$ and dst($o^{(1)}$)=$s^{(1)}$. It is obvious that $d_1.c \in s^{(1)}$.

$k \rightarrow (k+1)$: Assume that swapping o and $sq_d[n-k, n-1]$, the rightmost k operations in sq_d, is correct. As a result, suppose o becomes an admissible operation $o^{(k)}$ defined in state dst($o^{(k)}$)=$s^{(k)}$, which must include all the effect characters of $sq_d[n-k,n-1]$. Then consider swapping $o^{(k)}$ with the $(k+1)^{th}$ operation $sq_d[n\text{-}(k+1)]=d_{(k+1)}$. If $o^{(k)}.p \neq d_{(k+1)}.p$, by Corollary 3, the assertion

holds. If $o^{(k)}.p=d_{(k+1)}.p$, there must be no landmark character between $o^{(k)}.c$ and $d_{(k+1)}.c$, if any, present in $s^{(k)}$. We consider the following two cases:

First, C_{ld} $(o^{(k)}.c,\ d_{(k+1)}.c)=\emptyset$. Adding an edge between nodes containing these two characters by policy P_3 will not introduce a cycle.

Second, C_{ld} $(o^{(k)}.c,\ d_{(k+1)}.c)\neq\emptyset$. Without loss of generality, assume that there exists at least one landmark character x such that $d_{(k+1)}.c\prec x\prec o^{(k)}.c$ but $x\notin s^{(k)}$. Character x must not be an effect character of some deletion in $sq_d[n-k,n-1]$. Otherwise, it would have been put back in $s^{(k)}$ by the first k swappings (exclusion transformations). Since o is generated in s, characters that are inserted after the generation of o are out of the question. It is also impossible for x to be inserted after the execution of $d_{(k+1)}$ and before the generation of o because sq is an ID sequence and operations in $sq_d[n-k,n-1]$ are all deletions. Finally, if x is deleted by some operation in $sq_d[0,\ n-(k+2)]$, since $x\notin s^{(k)}$, there must be another character $y\in s^{(k)}$ such that $x\prec y\prec o^{(k)}.c$. By transitivity, we have $d_{(k+1)}.c\prec y\prec o^{(k)}.c$, which implies $o^{(k)}.p\neq d_{(k+1)}.p$, a contradiction. That is, the assumed landmark character x does not exist.

Therefore, the $(k+1)^{\text{th}}$ swapping is also correct. When o is an insertion, the resulting $(sq_i \bullet o') \bullet sq'_d$ is an admissible ID sequence of $(k+1)$ operations.

Theorem 6 Given an admissible operation o and an admissible CP sequence sq, where all operations in sq are insertions, there must exist an admissible *IHC* sequence sq' with regard to o such that $sq\approx sq'$.

Proof Figure 8. gives an algorithm for finding the required sq'. It scans the input sequence sq from left to right and appends every operation concurrent with o to sequence sq_c. Every operation that happened before o is swapped with sq_c and the result is appended to sequence sq_h. Obviously if sq is causality preserving, then sq_h and sq_c are each causality preserving. Since sq is an insertion sequence, SWAP is only used between insertions in the algorithm. By Corollary 3, this is correct. □

FUNCTION 4. **convert2HC**$(o,\ sq)$**:** sq'

```
1  sq_h ← ∅; sq_c ← ∅;        // empty sets
2  for(j = 0; j < |sq|; j++)
3      if sq[j] || o
4          sq_c ← sq_c • sq[j];
5      else //if sq[j] → o
6          for(k = |sq_c| − 1; k ≥ 0; k−−)
7              <sq[j], sq_c[k]> ← SWAP(sq[j], sq_c[k]);
8          sq_h ← sq_h • sq[j];
9  return sq' ← sq_h • sq_c;
```

Figure 8. To convert a CP sequence into an HC sequence.

The convert2HC function was adopted from previous work (Suleiman et al. 1997; Sun and Ellis 1998). It is understood that, in line 7, $sq[j] \parallel sq_c[k]$ must hold. Otherwise, the relation must be either $sq[j] \rightarrow sq_c[k]$ or $sq_c[k] \rightarrow sq[j]$. Since sq is a causality-preserving sequence, it is impossible to have $sq[j] \rightarrow sq_c[k]$. On the other hand, by line 5, we know $sq[j] \rightarrow o$. If $sq_c[k] \rightarrow sq[j]$, by transitivity, we would have $sq_c[k] \rightarrow o$, which contradicts $sq_c[k] \parallel o$.

5. Integrating local and remote operations

The conditions in Theorems 3, 4, 5 and 6 are sufficient because the assertions always hold once the conditions are satisfied. This section gives algorithms for integrating local and remote operations. The main idea is to ensure that every operation is admissible in its execution state by satisfying these sufficient conditions while avoiding conditions that are not covered. Section 5.1 presents the algorithms. Section 5.2 gives a complete example. Section 5.3 proves correctness.

5.1. Control procedure

We consider a group editor of N sites that start from the same initial state s_0. Every site maintains a state vector sv for achieving causality preservation. Every element of sv is an integer that is initialized as zero. The *ith* element, denoted as $sv[i]$, where i=1, 2, ..., N, is the current site's knowledge of the number of operations that have been executed at site i. Each time an operation o is generated at site i, it is executed immediately and $sv[i]$ is incremented by one. Then o is timestamped by sv (by assigning sv to attribute $o.v$) and broadcast to remote sites. Each time a remote operation generated at site i is integrated, $sv[i]$ is incremented by one. The vector timestamps are used for determining the $\parallel$ and $\rightarrow$ relations between operations (Ellis and Gibbs 1989).

