Comput Supported Coop Work (2008) 17:469–508
DOI 10.1007/s10606-005-9008-6
© Springer 2006

An Operational Transformation Algorithm and Performance Evaluation

DU LI & RUI LI
Department of Computer Science, Texas A&M University, College Station, TX, 77843-3112, USA (E-mail: lidu@cs.tamu.edu)

Abstract. Operational transformation (OT) is an optimistic concurrency control method that has been well established in realtime group editors and has drawn significant research attention in the past decade. It is generally believed that the use of OT automatically achieves high local responsiveness in group editors. However, no performance study has been reported previously on OT algorithms to the best of our knowledge. This paper extends a recent OT algorithm and studies its performance. By theoretical analyses and performance experiments, this paper proves that the worst-case execution time of OT only appears in rare cases, and shows that local responsiveness of OT-based group editors in fact depends on a number of factors such as the size of the operation log. The paper also reveals that these two results have general implications on OT algorithms and hence the design of OT-based group editors must pay attention to performance issues.

Key words: concurrency control, consistency, group editing, operational transformation, performance evaluation

1. Introduction

Realtime group editors are a classic model of interactive groupware applications that feature coordinated manipulation of shared data (Ellis et al., 1991; Prasun et al., 1994). They are also useful tools that allow a distributed group of people to cooperatively edit a shared document at the same time. For local responsiveness, they usually replicate the shared data to hide communication latencies. Then consistency maintenance and concurrency control become critical issues.

A special optimistic concurrency control method called operational transformation (OT) (Ellis and Gibbs, 1989; Sun and Ellis, 1998) has been well accepted in this context. Conceptually, OT always allows local operations to execute in a nonblocking manner and *transforms* remote operations before execution such that inconsistencies are repaired. In this way, local response time is not sensitive to networking latencies. As a result, it is generally believed in the literature that the use of OT algorithms automatically achieves high local responsiveness (Ellis and Gibbs, 1989; Ressel et al., 1996; Suleiman et al., 1998; Sun et al., 1998; Sun and Ellis, 1998; Li and Li, 2004).

A number of OT-based algorithms have been proposed since 1989, such as dOPT (Ellis and Gibbs, 1989), adOPTed (Ressel et al., 1996), SOCT (Suleiman et al., 1998; Vidot et al., 2000), and GOT/GOTO (Sun et al., 1998; Sun and Ellis, 1998; Shen and Sun, 2002b; Sun, 2002). State difference transformation (SDT) (Li and Li, 2004, 2005a) is a recent OT algorithm developed in our group that is proved to achieve convergence in pure peer-to-peer group editors for the first time, to the best of our knowledge. However, our early work on SDT was only intended to solve the correctness problem while leaving out performance issues. In this paper, we present a follow-up (called SDT Optimized or SDTO) that extends SDT in the following two directions: First, SDT follows the classic simplified operation set (insert and delete) and can only be used for cooperative editing of pure texts. SDTO includes a third primitive operation (update) and hence supports formatted documents. Secondly, the time complexity of SDT is $O(n^3)$ and space complexity $O(n^2)$, where n is the size of the operation log. SDTO improves the expected time complexity to $O(n^2)$ and space complexity by a large factor.

Moreover, this paper studies the performance of SDTO and argues that our findings have general implications on OT algorithms. First, we prove that the worst-case execution time of OT only happens in rare situations in which OT needs to break ties between two concurrent insert operations to address the convergence problem. Second, our experiments show that the performance (or local response time) of OT is actually dependent on a number of factors such as the ratio of operation types and the size of the operation log. The performance of concurrency control algorithms such as OT is important because interactive groupware applications such as group editors are demanded to achieve a local response time of 100 ms or less (Shneiderman, 1984; Bhola et al., 1998). The proof is also important because it justifies the optimization (for the first time to our knowledge) and improves our understanding of the causes of the convergence problem. Results in this paper inform the design of OT-based group editors in general.

The rest of this paper is organized as follows. Section 2 gives background information of group editors and the consistency model. Then Section 3 presents the SDTO algorithm and related correctness proofs. An example is described in Section 4. Section 5 analyzes the complexity of SDTO and experimental results. Section 6 compares related works. Section 7 summarizes contributions of this paper and points out future directions of research.

2. Background and models

2.1. A model of group editors

Group editors generally replicate the shared data (document) at each site. For simplicity and as a convention (Ellis and Gibbs, 1989; Sun and Ellis,

1998), we model the shared state s of the system as a string of characters (or a list, sequence, stream of objects). To distinguish different states at different sites, we annotate it like s_i^n, where i is the site id and n is the state number. The superscript or subscript is often omitted if it is not interesting. Suppose the first character of any nonempty string is indexed as zero. We assume that all state transitions in the system are caused by the following three primitive operations:

- ins(p,c): insert a character c at position p.
- del(p): delete the character at position p.
- upd(p,k,v): update attribute k of the character at position p to value v.

Obviously the position parameter p of any operation o is defined relative to some state s. Hence we often annotate o with its *definition state* s like o^s. If o^s is generated in s, then s is called its *generation state*, denoted by function gst(o). If o^s is to be executed in s, then s is called its *execution state*, denoted by function est(o). Note that est(o) is not necessarily equivalent to gst(o).

As in Ellis and Gibbs (1989), we use state vectors to timestamp states and operations. A state vector v(s) is represented by $\langle x_1, x_2, \ldots, x_n \rangle$, where n is the number of sites in the system and x_i ($i = 1, 2, \ldots, n$) is the number of operations executed at site i. We generally assume that all sites start from the same initial state. Each time a site j executes a (causally ready) operation o generated by site i, v $(s_j)[i]$, the ith element of v(s_j) is increased by one. Each time an operation o is generated at site j, it is executed immediately, which advances s_j to s_j'. Then o is propagated with timestamp v(s_j') to other sites.

Definition 1. The fact that the execution of operation o^s in state s results in state s' is denoted by exec $(s, o^s) = s'$. Function $t\,(o)$ returns the type of o, which is ins, del, or upd; p (o) returns the execution position of o relative to est (o); $c(o)$ returns the effect character to be inserted, deleted, or updated by o; function $k(o)$ returns the attribute to be updated if $t(o) = upd$; function $id(o)$ returns the id of the site at which o is generated; v (o) returns the vector timestamp of o.

Definition 2. Given two vectors, v_1 and v_2, $v_1 = v_2$ iff $\forall i : v_1[i] = v_2[i]$, and $v_1 < v_2$ iff $\forall i : v_1[i] \leq v_2[i]$ and $\exists j : v_1[j] \neq v_2[j]$. Function min (v_1, v_2) (or max (v_1, v_2)) returns $\langle x_1, x_2, \ldots, x_n \rangle$ where $\forall i : x_i = min\{v_1[i], v_2[i]\}$ (or max $\{v_1[i], v_2[i]\}$). Operation o_1 happened before o_2, or $o_1 \rightarrow o_2$, iff v (o_1). Operation o_1 is concurrent with o_2, or $o_1 \parallel o_2$, iff neither $o_1 \rightarrow o_2$ nor $o_2 \rightarrow o_1$.

Definition 3. Two operations o_1, o_2 are contextually serialized iff exec (est $(o_1), o_1$) = est (o_2). A sequence of operations $o_1, o_2, \ldots, o_n$ is contextually serialized iff for any i, $1 \leq i \leq n-1$, exec(est(o_i), o_i) = est (o_{i+1}). For brevity, we often refer to a contextually serialized sequence as a sequence.

Notation $sq[i,j]$ denotes a subsequence of sq ranging from its ith to jth element. Given state s^0 and sequence sq, where $s^0 = \text{est}(sq[0])$ and $|sq| = n$, $s^n = \text{exec}(s^0, sq) = \text{exec}(\text{exec}(s^0, sq[0]), sq[1, n-1])$. That is, executing a sequence sq in state s^0 is equivalent to executing $sq[0]$ in s^0, which results in s^1, then executing $sq[1]$ in s^1, and so forth. Given two sequences sq_1 and sq_2, if $\text{est}(sq_2[0]) = \text{exec}(\text{est}(sq_1[0]), sq_1)$, we also say that sq_1 and sq_2 are contextually serialized. For the same reason, operation $sq_1[n-1]$ and sequence sq_2, sequence sq_1 and operation $sq_2[0]$, are also contextually serialized, respectively. Notation $sq_1 + sq_2$ concatenates sq_1, sq_2 into one.

2.2. OT PRELIMINARIES

The basic idea of OT is to execute any local operation once it is generated and to transform a remote operation against previously executed concurrent operations before it is executed. Hence an operation log (or history buffer *HB*) is maintained at each site to keep track of all executed (local and remote) operations in their order of execution.

To illustrate the basic ideas of OT, consider the scenario in Figure 1. Suppose two sites start from the same initial state $s_1^0 = s_2^0 =$ "ab". Site 1 first performs $o_1 = \text{ins}(1,\text{'x'})$ to insert character 'x' before 'b', resulting in $s_1^1 =$ "axb", and then $o_2 = \text{upd}(0,u,t)$ to change the attribute named u (underlined) of character 'a' to t (true) so that it is underlined, resulting in $s_1^2 =$ "$\underline{\text{a}}$xb". Concurrently site 2 performs $o_3 = \text{del}(1)$ to delete character 'b', resulting in $s_2^1 =$ "a". When $o_3^{s_2^0}$ arrives at site 1, if it is executed as it is, then character 'x' will be deleted instead of 'b', which violates the original intention of operation o_3 (Sun et al., 1998). This is because $o_3^{s_2^0}$ is generated without knowledge of $o_1^{s_1^0}$ and $o_2^{s_1^1}$. However, its execution state s_1^2 is different from its

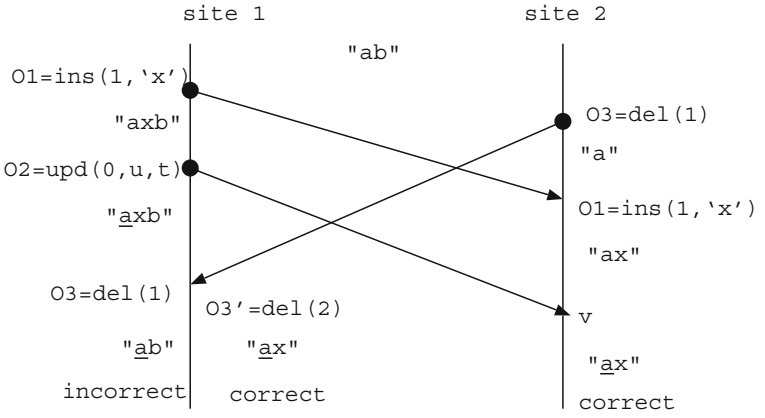


Figure 1. Transforming operation o_3 before executing it in current state of site 1.

generation state s_2^0. The intuition of OT is to shift the position parameter of $o_3^{s_2^0}$ to incorporate the effect of o_1 and o_2 such that the result $o_2^{s_1^2}$ can be correctly executed in state s_1^2. This process is called inclusion transformation (IT) in Sun et al. (1998).

Observe that the insertion of 'x' by $o_1^{s_1^0}$ shifts the position of $\mathrm{c}(o_3^{s_2^0})$ = 'b' to the right, while the execution of $o_2^{s_1^1}$ does not affect the position of $\mathrm{c}(o_3^{s_2^0})$. We should shift the position parameter of o_3 to position 2 in s_1^2. Hence we first have $o_3^{s_1^1} = \mathrm{IT}(o_3^{s_2^0}, o_1^{s_1^0}) = \mathrm{del}(2)$ and then $o_3^{s_1^2} = \mathrm{IT}(o_3^{s_1^1}, o_2^{s_1^1}) = \mathrm{del}(2)$. The execution of $o_3^{s_1^2}$ in s_1^2 leads to the correct state $s_1^2 =$ "ax",which is identical to the final state at site 2 after both o_1 and o_2 are executed. As a result, OT seems able to achieve convergence and preserve the original operation intentions despite the different orders of execution at different sites (Sun et al., 1998).

Another type of transformation seen in the literature is called *exclusion transformation* (ET) in Sun et al. (1998). In the above example, exec $(s_1^0, o_1^{s_1^0}) = s_1^1$ and $o_3^{s_1^1} = \mathrm{del}(2)$ is defined in state s_1^1. We use function $o_3^{s_1^0} = \mathrm{ET}(o_3^{s_1^1}, o_1^{s_1^0})$ to *exclude* the effect of $o_1^{s_1^0}$ from $o_3^{s_1^1}$ as if $o_1^{s_1^0}$ had not been executed in s_1^0. The result, $o_3^{s_1^0} = \mathrm{ET}(o_3^{s_1^1}, o_1^{s_1^0}) = \mathrm{del}(1)$, can be correctly executed in state s_1^0, which is not against intuition.

It has been generally accepted (Sun and Ellis, 1998) that each OT algorithm consists of two parts: *transformation functions* (IT and ET) determine how one operation is transformed against another, and the *control algorithm* determines how an operation is transformed against a given operation sequence (e.g., the history buffer). Hence the correctness of a specific OT algorithm depends on both parts. We will give our IT/ET functions and control algorithm in Section 3.

2.3. CONSISTENCY MODEL

The consistency model on which this work is based has been presented in Li and Li (2004, 2005a). It is not to be claimed as a contribution in this paper. For this paper to be self-contained, however, we will explain the main concepts.

Observe that characters in any string (or objects in any linear structure) can be uniquely identified by their positions. As a convention in group editors (Ellis and Gibbs, 1989; Sun and Ellis, 1998), we assume that every character in a string is an unique object, possibly with different attributes such as ASCII code, color, and boldness. Due to this assumption, any two appearances of the same character have different ids and no two insertions insert the same character. Note this assumption does not necessarily imply actual implementation. Given any non-empty string s and character c is the ith

element in s, we denote $s[c] = i$ and $c = s[i]$. For convenience we often use s to denote the set of characters in s.

We first define the concept of *character relation* $\prec$ as follows: Assume that two special invisible characters c_b, c_e mark the beginning and end of any string s, respectively. For any string s and two character $c_1, c_2 \in s$, we say $c_1 \prec c_2$ iff $s[c_1] < s[c_2]$. Relation $\prec$ is transitive and a total order. When a new character c is inserted into position p of s, the insertion establishes relation between c and the two existing characters $s[p-1]$ and $s[p]$ such that $s[p-1] \prec c \prec s[p]$. When a character is deleted from position p of s, however, it does not change the established relation $\prec$. Similarly, when an attribute of a character is updated, it does not change relation $\prec$, either.