Each site maintains an operation log H, which is an *ID* sequence. H is often rewritten as $H=H_i \bullet H_d$, where H_i includes all the insertions and H_d all the deletions. Note that this paper does not address group undo. Hence we omit the other log that is usually present in group editors (e.g., (Sun 2002)) that stores all operations in their order of execution.

Figure 9 gives the control procedure at site i. Every time a local operation o is generated, we always execute it directly in its generation state s=exec(s_0, H). All operations in H happened before o and o is admissible in s. After o is executed, we call updateHL(o, H) to add o to H and compute its admissible form o' in state s'=exec(s_0, H_i), the state just after only all insertions (that happened before o) have been invoked in s_0. Then we propagate o' to remote sites.

Each site maintains a queue RQ to store remote operations that are received from other sites. Every received remote operation is appended to RQ. To ensure causality preservation, a remote operation is invoked only when it is causally

```
(a) Initialize:
(a.1)  RQ←∅; H←∅;
(a.2)  sv←< 0, 0, ..., 0 >

(b) Invoke (generate) a local operation o in state s:
(b.1)  s ← exec(s, o);
(b.2)  sv[i] ← sv[i] + 1;
(b.3)  o.v ← sv;
(b.4)  < o′, H > ← updateHL(o, H);
(b.5)  broadcast o′ to other sites;

(c) Receive o from network:
(c.1)  RQ ← RQ • o;

(d) Invoke a causally-ready remote operation o in s:
(d.1)  < o′, H > ← updateHR(o, H);
(d.2)  s ← exec(s, o′) if( o′ ≠ φ );
(d.3)  sv[j] ← sv[j]+1 where j = o.id;
```

Figure 9. The control procedure at site i.

ready (Sun et al. 1998): Given a remote operation o generated at site i, o is causally ready at site j, iff (1) $o.v[i]=sv_j[i]+1$ and (2) for $\forall k\neq i$: $o.v[k]\leq sv_j[k]$.

We scan RQ from left to right for the first causally-ready remote operation o. Then o is removed from RQ and function updateHR(o, H) is called to add o to H and compute its admissible form o' in current state s=exec(s_0,H). If o' is not ϕ, we execute o' in s. In fact, o' is ϕ only when two sites concurrently attempt to delete the same character. In this case, to avoid the appearance of grey nodes in the (conceptual) effect relation graph, the deletion to be invoked later is transformed into an identity operation ϕ and discarded.

Figure 10 defines function updateHL(o, H), which serves two purposes: (1) to compute o', the admissible form of o in state s'=exec(s_0, H_i), and (2) to add o into H. We know that $H=H_i \bullet H_d$ is an ID sequence and o is generated in state s=exec (s_0, H). Hence its admissible form o' relative to s' can be computed by swapping o from right to left with every operation in H_d. Then if $o.t=del$, we append the

FUNCTION 5. **updateHL(o, H): <o',H'>**

```
1  o′ ← o; L ← H;
2  for(i = |H_d| − 1; i ≥ 0; i − −)
3        < o′, H_d[i] > ← SWAP(o′, H_d[i]);
4  if o.t = del
5        H′ ← L • o;
6  else // o.t = ins
7        H′ ← H_i • o′ • H_d;
8  return <o′,H′>;
```

Figure 10. To add a local operation o into H and make o' admissible in state s'=exec(s_0, H_i).

original form o to H; or if $o.t{=}ins$, we append o' to H_i. This is actually the algorithm used in the proof of Theorem 5.

Figure 11 defines function updateHR(o, H), which serves two purposes: (1) to compute o', the admissible form of o in current state s=exec(s_0, H), and (2) to add o into H. When a remote operation o is received, according to function updateHL (), it must have excluded the effects of all deletions that happened before o and have included the effects of all insertions that happened before o. However, o must have included neither the effects of operations in H_d nor the effects of operations in H_i that are concurrent with o.

Therefore, we first call function convert2HC(o,H_i) to transpose H_i into an IHC sequence $H_{ih} \bullet H_{ic}$ with regard to o. Due to causality preservation and function updateHL(), o must be admissible in state exec(s_0, H_{ih}). In line 2 we transform o such that the result o'' is admissible in state s'=exec(s_0, H_i) by calling o''=ITSQ(o, H_{ic}) to include the effects of H_{ic} into o. Then in line 3 we perform o'=ITSQ(o'', H_d) such that o' is admissible in current state s.

After that, we add o into H as follows: If $o'{=}\phi$, it is not added into H. If o' is a deletion, we append o' to H directly. If o' is an insertion, we add o'' between H_i and H_d (line 12). To achieve this, we have to include the effect of o'' into every operation in H_d, as in lines 7–11. Initially we have $o'' \sqcup H_d[0]$ and $o_x{=}o''$. For every $H_d[k]$, since $o_x \sqcup H_d[k]$, we include the effect of $H_d[k]$ into o_x and, at the same time, include the effect of o_x into $H_d[k]$ such that the resulting o_x satisfies $H_d[k{+}1] \sqcup o_x$ and next time we still inclusively transform two context equivalent operations $H_d[k{+}1]$ and o_x.

FUNCTION 6. **ITSQ**(o, sq)**:** o'

```
1   for(i=0; i < |sq|; i++)
2       o ← IT(o, sq[i]);
3       if( o = φ ) break;
4   return o' ← o;
```

FUNCTION 7. **updateHR**(o, H)**:** <o',H'>

```
1   H_ih • H_ic ← convert2HC(o, H_i);
2   o'' ← ITSQ(o, H_ic);
3   o' ← ITSQ(o'', H_d);
4   if o' = φ:   H' ← H
5   else if o'.t = del:   H' ← H • o';
6   else if o'.t = ins
7       o_x ← o'';
8       for(k = 0; k < |H_d|; k++)
9           o_y ← o_x;
10          o_x ← IT(o_x, H_d[k]);
11          H_d[k] ← IT(H_d[k], o_y);
12      H' ← H_i • o'' • H_d;
13  return <o',H'>;
```

Figure 11. To add a remote opreation o into H and make o' admissible in state s=exec(s_0, H).