Assuming that all sites in a group editing session start from the same initial state s^0, relation $\prec$ is first defined on s^0. Relation $\prec$ evolves as operations are executed in their generation and execution states. When an operation o is executed locally in gst(o), we can assume that its execution is correct and the relation $\prec$ resulted from its execution reflects the intention of o. When o is executed at a remote site, the characters in est(o) may not be the same as those in gst(o), due to concurrent execution of other operations. However, the execution of o must not violate the character order that has been established as it was executed in gst(o). That is, if o is an insertion, its execution must preserve the relation $\prec$ on all characters that appear in both est(o) and gst(o). If o is a deletion or update, it must delete or update the same character in est(o) as in gst(o). This is the so-called single-operation effects relation preservation condition defined in Li and Li (2004, 2005a).

In addition, the execution of o must not violate the partial relations that have been, respectively, established when o and its concurrent operations are executed in their generation states. This is the so-called multi-operation effects relation preservation condition defined in Li and Li (2004, 2005a). To achieve so, we introduce two tie-breaking policies: First, when two insertions o_1 and o_2 concurrently insert between the same two characters c_x and c_y in their generation states, i.e., $c_x \prec c(o_1) \prec c_y$ and $c_x \prec c(o_2) \prec c_y$, we enforce that the one with a smaller site id precedes the other. That is, if id $(o_1) \prec$ id(o_2), the resulted relation will be $c_x \prec c(o_1) \prec c(o_2) \prec c_y$. Secondly, when an operation o_1 inserts a character in gst(o_1) at the same position as a character that was deleted by o_2, we enforce that the inserted character precedes the deleted character in relation $\prec$. That is, $c_x \prec c(o_1) \prec c(o_2) \prec c_y$.

Due to these two policies, when any operation o is to be executed in est(o), all relations that have been established by o in gst(o) and operations executed so far at this site can be merged as a total order. More generally, if we use C_t to denote the set of characters that ever appear in a group editing session, relation $\prec$ defined over this set is a total order.

The *operation effects relation* can be considered a subset of the above explained character relation, with the part involving characters in the initial

state removed. Hence we use the same notation $\prec$ for the operation effects relation. Note there are cases in which the effect characters of two given operations o_1 and o_2 are equivalent or $c(o_1) \equiv c(o_2)$, e.g., o_1 and o_2 concurrently delete (or update) the same character, or o_1 deletes (or updates) the same character inserted by o_2. By combining the two conditions of single-operation and multi-operation effects relation preservation (Li and Li 2004, 2005a) into one, operation effects relation preservation, we have the following rephrased correctness criteria.

Definition 4. A group editor is correct if it always maintains the following two properties:

(1) Causality preservation: For any two operations o_1 and o_2, if $o_1 \rightarrow o_2$, then o_1 is executed before o_2 at all sites.
(2) Operation effects relation preservation: The execution of any operation in any state preserves the operation effects relation $\prec$.

It is shown in Li and Li (2004, 2005a) that the preservation of operation effects relation implies convergence. Hence the above-defined correctness criteria no longer need to include convergence as do previous models (Ellis and Gibbs, 1989; Ressel et al., 1996; Suleiman et al., 1998; Sun et al., 1998). The relation $\prec$ is mostly determined as operations are generated and executed in their generation states. The only "artificial" part is the two policies that are used to break ties, as explained above. Policies as such are widely used in distributed computing systems and OT algorithms, including dOPT (Ellis and Gibbs, 1989), SOCT2/3/4 (Suleiman et al., 1998; Vidot et al., 2000), and GOT/GOTO (Sun and Ellis, 1998; Sun et al., 1998). Compared to the intention preservation condition in previous work (Suleiman et al., 1998; Sun and Ellis, 1998; Sun et al., 1998; Vidot et al., 2000), our notion of operation effects relation preservation is easier to formalize and verify.

To determine the relation between any two given characters, if we know whether or not there exists at least one character between them, the problem becomes more straight-forward. The following concept of landmark character will be used in Section 3. Note, however, that we will never really need to compute the set of landmark characters. It is the existence of this set that matters.

Definition 5 [Landmark Character]. For any three characters $c_1, c_2, c_3 \in C_t$, we say that c_3 is a landmark between c_1 and c_2, iff either $c_1 \prec c_3 \prec c_2$ or $c_2 \prec c_3 \prec c_1$. We denote the set of landmark characters between c_1 and c_2 as $C_{ld}(c_1, c_2) = \{c_3 \in C_t | c_1 \prec c_3 \prec c_2 \vee c_2 \prec c_3 \prec c_1\}$.

3. The SDTO Algorithm

We maintain two data structures at each site. The first is the operation log (or history buffer *HB*), which records executed local and remote operation in their order of execution. The second data structure is the effects relation (*ER*). Before executing any remote operation o, we need to compute its position parameter relative to the current state. To achieve so, we must derive the effects relation $\prec$ between $\mathrm{c}(o)$ and those of the executed concurrent operations stored in *HB*. In some cases, however, it is expensive to compute the effects relation between a given pair of operations. Hence we compute $\prec$ only when it is necessary and save the result to *ER* for future uses.

We use *operation keys* to uniquely identify operations. The key of any operation o, denoted by $\mathrm{key}(o)$, is a pair of its generation site id and its sequence number, i.e., $\langle \mathrm{id}(o), \mathrm{v}(o)[\mathrm{id}(o)]\rangle$. Due to the established state vector maintenance protocol summarized in Section 2, no two different operations share the same key. Unlike $\mathrm{p}(o)$, the value of $\mathrm{key}(o)$ does not change. Since each operation has a unique effect character, $\mathrm{c}(o)$ can often be safely replaced by $\mathrm{key}(o)$. The relation *ER* is implemented as a hash table, which stores pairs of operations: we infer $\mathrm{c}(o_x) \prec \mathrm{c}(o_y)$ if pair $\langle \mathrm{key}(o_x), \mathrm{key}(o_y)\rangle$ is found in *ER*, or equivalently, $\langle \mathrm{c}(o_x), \mathrm{c}(o_y)\rangle \in ER$.

In the rest of this section, we first define IT/ET functions and their correctness conditions based on relation $\prec$ in Sections 3.1 and 3.2, respectively. Then Section 3.3 presents the control algorithm, which ensures the conditions of IT/ET are satisfied. In particular, we identify the worst cases in which relation $\prec$ must be computed. Sections 3.4 and 3.5 explain some in-depth algorithm details. Section 3.6 sketches a correctness proof.

3.1. INCLUSION TRANSFORMATION

The purpose of IT (o_1^s, o_2^s) is to incorporate the effect of an executed operation o_2 into o_1 such that $o_1^{s'}$ can be correctly executed in state s', where

Function 1 IT (o_1^s, o_2^s): $o_1^{s'}$, *where* $s' = \mathrm{exec}(s, o_2^s)$.

```
1  if(p(o2) = p(o1) ∧ t(o2) = t(o1) = del
2    ∨ p(o2) = p(o1) ∧ t(o2) = del ∧ t(o1) = upd
3    ∨ p(o2) = p(o1)∧ t(o2) = t(o1) = upd ∧ k(o2) = k(o1) ∧ id(o1) < id(o2))
4   return o1' ← φ;
5 else o1' ← o1;
6   if(p(o2) < p(o1))
7     ∨ p(o2) = p(o1)∧ t(o2) = ins ∧ (t(o1) = del ∨ t(o1) = upd)
8     ∨p(o2) = p(o1) ∧ t(o2) = t(o1) = ins ∧ ⟨c(o2), c(o1)⟩ ∈ ER)
9     if t(o2) = ins
10          p(o1') ← p(o1) + 1;
11      else if t(o2) = del
12          p(o1') ← p(o1)−1;
13   return o1';
```

$s' = \text{exec}(s, o_2^s)$. As has been established in Sun and Ellis (1998) and Sun et al. (1998), the precondition of IT is that the two operations o_1 and o_2 are defined relative to the same state, or $s = \text{est}(o_1) = \text{est}(o_2)$. This is because it may not make sense to compare their position parameters if the two operations are defined relative to different states. However, condition $\text{est}(o_1) = \text{est}(o_2)$ alone is insufficient for determining the relation of $\text{c}(o_1)$ and $\text{c}(o_2)$. There are well-known scenarios in which this condition is satisfied but the relation cannot be correctly determined (Li and Li, 2004).

We illustrate this with a well-known counterexample that first appeared in Suleiman et al. (1998) and Sun et al. (1998). Suppose three sites start from the same initial state "abc". Three concurrent operations are generated at the three sites, respectively: $o_1 = \text{ins}(2,\text{'y'})$, $o_2 = \text{del}(1)$, and $o_3 = \text{ins}(1,\text{'x'})$. The landmark character between 'x' and 'y' is 'b'. Hence the only correct final state must be "axyc" after all three operations are executed. In early work (e.g., Ressel et al., 1996), however, the two insertions o_1 and o_3 often cannot be correctly transformed after they have both transformed with the deletion o_2. More specifically, $o_1' = \text{IT}(o_1, o_2) = \text{ins}(1, \text{'x'})$ and $o_3' = \text{IT}(o_3, o_2) = \text{ins}(1,\text{'y'})$. When o_1' is inclusively transformed against o_3', if we directly compare their site ids to break the tie, since the site id of o_1 is smaller than that of o_3, we will get $o_1'' = \text{IT}(o_1', o_3') = \text{ins}(1, \text{'y'})$. Consequently the wrong final state "ayxc" is obtained. The problem exists in early works because their IT functions cannot correctly break the tie when the landmark character 'b' is deleted by o_2.

Function 1 is divided into two distinct parts: First, lines 1–4 define policies for *resolving conflicts*. Two concurrent operations conflict if (1) they both intend to delete the same character, (2) one deletes and another updates the same character, or (3) both updates the same attribute of the same character. These three cases are defined in line 1, 2 and 3, respectively. Let ϕ denote the *identity operation*. As will be explained in Section 3.3, the identity operation is not saved in *HB* and not executed.

To address case (1), we ensure that the character is deleted only once by turning the late operation o_1 into ϕ. In case (2), if o_1 is to update the character already deleted by o_2, we also return ϕ. Note that if o_1 is to delete the character o_2 updates, we return o_1 since the character will be deleted. In case (3), if the same attribute of the same object is updated by a number of concurrent operations, we mandate that the one with the greatest site id eventually wins. Hence if $\text{id}(o_1) < \text{id}(o_2)$, there is no need to execute o_1 and thus ϕ is returned.

Second, lines 5–13 define the cases in which $\text{p}(o_1)$ may be shifted. In particular lines 6–8 define conditions for determining $\text{c}(o_2) \prec \text{c}(o_1)$. If $\text{c}(o_2) \prec \text{c}(o_1)$ and $\text{t}(o_2) = \text{ins}$ or del, the execution of o_2 invalidates $\text{p}(o_1)$. Hence $\text{p}(o_1)$ should be increased by one if $\text{t}(o_2) = \text{ins}$, and decreased by one if $\text{t}(o_2) = \text{del}$. However, if $\text{c}(o_1) \prec \text{c}(o_2)$, meaning $\text{c}(o_2)$ is on the right side of

$p(o_1)$, or if $c(o_2) \prec c(o_1)$ but $t(o_2) = upd$, meaning that a character on the left of $p(o_1)$ is updated, the execution of o_2 does not affect $p(o_1)$ and thus no shifting of $p(o_1)$ is necessary. To verify $c(o_2) \prec c(o_1)$, we consider the following three conditions: (1) $p(o_2) < p(o_1)$; (2) $p(o_2) = p(o_1)$, $t(o_2) = ins$, and $t(o_1) = del$ or upd; and (3) $p(o_2) = p(o_1)$, $t(o_2) = t(o_1) = ins$, and relation $c(o_2) \prec c(o_1)$ is already known, i.e., it is computed previously and saved in *ER*. In the following lemmas, we establish that the effects relation of o_1 and o_2 can be correctly determined as described above.

Lemma 1. *Function 1 can correctly determine the relation of $c(o_1)$ and $c(o_2)$ if (1) $s = est(o_1) = est(o_2)$, and (2) at least one of o_1 and o_2 is del or upd.*

Proof. First consider the case where $t(o_1) = del/upd$ and $t(o_2) = del/upd$. Since both o_1 and o_2 target characters that are already in s, the relation between $c(o_1)$ and $c(o_2)$ can be correctly determined by comparing $p(o_1) = s[c(o_1)]$ and $p(o_2) = s[c(o_2)]$. If $p(o_1) < p(o_2)$, we have $c(o_1) \prec c(o_2)$; if $p(o_1) > p(o_2)$, we have $c(o_2) \prec c(o_1)$; and if $p(o_1) = p(o_2)$, we have $c(o_1) \equiv c(o_2)$.

Then consider the case where $t(o_1) = ins$ and $t(o_2) = del/upd$. By operation semantics, o_1 is to insert a new character $c(o_1)$ into s and o_2 is to delete or update an existing character $c(o_2)$ in s. It is impossible to have $c(o_1) \equiv c(o_2)$. Hence the relation of $p(o_1)$ and $p(o_2) = s[c(o_2)]$ determines the relation between $c(o_1)$ and $c(o_2)$ in s'. That is, if $p(o_1) \leq p(o_2)$, we have $c(o_1) \prec c(o_2)$. Otherwise, $c(o_2) \prec c(o_1)$. The case of $t(o_1) = del/upd$ and $t(o_2) = ins$ is symmetric. □

Lemma 2. *Function 1 can correctly determine the relation between $c(o_1)$ and $c(o_2)$ if (1) $s = est(o_1) = est(o_2)$, (2) $t(o_1) = t(o_2) = ins$, and additionally (3) relation $\prec$ is already available in ER if $p(o_1) = p(o_2)$.*

Proof. If $p(o_1) < p(o_2)$, there must exist at least one character c between $p(o_1)$ and $p(o_2)$ in s. By the semantics of insertion, $p(o_1) \leq s[c] < p(o_2)$. That is, $c(o_1) \prec c \prec c(o_2)$, or $c(o_1) \prec c(o_2)$. The case of $p(o_1) > p(o_2)$ is symmetric. If $p(o_1) = p(o_2)$ in state s, we cannot naively break the tie by comparing their site ids. Due to well-known counterexamples discussed earlier in this subsection, this tie could be caused by the deletion of the set of landmark characters between $c(o_1)$ and $c(o_2)$. We assume their correct relation has been computed previously (by the control algorithm) and is available in the *ER* table. □

The conditions of Lemmas 1 and 2 completely cover all the possible operation type combinations. They can be further merged as shown in Corollary 1. The conditions are *sufficient* simply because, once they are satisfied, IT(o_1, o_2) can always correctly determine the effects relation between o_1 and o_2. It is clear that we will need the effects relation $\prec$ only when two insertions tie.

Corollary 1. *IT* (o_1^s, o_2^s) *is correct if (1)* $s = est(o_1) = est(o_2)$, *and additionally (2) relation* $\prec$ *is already available in ER if* $t(o_1) = t(o_2) = ins$ *and* $p(o_1) = p(o_2)$.