5.2. An integrated example

Notations in this example follow last subsection. Suppose that three sites start from the same initial state s_0="abc". The three sites concurrently generates o_1= del(1,'b'), o_2=ins(2,'x') and o_3=ins(1,'y'), respectively. Site 1 generates o_4=del (0,'a') after integrating o_1, o_2 and o_3. Site 2 generates o_5=del(0,'a') after integrating o_2 and o_1. Site 3 generates o_6=ins(2,'z') after integrating o_3, o_2 and o_1. Eventually all sites converge in state s_6="yzxc". These operations are integrated as in Figure 12, which is explained in three stages.

5.2.1. *Stage one*

Initially the history buffer at each site is empty. Hence o_1, o_2 and o_3 are propagated to remote sites as they are. The effects graph after local invocations of these three operations is as shown in Figure 12.

At site 1, after executing o_1 locally, its state is s_1^1 ="ac" and history buffer is H_1^1 =[o_1]. When o_2 is received, since there is no insertion in H_1^1, we have o_2''=o_2 and o_2'=IT(o_2, o_1)=ins(1,'x'). After executing o_2', the state is s_1^2 ="axc". Then o_2'' is added into H_1^1, which yields H_2^1 =[o_2'' ,o_1'], where o_1'=IT(o_1, o_2'')=del(1,'b')=o_1. To be concise, we rewrite H_2^1 =[o_2, o_1]. When o_3 arrives, there is an insertion and a deletion in H_2^1. We first get o_3''=IT(o_3,o_2)=ins(1,'y')=o_3 and then o_3'=IT(o_3'' ,o_1)= ins(1,'y')=o_3 . After executing o_3', the state becomes s_3^1 ="ayxc". Adding o_3''

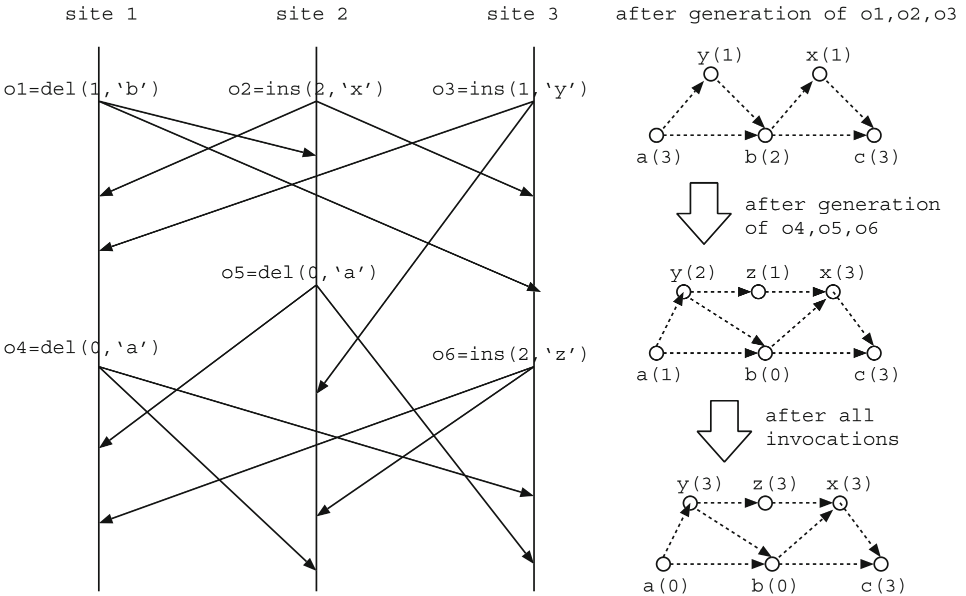

Figure 12. Three sites start from s_0="abc" and converge in s_6="yzxc".

between o_2 and o_1 turns o_1 into o'_1=IT(o_1, o''_3)=del(2,'b'). The history buffer becomes H^1_3 =[o_2,o_3, o'_1], where o'_1=del(2,'b').

At site 2, after executing o_2 locally, its state is s^2_1 ="abxc" and history buffer is H^2_1 =[o_2]. When o_1 arrives, we have o'_1=o''_1=IT(o_1,o_2)=del(1,'b')=o_1. The state after executing o_1 is s^2_2 ="axc" and the history buffer is H^2_2 =[o_2,o_1].

At site 3, after executing o_3 locally, the state is s^3_1 ="aybc" and history buffer H^3_1 =[o_3]. When o_2 arrives, we get o'_2=IT(o_2,o_3)=ins(3,'x'). Executing o'_2 yields s^3_2 ="aybxc" and H^3_2 =[o_3, o'_2]. When o_1 arrives, we get o'_1=ITSQ(o_1,[o_3, o'_2])= del(2,'b'). Its execution yields s^3_3 ="ayxc" and H^3_3 =[o_3,o'_2, o'_1].

5.2.2. *Stage two*

Three concurrent operations o_4, o_5 and o_6 are generated at three sites, respectively. They are integrated and propagated as follows. The effects relation graph after locally invoking these operations is shown in Figure 12.

At site 1, when o_4 is generated, the history buffer is H^1_3 =[o_2, o_3, o'_1], where o'_1=del(2,'b'). The new state is s^1_4 =exec(s^1_3, o_4)="yxc". Appending o_4 to H^1_3 yields the new history buffer H^1_4 =[o_2, o_3, o'_1, o_4]. Next, we swap o_4 and o'_1 to get o'_4=ET(o_4, o'_1)=del(0,'a')=o_4. Then, we propagate o'_4 (or o_4) to remote sites.