3.2. EXCLUSION TRANSFORMATION

The purpose of ET $(o_1^{s'}, o_2^s)$ is to exclude the effect of o_2^s from $o_1^{s'}$ as if o_2^s had not been executed. Note that the relation between o_1 and o_2 could be either $o_2 \rightarrow o_1$ or $o_2 \parallel o_1$. Due to causality preservation, the case of $o_1 \rightarrow o_2$ will not appear. As established in Sun and Ellis (1998) and Sun et al. (1998), the precondition of ET is $s' = \text{exec}(s, o_2)$, where $s = \text{est}(o_2)$ and $s' = \text{est}(o_1)$. However, condition $s' = \text{exec}(s, o_2)$ alone is insufficient for determining the relation between c(o_2) and c(o_1) (Li and Li, 2004).

We summarize related discussions in Li and Li (2004) as follows. In ET(o_1, o_2), suppose $s =$ "abc", $o_2^s = \text{del}(1)$, $s' = \text{exec}(s, o_2^s) =$ "ac", $o_1^{s'} = \text{ins}(1, \text{'x'})$, and $s'' = \text{exec}(s', o_1^{s'}) =$ "axc". There are three possible cases: (1) $o_2 \rightarrow o_1$, (2) $o_2 \parallel o_1$ and $o_1^s = \text{ins}(1, \text{'x'})$, and (3) $o_2 \parallel o_1$ and $o_1^s = \text{ins}(2, \text{'x'})$. In case (1), 'x' is inserted at the same position where 'b' was deleted. The relation between these two characters is not inherent and hence either $o_1^s = \text{ins}(1,\text{'x'})$ or ins(2,'x') is acceptable. However, a tie-breaking policy must be mandated to get a deterministic result of o_1^s. In case (2) and (3), the relation between 'b' and 'x' is inherent when o_1 and o_2 are generated. However, we cannot correctly distinguish between them unless the relation is already known (i.e., saved in the *ER* table). In Lemma 6, we will study how to cope with these three cases.

As shown in Function 2 lines 1–2, if o_1 is to delete or update the character inserted by o_2, or $o_2 \rightarrow o_1$ and c $(o_1) \equiv$c(o_2), due to causality, it does not make sense to pretend that o_2 had not been executed. In this case we say that o_1 depends on o_2 and a dependency error is raised. The control algorithm will prevent this case from happening.

When processing ET(o_1, o_2), if c$(o_1) \prec$ c(o_2), meaning that c(o_2) appears on the right side of c(o_1), then excluding the effect of o_2 does not affect p(o_1). Hence o_1 should be returned as it is. If c$(o_2) \prec$ c(o_1), the effect of o_1 should be shifted by one position to the left had o_2 not inserted a character on its left; or the position parameter of o_1 should be increased by one had o_2 not deleted a character on its left. We get c$(o_2) \prec$ c(o_1) if the conditions in lines 4–7 hold.

Function 2 $ET(o_1^{s'}, o_2^s) : o_1^s$, *where* $s' = \text{exec}(s, o_2^s)$.

```
1 if(p(o2) = p(o1) ∧ t(o2) = ins ∧ (t(o1) = del ∨ t(o1) = upd))
2     return dependency error;
3 else o1' ← o1;
4     if p(o2) < p(o1)
5         ∨ p(o2) = p(o1) ∧ t(o2) = t(o1) = del
6         ∨ p(o2) = p(o1) ∧ t(o2) = del ∧ t(o1) = upd
7         ∨ p(o2) = p(o1) ∧ t(o2) = del ∧ t(o1) = ins ∧ ⟨c(o2), c(o1)⟩ ∈ ER
8         if t(o2) = ins
9          p(o1')]] > ← p(o1)-1;
10         else if t(o2) = del
11           p(o1') ← p(o1) + 1;
12     return o1';
```

In the following lemmas, we establish the sufficient conditions under which Function 2 can correctly determine the effects relation of o_1 and o_2.

Lemma 3. *Function 2 can correctly determine the relation of* $c(o_2)$ *and* $c(o_1)$ *if (1)* $t(o_2) = ins/upd$ *and* $t(o_1) = ins$, *and (2)* $s' = exec(s, o_2)$.

Proof. From the given conditions, $c(o_2)$ must be already in s'. Since o_1 is an insertion, it is impossible to have $c(o_2) \equiv c(o_1)$. If o_1 inserts on the right of $c(o_2)$ in s', i.e., $p(o_1) > p(o_2)$, it must be that $c(o_2) \prec c(o_1)$. If o_1 inserts on the left of $c(o_2)$ in s', or $p(o_1) \leq p(o_2)$, it must be that $c(o_1) \prec c(o_2)$. □

Lemma 4. *Function 2 can correctly determine the relation of* $c(o_2)$ *and* $c(o_1)$ *if (1)* $t(o_2) = ins$ *and* $t(o_1) = del/upd$, *and (2)* $s' = exec(s, o_2)$.

Proof. From the given conditions, $c(o_2)$ must appear in s' and $p(o_2) = s'[c(o_2)]$. Since o_1 is to delete/update a character in s', it must be that $p(o_1) = s'[c(o_1)]$. Therefore, if $p(o_2) = p(o_1)$, it must be that $c(o_2) \equiv c(o_1)$; if $p(o_2)$, it must be that $c(o_2) \prec c(o_1)$; if $p(o_1) < p(o_2)$, we have $c(o_1) \prec c(o_2)$. □

Lemma 5. *Function 2 can correctly determine the relation of* $c(o_2)$ *and* $c(o_1)$ *if (1)* $t(o_2)$ *and* $t(o_1)$ *are either del or upd, and (2)* $s' = exec(s, o_2)$.

Proof. First consider the case of $t(o_2) = t(o_1) = upd$. Then $c(o_2)$, $c(o_1)$ must appear in both s and s'. If $p(o_2) < p(o_1)$, we have $c(o_2) \prec c(o_1)$; and if $p(o_2) > p(o_1)$, we have $c(o_1) \prec c(o_2)$; If $p(o_2) = p(o_1)$ and $k(o_2) \neq k(o_1)$,

excluding the effect of o_2 does not affect o_1. In the case of $\mathrm{p}(o_2) = \mathrm{p}(o_1)$ and $\mathrm{k}(o_2) = \mathrm{k}(o_1)$, if $o_2 \rightarrow o_1$, this is not considered a conflict and o_1 is returned as if o_2 had not been executed; if $o_2 \parallel o_1$, then o_1 should have been inclusively transformed against o_2, which should have resulted in $o_1 = \phi$. According to our control algorithm, ϕ is not added into *HB* and $\mathrm{ET}(\phi, o_2)$ will never happen.

Next consider the case of $\mathrm{t}(o_2) = \mathrm{upd}$ and $\mathrm{t}(o_1) = \mathrm{del}$. Then $\mathrm{c}(o_2)$ and $\mathrm{c}(o_1)$ also must appear in both s and s'. If $\mathrm{p}(o_2) \neq \mathrm{p}(o_1)$, we have either $\mathrm{c}(o_2) \prec \mathrm{c}(o_1)$ or $\mathrm{c}(o_1) \prec \mathrm{c}(o_2)$. If $\mathrm{p}(o_2) = \mathrm{p}(o_1)$, excluding the effect of o_2 does not affect o_1.

Lastly consider the case of $\mathrm{t}(o_2) = \mathrm{del}$ and $\mathrm{t}(o_1) = \mathrm{del/upd}$. It is impossible to have $\mathrm{c}\ (o_2) \equiv \mathrm{c}(o_1)$, because two contextually serialized operations cannot delete the same character, and one cannot update the character already deleted by another. Besides, $\mathrm{c}(o_1)$ must appear in s and s'. If $\mathrm{p}(o_2) \leq \mathrm{p}(o_1)$, we have $\mathrm{c}(o_2) \prec \mathrm{c}(o_1)$; and if $\mathrm{p}(o_2) > \mathrm{p}(o_1)$, we have $\mathrm{c}(o_1) \prec \mathrm{c}(o_2)$. □

Lemma 6. *Function 2 can correctly determine the relation of* $c(o_2)$ *and* $c(o_1)$ *if (1)* $t(o_2) = del$ *and* $t(o_1) = ins$, *(2)* $s' = exec(s, o_2)$, *and additionally (3) relation between* $c(o_2)$ *and* $c(o_1)$ *is saved in ER or computable if* $p(o_2) = p(o_1)$.

Proof. Since o_1 inserts a new character after o_2 deletes an existing one from s, it is impossible to have $\mathrm{c}\ (o_2) \equiv \mathrm{c}(o_1)$. If $\mathrm{p}(o_2) < \mathrm{p}(o_1)$, there must be at least one character between positions $\mathrm{p}(o_2)$ and $\mathrm{p}(o_1)$ in s'. Hence we have $\mathrm{c}(o_2) \prec \mathrm{c}(o_1)$. Similarly if $\mathrm{p}(o_2) > \mathrm{p}(o_1)$, we have $\mathrm{c}(o_1) \prec \mathrm{c}(o_2)$.

For condition $\mathrm{p}(o_2) = \mathrm{p}(o_1)$ we consider the following two cases. First, suppose $o_2 \parallel o_1$. Since they are contextually serialized, o_1 must have been inclusively transformed with o_2. Hence if their effects relation is saved in *ER*, there will be no ambiguity. Second, suppose $o_2 \rightarrow o_1$ and we can somehow ensure $C_{ld}(\mathrm{c}(o_1), \mathrm{c}(o_2)) \subseteq s'$, meaning that the landmark characters between $\mathrm{c}(o_1)$ and $\mathrm{c}(o_2)$, if any, must be present in state s'. Since $\mathrm{p}(o_1) = \mathrm{p}(o_2)$ and $\mathrm{c}(o_1)$ is not in s' yet, it must be that $C_{ld}(\mathrm{c}(o_1),\mathrm{c}(o_2)) = \emptyset$. In this case we mandate $\mathrm{c}(o_1) \prec \mathrm{c}(o_2)$ by the tie-breaking policy used in the definition of relation $\prec$ (Section 2.3). □

The above four lemmas cover all the possible type combinations of o_1 and o_2 in $\mathrm{ET}(o_1, o_2)$. We merge them in the following corollary. Note the conditions are *sufficient* because, once they are satisfied, $\mathrm{ET}(o_1, o_2)$ can always correctly determine the effects relation between o_1 and o_2. It is clear that the relation $\prec$ is required only when o_2 is deletion, o_1 is insertion, their positions tie, and $o_2 \parallel o_1$. Under the same conditions but with $o_2 \rightarrow o_1$, the control algorithm will ensure that the landmark characters between $\mathrm{c}(o_1)$ and $\mathrm{c}(o_2)$, if

any, are all present in s'. In this case, our tie-breaking policy ensures $c(o_1) \prec c(o_2)$. Correspondingly in Function 2, $\langle c(o_2), c(o_2)\rangle \notin ER$ and hence $o_1^s = o_1^{s'}$.

Corollary 2. *$ET(o_1^{s'}, o_2^s)$ is correct if (1) $s' = exec(s, o_2)$, and additionally (2) if $t(o_2) = del$, $t(o_1) = ins$, and $p(o_2) = p(o_1)$, relation between $c(o_2)$ and $c(o_1)$ is saved in ER where $o_2 \parallel o_1$, or $C_{ld}(c(o_1), c(o_2)) \subseteq s'$ where $o_2 \rightarrow o_1$.*

3.3. THE CONTROL ALGORITHM

We consider a group editor of N sites that start from the same initial state. For simplicity, we assume a reliable network environment without network partition or node failure. Figure 2 shows the top-level control algorithm at site i. Let s be the current state of site i. Each site maintains a state vector sv, which is zeroed initially, an operation log HB, and an operation effects relation table ER, which are empty initially. Assume every operation o has three attributes: $o.id$ for its generation site id, $o.key$ for the operation key, and $o.v$ for its vector timestamp. The use of state vector has been established (Ellis and Gibbs, 1989; Sun et al., 1998) and explained in Section 1. The concepts

```
Initialize:
1   HB←[]; ER←[]; RQ←[];
2   sv←< 0, 0, ..., 0 >;

Invoke (generate) a local operation o:
1   s ← exec(s, o);
2   sv[i] ← sv[i] + 1;
3   o.id ← i; o.key ← < i, sv[i] >; o.v ← sv;
4   HB ← HB + o;
5   broadcast o to other sites;

Receive o from network:
1   RQ ← RQ + o;

Invoke a remote operation:
1   if ∃RQ[k] that is causally-ready
2       o ← RQ[k]; remove RQ[k] from RQ;
3       o' ← Integrate(o, HB);
4       if( o' ≠ φ )
5           s ← exec(s, o');
6           HB ← HB + o';
7       sv[j] ← sv[j]+1 where j = o.id;
```

Figure 2. The control algorithm at site i.

of operation log, effects relation, and operation key have been explained in the beginning of Section 3.

As a convention in group editors (Ellis and Gibbs, 1989; Ressel et al., 1996; Suleiman et al., 1998; Sun et al., 1998), as soon as a local operation o is generated, it is executed and appended to HB and then multicast to other cooperating sites. Each site maintains a queue RQ to store remote operations received from other sites. To ensure causality preservation, a remote operation o is invoked only when it is causally ready. Function Integrate(o, HB) is called to compute its execution form o' relative to the current state s. If o' is an identity operation ϕ, it is discarded. Otherwise o' is executed and appended to HB. A remote operation o is causally ready in state s, iff all operations that happened before o have been executed in s. According to Ellis and Gibbs (1989) and Sun et al. (1998), given a remote operation o generated at site j, o is causally ready (at site i), iff (1) $o.v[j] = sv[j] + 1$ and (2) for $\forall k \neq j : o.v[k] \leq sv[k]$.

Function Integrate(o, HB) first calls function TransposeR2L(o, HB) to transpose HB into two contextually serialized subsequences, L_{prec} and L_{par}, such that all operations in L_{prec} happened before o, and all operations in L_{par} are concurrent with o. The algorithm of TransposeR2L() will be explained in Section 3.4. By causality preservation, o and $L_{par}[0]$ are contextually equivalent, i.e., they are defined in the same state. Then in the loop of lines 3–14, it inclusively transforms o against operations in L_{par} one by one. Hence at each step IT($o, L_{par}[i]$), where $o \parallel L_{par}[i]$ and $0 \leq i < |L_{par}|$, the two operations are also contextually equivalent. Due to Corollaries 1 and 2, however, we need to first derive the effects relation of o and $L_{par}[i]$ under the following two conditions: (1) $\mathrm{t}(L_{par}[i]) = \mathrm{del} \wedge \mathrm{t}(o) = \mathrm{ins}$, and (2) $\mathrm{t}(L_{par}[i]) = \mathrm{t}(o) = \mathrm{ins}$.