At site 2, o_5 is generated in state s^2_2 ="axc" and H^2_2 =[o_2, o_1]. Integrating o_5 yields s^2_3 ="xc" and H^2_3 =[o_2, o_1, o_5]. Swapping o_5 and o_1 yields o'_5=ET(o_5,o_1)= del(0,'a')=o_{5a}. Hence we propagate o_5 to remote sites as-is.

At site 3, o_6 is generated in the context of s^3_3 ="ayxc" and H^3_3 =[o_3, o'_2, o'_1], where o'_2=ins(3,'x') and o'_1=del(2,'b'). Integrating o_6 results in s^3_4 ="ayzxc". Now swap o_6 and o'_1: first o'_6=ET(o_6,$o_{1'}$)=ins(2,'z')=o_6 and then o''_1=IT(o'_1, o'_6)= del(3,'b'). Adding o'_6 (or o_6) into H^3_3 results in H^3_4 =[o_3, o'_2, o_6, o''_1]. Then, we propagate o'_6 (or o_6).

5.2.3. *Stage three*

At site 1, when o_5 is received, the state is s^1_4 ="yxc" and the history is H^1_4 =[o_2, o_3,o'_1, o_4], where o'_1=del(2,'b'). We first get o''_5=IT(o_5,o_3)=del(0,'a') and then o'_5= ITSQ(o''_5 ,[o'_1 ,o_4])=ϕ. Therefore, o_5 is not executed nor added into the history. That is, $s^1_5 = s^1_4$ ="yxc" and $H^1_5 = H^1_4$ =[o_2, o_3, o'_1 ,o_4]. When o_6 is received, $o_2 \rightarrow o_6$ and $o_3 \rightarrow o_6$. Hence o''_6=o_6 and o'_6=ITSQ(o''_6, [o'_1 ,o_4])=ins(1,'z'). Adding o''_6 (or o_6) into H^1_5 yields H^1_6 =[o_2, o_3, o_6, o''_1, o_4], where o''_1=del(3,'b'). Executing o'_6 in s^1_5 yields s^1_6 ="yzxc".

At site 2, when o_3 arrives, the state is s^2_3 ="xc" and the history is H^2_3 =[o_2, o_1, o_5]. We first get o''_3=ITSQ(o'_3,[o_2])=ins(1,'y')=o_3 and then o'_3=ITSQ(o''_3,[o_1, o_5])= ins(0,'y'). Executing o'_3 yields s^2_4 ="yxc" and adding o''_3 to history yields H^2_4 =[o_2, o''_3, o'_1, o'_5]=[o_2, o_3, o'_1, o_5], where o'_1=del(2,'b') and o'_5 = del(0,'a')=o_5. When o_6 arrives, since $o_2 \rightarrow o_6$ and $o_3 \rightarrow o_6$, we get o''_6=o_6 and o'_6=ITSQ(o''_6, [o'_1, o_5])=ins (1,'z'). Executing o'_6 yields s^2_5 ="yzxc" and adding o''_6 to history yields H^2_5 =[o_2, o_3, o''_6, o''_1, o''_5]=[o_2, o_3, o_6, o''_1, o_5], where o''_1=del(3,'b') and o''_5 = del(0,'a')=o_5.

When o_4 arrives, since $o_2 \rightarrow o_4$, $o_3 \rightarrow o_4$ and $o_6 \parallel o_4$, we get =IT(o_4, o_6)=del(0,'a')= o_4 and o_4'=ITSQ(o_4'', [o_1'' ,o_5])=ϕ. Hence o_4 is not executed nor added into the history. That is, the final state is s_6^2 ="yzxc" and history is H_6^2 =[o_2, o_3, o_6, o_1'', o_5], where o_1''=del(3,'b').

At site 3, when o_4 arrives, the state is s_4^3 ="ayzxc" and history is H_4^3 =[o_3, o_2', o_6, o_2'] where o_2'=ins(3,'x') and o_1''=del(3,'b'). We get o_4''=ITSQ(o_4, o_6)=del (0,'a')=o_4 and o_4'=IT(o_4'', o_1'')=del(0,'a')=o_4. Executing o_4' yields s_5^3 ="yzxc" and adding o_4'' to history yields H_5^3 =[o_3, o_2', o_6, o_1'', o_4]. When o_5 arrives, since $o_3 \parallel o_5$, $o_2 \rightarrow o_5$ and $o_6 \parallel o_5$, we first need to transpose the insertion sequence [o_3, o_2', o_6] into [o_2, o_3', o_6']=[o_2, o_3, o_6]. Then we get o_5''=ITSQ(o_5,[o_3, o_6])=del(0,'a')= o_5 and o_5'=ITSQ(o_5'',[o_1'', o_4])=ϕ. Hence o_5' is not executed and o_5'' is not added into history. That is, the final state is s_6^3 ="yzxc" and history H_6^3 =[o_3, o_2', o_6, o_1'', o_4], where o_2'=ins(3,'x') and o_1''=del(3,'b').

It is obvious that all three sites converge in final state s_6="yzxc". The final effects relation is shown in Figure 12. From the graph we can see that there is no order only between two characters, 'z' and 'b'. This is not surprising because 'z' is inserted after 'b' is deleted at the same position and they never appear in the same state. Hence there is no need to decide an order between them. That is, the effects relation is only a partial order. However, when 'b' and 'z' appear in the same state as a result of undo, adding an edge between node 'z' to 'b' converts the relation into a total order.

5.3. Proof of correctness

According to Definition 12, a group editor must preserve causality and admissibility in every operation invocation. By our control procedure, causality is obviously preserved. Here we only need to prove admissibility preservation.

Theorem 7 *The control procedure in Figure 9 ensures that the invocation of any operation is admissible.*

Proof We prove this assertion by induction on n invocations at any site i. When n=1, it is trivially true. Assume that the first k invocations are admissible at current site. Consider the $(k+1)^{th}$ invocation o in the following two cases.

First, o is a local operation. By Assumption 1, o is admissible in current state in which it is generated. All the k operations in H happened before local operation o. To ensure the correctness of invoking forthcoming operations, we maintain H as an ID sequence $H_i \bullet H_d$ and call function updateHL(o ,H) to reorder $H \bullet o$ into another ID sequence H', the correctness of which is ensured by Theorem 5. Therefore H' is an admissible ID sequence of $(k+1)$ operations and the propagated o' is admissible in state exec(s_0, H_i). Note that the H_i here actually includes all insertions in H that happened before o because o is generated in current state s=exec(s_0, H).

Secondly, a remote operation o is invoked. Function updateHR(o, H) is called to make it admissible in the current state s=exec(s_0, H) and $H=H_i \bullet H_d$. In updateHR(), H_i is first transposed into an IHC sequence $H_{ih} \bullet H_{ic}$ with regard to o. Theorem 6 ensures the correctness of this step. Hence we have $H \approx (H_{ih} \bullet H_{ic}) \bullet H_d = H_{ih} \bullet (H_{ic} \bullet H_d)$. The next step of updateHR() is to compute o'=ITSQ(o, $H_{ic} \bullet H_d$). Due to causality preservation and function updateHL(), o is admissible in state s'= exec(s_0, H_{ih})=dst($H_{ic} \bullet H_d$) = dst(o). In s', no character has been deleted because only an insertion sequence H_{ih} has been executed. Since H_{ic} is also an insertion sequence, when processing o''=ITSQ(o, H_{ic}), all landmark characters are present and thus the result of every step is admissible due to Theorem 3 and Corollary 2. By Theorem 3, o'=ITSQ(o'', H_d) is also admissible.

Additionally, we prove that adding the remote operation o into an admissible ID sequence H with k operations results in an admissible ID sequence H' with (k+1) operations. By the control procedure, if $o'=\phi$, it is not added to H for simplicity because $H \bullet \phi \approx H$. If o is a deletion, we append o' to $H=H_i \bullet H_d$ and get $H'=H_i \bullet (H_d \bullet o')$, which is an admissible ID sequence. If o is an insertion, we add o'' between H_i and H_d, which entails including the effect of o'' into $H_d[k]$. Since IT is between an insertion and a deletion at each step, by Theorem 3, the resulting H_d is admissible and hence $(H_i \bullet o'') \bullet H_d$ is an admissible ID sequence. □

6. Comparisons with related works

This paper is a significant extension to its conference version in (Li and Li 2005). Specifically, Sections 1, 2, 4.2, 5.2, 6, and 7 are new; and the presentation of other sections has been refined. Section 2 conceptually compares the proposed approach with other approaches. This section makes more indepth and specific comparisons that cannot be made earlier without presenting details of ABT.

6.1. Transformation functions

The inclusion transformation (IT) and exclusion transformation (ET) functions given in Functions 1 and 2, respectively, are so called due to (Sun et al. 1998). IT as defined in Function 1 is similar to (Ellis and Gibbs 1989; Ressel et al. 1996) because they also only use the basic parameters of an operation o, namely, position ($o.p$), site id ($o.id$), and type ($o.t$). These definitions are simple and intuitive.

IT functions defined in SOCT2 (Suleiman et al. 1997) and SDT (Li and Li 2004) use explicit extra parameters, which take extra computation to derive. Specifically, for instance, SDT uses an extra parameter $o.\beta$ to denote the position of o relative to its last synchronization point (a previous state) when it ties with a concurrent operation in the "current" state. Computing the β parameters is costly although it is only needed when two insertions tie in IT (Li and Li 2008b).

Although IT and ET functions defined in GOT (Sun et al. 1998), GOTO (Sun and Ellis 1998; Sun 2002), TTF (Oster et al. 2006b), and LBT (Li and Li 2007), and ET functions defined in SDT (Li and Li 2004; Li and Li 2008a; Li and Li 2008b] do not use explicit extra parameters, they use extra internal data structures to save information that helps break ties in some cases.

The SWAP function (Function 3) resembles those in (Knister and Prakash 1994; Suleiman et al. 1997; Sun et al. 1998; Sun and Ellis 1998) with different specifications. However, (Knister and Prakash 1994) does not handle boundary cases in which the operation positions tie; (Sun et al. 1998; Sun and Ellis 1998) need extra internal data structures for breaking ties in some cases.

The transpose_bk function in SOCT2 (Suleiman et al. 1997; Suleiman et al. 1998) is also similar to our SWAP function. Note that SOCT2 only transforms two concurrent operations. Given two concurrent operations o_1 and o_2, consider the scenario in Example 5. The step o'_1=Transpose_fd(o_1, o_2) is executed before the swap step Transpose_bk(o'_1, o_2). The deleted landmark character is effectively recorded in the updated parameters of the returned insert operation and hence the tie could be correctly broken inside function Transpose_fd. By comparison, we do not need to explicitly record deleted characters for the same purpose.

Because of these subtle differences, it is necessary to clearly specify the three transformation functions for this paper to be self-contained and then study their preconditions to avoid ambiguity and confusions.

6.2. Design approaches

Many OT algorithms follow the approach proposed in (Ressel et al. 1996), including SOCT2 (Suleiman et al. 1997), GOTO (Sun and Ellis 1998), and SDT [Li and Li 2004). Typically, each time a remote operation o is to be integrated, it transposes the history H into an HC sequence $H_h \bullet H_c$ by calling convert2HC(o, H) and then inclusively transforms o with every operation in H_c. Since operations in H_c are allowed to be in different orders at different sites, IT is required to verify two transformation properties (TP1 and TP2) such that transforming o along arbitrary paths yields the same result, i.e., convergence (Ressel et al. 1996). Unfortunately, as revealed in (Li and Li 2008a), due to complicated case coverage, it is extremely difficult to develop formal proofs manually.