Function 3 *Integrate* (o,HB): o', where o is causally ready.

```
1  o' ← o;
2  L_prec + L_par ← TransposeR2L(o,HB);
3  for (i=0; i<|L_par|; i++)
4     o_x ← o'; o_y ← L_par[i];
5    if(t(o_x)=ins ∧ t(o_y)=del ∧ p(o_y)<p(o_x))
6       record relation c(o_y) ≺ c(o_x) in ER;
7    else if(t(o_x)=t(o_y)=ins ∧ p(o_y)=p(o_x))
8       L ← L_prec + L_par[0, i − 1];
9       v_min ← min(v(o_x), v(o_y));
10      β_x ← Compute β(o_x,v_min,L);
11      β_y ← Compute β(o_y,v_min,L);
12      derive and record relation of c(o_x) and c(o_y) in ER;
13    o' ← IT( o', L_par[i]);
14    if(o'=φ) break;
15 return o';
```

The case of condition (1) is handled as follows. By Lemma 1, we can correctly determine the relation of an insertion and a deletion that are defined relative to the same state by simply comparing their position parameters. Hence if p $(L_{par}[i])$, we infer $\mathrm{c}(L_{par}[i]) \prec \mathrm{c}(o)$ and record it in *ER*. Note that the opposite case $\mathrm{c}(o) \prec \mathrm{c}(L_{par}[i])$ is implied in Function 2 (ET) if pair $\langle c(L_{par}[i]), c(o)\rangle$ is not found in the *ER* table. In fact, due to Corollary 2, this relation is only necessary when exclusively transforming an insertion o_x against a concurrent deletion o_y and their positions tie. It can be shown that $\mathrm{ET}(o_x, o_y)$ happens only when o_y is executed earlier than o_x. Hence we do not need to consider the case in which $\mathrm{t}(L_{par}[i]) = \mathrm{ins}$ and $\mathrm{t}(o) = \mathrm{del}$.

In the case of condition (2), if the two concurrent operations are both insertions and their positions tie, by Corollary 1, we must derive their effects relation before calling $\mathrm{IT}(o, L_{par}[i])$. This is achieved in lines 8–12 of Function 3. Let $o_x = o, o_y = L_{par}[i]$, and $L = L_{prec} + L_{par}[0,i-1]$. We compute β_x and β_y, the positions of o_x and o_y relative to their last synchronization point (LSP), and then use β_x and β_y to determine the effects relation of o_x and o_y.

The concept of LSP was first introduced in Li and Li (2004). The main idea is as follows. Conceptually, the LSP of two concurrent operations o_x and o_y is the closest previous state prior to the generation states of both o_x and o_y. In line 9, we first compute state vector $v_{min} = \min(\mathrm{v}(o_x), \mathrm{v}(o_y))$, every element of which is the minimum of the corresponding elements in the vector time-stamps of o_x and o_y. Then their LSP is state s^{lsp}, where $\mathrm{v}(s^{lsp}) = v_{min}$. Suppose L is further transposed into two contextually serialized subsequences, L_{lsp} and L_{ex}, such that L_{lsp} contains all the operations that happened before both o_x and o_y (or bear timestamps less than v_{min}). Then, by excluding the effects of all operations in L_{ex} from o_x (or o_y), we obtain the position of o_x (or o_y) relative to s^{lsp}, that is, β_x (or β_y). The algorithm for computing β values will be explained later in Section 3.5.

When $\mathrm{t}(o_x) = \mathrm{t}(o_y) = \mathrm{ins}$ and $\mathrm{p}(o_x) = \mathrm{p}(o_y)$, we determine their relation by the following rule: $\mathrm{c}(o_x) \prec \mathrm{c}(o_y)$ if and only if (1) $\beta_x \prec \beta_y$, or (2) $\beta_x = \beta_y$ and $\mathrm{id}(o_x) < \mathrm{id}(o_y)$. The resulting effects relation, $\mathrm{c}(o_x) \prec \mathrm{c}(o_y)$ or $\mathrm{c}(o_y) \prec \mathrm{c}(o_x)$, is saved in the *ER* table for future use in IT functions. Note that we do not consider it a tie until $\mathrm{p}(o_x) = \mathrm{p}(o_y)$ and $\beta_x = \beta_y$. This is different from previous work (e.g., Ellis and Gibbs, 1989; Ressel et al., 1996) in which site ids are compared directly when $\mathrm{p}(o_x) = \mathrm{p}(o_y)$.

3.4. TRANSPOSING OPERATING LOG

Function TransposeR2L() first appeared in Suleiman et al. (1998) and Sun and Ellis (1998). We include it here with a different formalization. As defined in Function 4, TransposeR2L(o,L) transposes a sequence L (or *HB*) into L_{prec} and L_{par}, where all operations in L_{prec} happened before o, all operations in

Function 4 TransposeR2L(o,L):$L_{prec} + L_{par}$

```
1    L_prec ← []; L_par ← []; //empty sequence
2    for (i=0; i<|L|; i++)
3      if(L[i] → o)
4        L_prec ← L_prec + L[i];
5      else //L[i] ‖ o
6        break;
7    if i < |L|
8      L_par ← L_par + L[i];
9      for (j=i+1; j<|L|; j++)
10       if(L[j] || o)
11         L_par ← L_par + L[j];
12       else //L[j] → o
13         o_j ← L[j];
14         for ( k = |L_par| − 1; k ≥ 0; k - -)
15           o_j ← ET(o_j, L_par[k]);
16           L_par[k] ← IT(L_par[k],o_j);
17         L_prec ← L_prec + o_j;
18 return L_prec + L_par;
```

L_{par} are concurrent with o, and L_{prec} is contextually serialized before L_{par}. Lines 2–6 scans L from left to right and appends any operation that happened before o to L_{prec} until the first operation $L[i] \parallel o$, where $0 \leq i < |L|$. Then $L[i]$ is added to L_{par} (line 8). After that, it loops (lines 9–17) to add concurrent operations to L_{par} until some $o_j = L[j]$ such that $o_j \rightarrow o$. At this point L_{prec}, L_{par}, and o_j are contextually serialized. We transpose o_j with every $L_{par}[k]$ such that o_j is contextually serialized before L_{par}, and then append o_j to L_{prec} (line 17). The transposition is achieved by the loop in lines 14–16. For every $L_{par}[k], L_{par}[k]$ and o_j are contextually serialized. To exchange their positions in the sequence, we must exclude the effect of $L_{par}[k]$ from o_j (line 15) such that they are defined in the same state, and then include the effect of o_j into $L_{par}[k]$ such that o_j and $L_{par}[k]$ are contextually serialized (line 16).

Lemma 7. *IT/ET in TransposeR2L( ) are correct.*

Proof. For any k, it is impossible to have $L[j] \rightarrow L_{par}[k]$ due to causality preservation. Because $L[j] \rightarrow o$ and $L_{par}[k] \parallel o$, the relation between $L[j]$ and $L_{par}[k]$ must be $L[j] \parallel L_{par}[k]$. Indeed, if we had $L_{par}[k] \rightarrow L[j]$, we would have $L_{par}[k] \rightarrow o$ due to transitivity of relation $\rightarrow$, which contradicts $L_{par}[k] \parallel o$. Then, since $L[j] \parallel L_{par}[k]$, we must have correctly performed IT($L[j], L_{par}[k]$) earlier. The effects relation that ensures the correctness of IT and ET has been computed and stored in *ER* by Function Integrate( ). □

3.5. COMPUTING β VALUES

Now we continue the discussions in Section 3.3 to explain how to compute β values in function Integrate( ). Conceptually suppose $L = L_{prec} + L_{par}[0, i-1]$ is transposed into $L_{lsp} + L_{ex}$. Since o_x and o_y are both defined relative to the state in which L (or $L_{lsp} + L_{ex}$) have been executed, the problem is reduced to excluding the effects of operations in sequence L_{ex} from a given operation o, which is either o_x or o_y.