Several approaches, such as GOT (Sun et al. 1998), SOCT3/4 (Vidot et al. 2000), NICE (Shen and Sun 2002), TIBOT (Li et al. 2004), and COT (Sun and Sun 2006) are proposed to free TP2 by maintaining the same transformation path (total order of operations) at all sites every time an operation o is transformed. Although these algorithms can converge, those approaches are not always able to preserve the correct object order because they cannot prevent the loss of landmark characters in their transformation paths, as shown in (Li and Li 2008a; Li and Li 2007). COT as published does not address how to break the so-called false ties.

In general, the above approaches are developed under the framework of (Sun et al. 1998), in which the contraint of intention preservation is not intended to be a rigorous correctness condition for purposes of formal proofs. In the literature, correctness problems of OT are often attributed to so-called TP2 puzzle (Sun et al. 1998) or false tie (Sun and Sun 2006). This work to some extent provides theoretical analyses of those problems because it formally identifies and proves conditions under which transformation functions or control procedures may go wrong. The analyses inform design of the ABT algorithm and may provide insights in the design practice of other OT algorithms. The presented algorithm is but one of the possible embodiments of the design principles of our ABT framework.

In WOOT (Oster et al. 2006a) and TTF (Oster et al. 2006b), Oster et al propose approaches that differ from the above. According to (Oster et al. 2005b), WOOT uses a model checker to prove convergence by verifying all cases that involve up to four sites and five characters, which deservers further work with regard to convergence. TTF uses a theorem prover to verify TP1 and TP2, which are sufficient conditions for convergence by (Ressel et al. 1996). In both works, they explain the concept of operation intention somewhat between the interpretation of (Sun et al. 1998) and our definitions of operation effects relation (Li and Li 2004; Li and Li 2007; Li and Li 2005). Nevertheless, they do not provide proofs with regard to their interpretation of intention preservation. It seems that our formalization of effects relation (Li and Li 2007) or admissibility in this paper is compatible with and complementary to their approach of automated proofs. There is a potential that their approach and ours can leverage each other in future research.

LBT (Li and Li 2007) is the first work that builds special transformation paths (versus arbitrary paths). Each time a remote operation o is to be integrated, we first transpose H into an HC sequence $H_h \bullet H_c$. If o is a deletion, we inclusively transform o with H_c. If o is an insertion, however, we first transpose H_h into an ID sequence $H_{hi} \bullet H_{hd}$ and then transpose $H_{hd} \bullet H_c$ into another ID sequence $H_i \bullet H_d$. After that, o is first exclusively transformed with H_{hd} (the backward path) to exclude its effects and then inclusively transformed with $H_i \bullet H_d$ (the forward path) to include its effects. The correctness of IT is thus ensured: Since $H \approx H_{hi} \bullet H_i \bullet H_d$, when processing IT between insertion o with H_i, the landmark characters are all present. However, ET in the two transposition steps has to handle happened-before and concurrent operations ordered arbitrarily. The solution in LBT is to build ET-safe sequences by reordering the operations according to their effects relation. Consequently, ET is still very complicated.

In the ABT algorithm presented in this paper, the history H is maintained as an ID sequence $H_i \bullet H_d$ at every site and, before any operation o is propagated, the effects of H_d (the backward path), which contains all deletions that happened before o, have been excluded from o. As a result, when integrating a remote operation o, we only need to transpose H_i into an HC sequence $H_{ih} \bullet H_{ic}$ and then

inclusively transform o with $H_{ic} \bullet H_d$ (the forward path). Hence the correctness of IT is ensured as simply as in LBT. However, ET is no longer required to work on arbitrary paths: In function updateHL(), ET is only between an (insert or delete) operation and deletions that happened before it. In function updateHR(), ET is only between concurrent insertions. Therefore, the handling of ET in ABT is much simpler.

6.3. Complexities and performance

ABT does not need extra storage space beyond the history buffer. By comparison, GOTO (Sun et al. 1998; Sun 2002) need extra memory for handling the so-called "lossy IT" problem. Both GOTO and SOCT2 use function convert2HC(), which requires ET between an insert and a concurrent delete. As shown in Example 5, its correctness is not guaranteed unless some information is saved when IT is performed between concurrent operations. LBT needs to save and retrieve the effects relation between operations. The space complexity of ABT is only $O(|H|)$, while those of GOTO and SOCT2 are $O(|H|^2)$. However, it is also worth noting that the control procedure of GOTO and SOCT2 is more general under the established design framework of (Sun and Ellis 1998); the quadratic space complexity is derived basing on their provided transformation functions.

In ABT, the time to invoke a local operation is dominated by the execution of function updateHL(), which is obviously $O(|H_d|)$. By comparison, most other OT algorithms (except (Ellis and Gibbs 1989; Ressel et al. 1996)) to our knowledge invoke local operations immediately without any transformation, which takes constant time $O(1)$ and is conceptually more efficient. The time to invoke a remote operation is dominated by function updateHR(). Inside function updateHR(), line 2 takes time $O(|H_{ic}|)$, while line 3 and lines 8–11 both take $O(|H_d|)$. By Figure 8, the worst-case and expected execution time of convert 2HC (o, H_i) is $O(|H_i|^2)$. Hence the time to invoke a remote operation is $O(|H_i|^2+|H_d|)$.

Every time a remote operation is integrated, other algorithms (SOCT2, GOTO, SDT, and LBT) call convert2HC() to transpose the whole history H. Hence the time complexities of all those algorithms are at least in the order of magnitude of $O(|H|^2)$, which is slower than ABT at least by some factor determined by the ratio of insertions in H.As revealed in our recent analyses and experiments (Li and Li 2008b; Li and Li 2006), the integration time affects scalability, feedback and feedthrough times. Hence the efficiency improvement is meaningful.

WOOT (Oster et al. 2006a) needs to uniquely identify every character because it needs the information to break insertion ties. By their own analyses in (Oster et al. 2005b), its space complexity is linear in the size of the shared document since every W-character in WOOT is represented by a five-tuple, which is expensive when a large document is shared; its time complexity to integrate one (local or remote) operation is $O(|H|^3)$. By their algorithm specification, TTF (Oster et al. 2006b) requires $O(|H|)$ space and takes $O(|H|)$ to integrate one (local or remote)

operation. TTF keeps a record of deleted characters for breaking insertion ties, which is not required in this work. Nonetheless, we acknowledge that it represents a tradeoff between space and time complexity when a remote operation is integrated. In general, our work (Li and Li 2004; and Li and Li 2007; Li and Li 2005) complements theirs with regard to correctness conditions and formal proofs.

According to (Sun et al. 1998; Li and Li 2008b; Li and Li 2006), OT algorithms must maintain a relative small history H to achieve interactive responsiveness in real-time group editors. A garbage collection (GC) scheme (Sun et al. 1998) can be adopted to periodically remove a prefix of H. Other OT algorithms store operations in H in execution order and only need to remove the identified prefix of H. They must also remove those operations from the extra storage space on which their correctness depends, which however has not been discussed in the literature. By comparison, GC in ABT is more complicated on the history but it does not have the memory part. Since $H=H_i \bullet H_d$ and both H_i and H_d store operations in execution order, it is easy to identify the prefix of H_i and the prefix of H_d that are to be removed. Note, however, that we cannot remove the prefixes directly because the removal changes the definition states of the remaining subsequences in H_i and H_d. To reduce time complexities, we can first remove the prefix of H_d, adjust the positions of operations in the remainder of H_d, then remove the prefix of H_i and finally adjust the positions of operations in the remainders of H_i and H_d. Correctness of this process is ensured by Theorem 4. Nevertheless, this GC process is more complex than in previous works (e.g., (Sun et al. 1998)). A careful scheduling is necessary in implementation so as not to degrade local responsiveness.

7. Conclusions

This paper contributes a novel framework for developing OT algorithms. Theoretically, we formalize an alternative correctness criterion, called admissibility preservation, based on a new graph-based analysis tool. It only requires that an OT algorithm integrate every operation in an admissible manner, i.e., without violating the character order established earlier by the algorithm itself. Compared to the established model of (Sun et al. 1998), admissibility is formalized and can be proved. Compared to our early results (Li and Li 2004; Li and Li 2007), it no longer requires a predefined total order of characters in the consistency model, which in turn greatly simplifies algorithm design and proofs; the partial order used in this work does not involve algorithm specific tie-breaking policies and hence can be used for verifying other algorithms as well.

Practically, this work establishes a principled methodology for developing and proving OT algorithms. In this approach, it first identifies sufficient conditions for basic transformation functions and then builds special transformation paths to synergistically ensure correctness. Compared to previous works, our transforma-

tion functions do not need to work correctly in all cases, which makes it possible to just use the most simplified and intuitive transformation functions, and the control procedure does not need to save extra information for ensuring correctness. As a result, the algorithm is lucid, without hidden details and costs, and completely proved, without lurking correctness puzzles.

This paper focuses on the theoretical aspect of OT and is limited to group do in collaborative editing systems that support two characterwise primitives on a linear document. As revealed in the literature (Sun et al. 1998; Sun 2002; Davis et al. 2002; Sun et al. 2004), group do is the foundation of group undo, characterwise operations are the foundation of stringwise operations, pure text editors are the foundation of more advanced editors. Based on the new foundation laid by this work, more recently we have shifted focus to the performance of OT in mobile applications. Our latest algorithm supports stringwise operations and sequence transformation, which is able to integrate a remote operation or even a sequence in $O(|H|)$ time. In addition, our experiments show that, although ABT takes $O(|H_d|)$ or $O(|H|)$ to invoke a local operation, the real execution time (even on a Nokia N810 tablet) is less than 4 milliseconds even when $|H|$ is greater than 5,000. Hence the impact on local responsiveness is negligible in practice. Those results will be reported in separate publications. Our ongoing work is investigating how to extend ABT for supporting other data structures and group undo. With correctness criteria formalized and OT algorithms growing more intricate, automated runtime verification (e.g., as in (Godefroid et al. 2000)) will also be a complementary direction to explore.

Acknowledgements

The authors thank the anonymous reviewers for their insightful, constructive, and detailed feedback, which improves the presentation of this work and our understanding of some of the related works. The authors also thank other researchers, especially Bin Shao (Fudan University, China) and David Sun (University of California, Berkeley), for valuable discussions. This research was primarily conducted when the authors were at Texas A&M University. It was supported in part by the National Science Foundation under CAREER award IIS-0133871.

References

Begole, J. B., Rosson, M. B., & Shaffer, C. A. (1999). Flexible collaboration transparency: supporting worker independence in replicated application-sharing systems. *ACM Transactions on Computer-Human Interaction, 6*(2), 95–132.

Bellini, P., Nesi, P., & Spinu, M. B. (2002). Cooperative visual manipulation of music notation. *ACM Transactions on Computer-Human Interaction, 9*(3), 194–237.

Davis, A. H., Sun, C., & Lu, J. (2002). Generalizing operational transformation to the standard general markup language. In *ACM CSCW'02 Conference on Computer-Supported Cooperative Work* (Nov. 2002), (pp. 58–67).

Ellis, C. A. & Gibbs, S. J. (1989). Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD'89 Conference on Management of Data*. Portland Oregon, 1989, (pp. 399–407).

Ellis, C. A., Gibbs, S. J., & Rein, G. L. (1991). Groupware: Some issues and experiences. *Communications of the ACM, 34*(1), 38–58.

Godefroid, P., Herbsleb, J. D., Jagadeesan, L. J., & Li, D. (2000). Ensuring privacy in presence awareness systems: An automated verification approach. In *ACM CSCW'2000 Conference Proceedings* (Philadelphia, Dec. 2000), (pp. 59–68).

Hymes, C.M. & Olson, G. M. (1992). Unblocking brainstorming through the use of simple group editor. In *ACM CSCW'92 Proceedings* (Nov. 1992), (pp. 99–106).

Imine, A., Molli, P., Oster, G., & Rusinowitch, M. (2003). Proving correctness of transformation functions in real-time groupware. In *Proceedings of the European Conference on Computer Supported Cooperative Work (ECSCW'03)* (Sept. 2003).

Imine, A., Rusinowitch, M., Oster, G., & Mollis, P. (2006). Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science, 351*(2), 167–183.

Knister, M. J., & Prakash, A. (1994). A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction, 1*(4), 295–330.

Li, D. & Li, R. (2004). Preserving operation effects relation in group editors. In *Proceedings of the ACM CSCW'04 Conference on Computer-Supported Cooperative Work* (Nov. 2004), (pp. 457–466).

Li, D. & Li, R. (2006). A performance study of group editing algorithms. In *The 12th International Conference on Parallel and Distributed Systems (ICPADS'06)* (Minneapolis, MN, July 2006), (pp. 300–307).

Li, D., & Li, R. (2008a). An approach to ensuring consistency in peer-to-peer real-time group editors. *Computer Supported Cooperative Work: The Journal of Collaborative Computing, 17* (5–6), 553–611.

Li, D., & Li, R. (2008b). An operational transformation algorithm and performance evaluation. *Computer-Supported Cooperative Work: The Journal of Collaborative Computing, 17*(5–6), 469–508.

Li, R. & Li, D. (2005). Commutativity-based concurrency control in groupware. In *Proceedings of the First IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom'05)* (San Jose, CA, Dec. 2005).

Li, R., & Li, D. (2007). A new operational transformation framework for real-time group editors. *IEEE Transactions on Parallel and Distributed Systems, 18*(3), 307–319.

Li, R., Li, D., & Sun, C. (2004). A time interval based consistency control algorithm for interactive groupware applications. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)* (July 2004), (pp. 429–436).

Molli, P., Oster, G., Skaf-Molli, H., & Imine, A. (2003). Using the transformational approach to build a safe and generic data synchronizer. In *GROUP '03: Proceedings of the 2003 International ACM SIGGROUP Conference on Supporting Group Work* (New York, NY, USA, 2003), (pp. 212–220).

Oster, G., Urso, P., Molli, P., & Imine, A. (2005a). Proving correctness of transformation functions in collaborative editing systems. Technical Report 5795 (Dec.), INRIA.

Oster, G., Urso, P., Molli, P., & Imine, A. (2005b). Real-Time Group Editors Without Operational Transformation. Research Report RR-5580 (May), LORIA — INRIA Lorraine.

Oster, G., Urso, P., Molli, P., & Imine, A. (2006a). Data consistency for P2P collaborative editing. In *Proceedings of the 20th Anniversary Conference on Computer-Supported Cooperative Work* (Banff, Alberta, Canada, Nov. 2006), (pp. 259–268).

Oster, G., Urso, P., Molli, P., & Imine, A. (2006b). Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *The Second International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2006)* (Atlanta, Georgia, USA, November 2006). IEEE Press.

Ressel, M., Nitsche-Ruhland, D., & Gunzenhäuser (1996). An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the ACM CSCW'96 Conference on Computer-Supported Cooperative Work* (Nov. 1996), (pp. 288–297).

Shen, H. & Sun, C. (2002). Flexible notification for collaborative systems. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (Nov. 2002), (pp. 77–86).

Suleiman, M., Cart, M., & Ferrié, J. (1997). Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the ACM GROUP'97 Conference on Supporting Group Work* (Phoenix, AZ, Nov. 1997), (pp. 435–445).

Suleiman, M., Cart, M., & Ferrié, J. (1998). Concurrent operations in a distributed and mobile collaborative environment. In *IEEE ICDE'98 International Conference on Data Engineering* (Feb. 1998), (pp. 36–45).

Sun, C. (2002). Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction, 9*(4), 309–361.

Sun, C., & Chen, D. (2002). Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction, 9*(1), 1–41.

Sun, C. & Ellis, C. (1998). Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work* (Dec. 1998), (pp. 59–68).

Sun, C., Jia, X., Zhang, Y., Yang, Y., & Chen, D. (1998). Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction, 5*(1), 63–108.

Sun, C., Xia, S., Sun, D., Chen, D., Shen, H., & Cai, W. (2006). Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction, 13*(4), 531–582.

Sun, D. & Sun, C. (2006). Operation context and context-based operational transformation. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW'06)* (Nov. 2006), (pp. 279–288).

Sun, D., Xia, S., Sun, C., & Chen, D. (2004). Operational transformation for collaborative word processing. In *Proceedings of ACM CSCW'04 Conference on Computer-Supported Cooperative Work* (Nov. 2004), (pp. 162–171).

Vidot, N., Cart, M., Ferrie, J., and Suleiman, M. (2000). Copies convergence in a distributed realtime collaborative environment. In *Proceedings of ACM CSCW'00 Conference on Computer-Supported Cooperative Work* (Dec. 2000), (pp. 171–180).