Some operations in L_{ex} may be concurrent with o while the others happened before o. For any $L_{ex}[i] \parallel o$, the correctness of $\mathrm{ET}(o, L_{ex}[i])$ is ensured due to the effects relation saved in *ER* by function Integrate( ). However, for any $L_{ex}[j] \rightarrow o$, we may not be able to perform $\mathrm{ET}(o, L_{ex}[j])$ directly due to the discussions in Section 3.2. First, we must handle the cases where o depends on $L_{ex}[j]$. This is not a problem here because $\mathrm{t}(o) = \mathrm{ins}$. Second, when $\mathrm{t}(o) = \mathrm{ins}$, $\mathrm{t}(L_{ex}[j]) = \mathrm{del}$, and $\mathrm{p}(o) = \mathrm{p}(L_{ex}[j])$, we must ensure that all the landmark characters between $\mathrm{c}(o)$ and $\mathrm{c}(L_{ex}[j])$, if any, are present in the execution state of o, or more formally, $C_{ld}(c(o), c(L_{ex}[j])) \subseteq est(o)$. This is achieved by building a special ET-Safe Operation Sequence (ETSOS) from L, as follows.

Lemma 8. *Suppose* o_2 *and* o_1 *are defined in* s *and* s', respectively, and $s' = exec(s, o_2)$. $ET(o_1, o_2)$ *returns* o_1 as it is if $t(o_2) = upd$.

Proof. If $\mathrm{t}(o_1) = \mathrm{ins/del}$, $\mathrm{p}(o_1)$ will not be affected if the effect of an update operation o_2 is excluded. Hence $\mathrm{ET}(o_1, o_2) = o_1$. On the other hand, if both o_1 and o_2 are updates, the fact that o_2 and o_1 are contextually serialized implies that o_1 and o_2 update different objects or different attribute of the same object. If they update different objects, excluding the effect of o_2 does not affect o_1. That is, $\mathrm{ET}\ (o_1, o_2) = o_1$. Even if they update the same attribute of the same object, it must be either $o_2 \rightarrow o_1$ or $o_2 \| o_1$. In the case of $o_2 \rightarrow o_1$, excluding the effect of o_2 does not affect o_1. The case of $o_2 \parallel o_1$ will not occur because, otherwise, o_2 should have inclusively transformed with o_1, which results in an identity operation ϕ and will not be added into *HB*. □

Definition 6. A given sequence sq is an ET-Safe Operation Sequence (ETSOS) if for any i, j, where $0 \leq i < j \leq n-1$ and $|sq| = n$, either (1) $c(sq[i]) \prec c(sq[j])$ or (2) $c(sq[i]) \equiv c(sq[j])$ and $t(sq[i]) = \mathrm{ins}$ and $t(sq[j]) = \mathrm{del}$.

By Definition 6, operations in an ETSOS *sq* are ordered by the effects relation $\prec$. If a deletion o_j deletes the object inserted by o_i, or more formally, $\mathrm{t}(o_i) = \mathrm{ins}$, $\mathrm{t}(o_j) = \mathrm{del}$, and $\mathrm{c}(o_j) \equiv \mathrm{c}(o_i)$, then o_i must be ordered before o_j.

Given a sequence *sq* and an operation o, where *sq* and o are contextually serialized, function ETSQ $(o, sq) = o'$ excludes the effects of *sq* from o by

excluding the effects of $sq[n-1]$, $sq[n-2], \ldots,$ and $sq[0]$ in turn, where $n = |sq|$. To ensure the correctness of ETSQ(o, sq), or the correctness of ET for every $sq[i]$, where $0 \leq i < |sq|$, we first build an ETSOS sq' out of sq and then perform ETSQ(o, sq') instead. Lemma 8 implies that sq' does not need to include updates because their existence does not affect the result. The following Lemma 9 states that ETSQ(o, sq') does not need to scan the whole sequence of sq'. It stops at the first operation $sq'[i]$ where $\mathrm{c}(sq'[i]) \prec \mathrm{c}(o)$ or $\mathrm{c}(sq'[i]) \equiv \mathrm{c}(o)$, because the effects relation between o and the rest of the operations in sq' can be inferred.

Lemma 9. *When processing ETSQ(o, sq), where sq is an ETSOS and* $|sq| = n$, *if there exists some* $sq[i]$, $0 \leq i \leq n-1$, *such that* $c(sq[i]) \prec c(o)$ *or* $c(sq[i]) \equiv c(o)$, *then* $c(sq[j]) \prec c(o)$ *for all* $j : 0 \leq j \leq i-1$.

Proof. By Definition 6, if $\mathrm{c}(sq[i]) \prec \mathrm{c}(o)$, then any operation $sq[j]$ that is ordered before $sq[i]$ must satisfy $\mathrm{c}(sq[j]) \prec \mathrm{c}(sq[i])$ or $\mathrm{c}(sq[j]) \equiv \mathrm{c}(sq[i])$. Either implies $\mathrm{c}(sq[j]) \prec \mathrm{c}(o)$. On the other hand, if $\mathrm{c}(sq[i]) \equiv \mathrm{c}(o)$, then $sq[i]$ must be an insertion. Hence any operation $sq[j]$ that is ordered before $sq[i]$ must satisfy $\mathrm{c}(sq[j]) \prec \mathrm{c}(sq[i])$, which implies $\mathrm{c}(sq[j]) \prec \mathrm{c}(o)$. □

We introduce the notation C_{sq} to denote the set of characters that appear in $\mathrm{est}(sq[0])$ and sq. That is, $C_{sq} = \bigcup_{i=0}^{n-1} \{\mathrm{c}(sq[i])\} \cup \mathrm{est}(sq[0])$. Set C_{sq} effectively collects all the characters that appeared before sq is executed and those newly inserted by operations in sq. The following Lemma 10 establishes a sufficient condition under which ETSQ(o,sq) can be correctly processed.

Lemma 10. *Given an operation o and a sequence sq, where sq and o are contextually serialized, then ETSQ(o, sq) is correct if (1) sq is an ETSOS, and (2) for any* $sq[i] \rightarrow o$, *the landmark characters between c(o) and c(sq[i]), if any, are included in set* C_{sq}, *or more formally,* $C_{ld}(c(o), c(sq[i])) \subseteq C_{sq}$.

Proof. We only need to prove the correctness of ET(o, $sq[i]$) for any $sq[i]$ under the unsafe condition. That is, $sq[i] \rightarrow o$, $\mathrm{t}(sq[i]) = \mathrm{del}$, $\mathrm{t}(o) = \mathrm{ins}$, and $\mathrm{p}(sq[i]) = \mathrm{p}(o)$. We prove it by induction on i, as follows.

$n-1$: consider ET$(o, sq[n-1])$. By definition, we know that for $0 \leq k < n-1$, $c(sq[k]) \prec \mathrm{c}(sq[n-1])$ or $c(sq[k]) \equiv \mathrm{c}(sq[n-1])$. By condition $C_{ld}(c(o), c(sq[n-1])) \subseteq C_{sq}$, we know that there does not exist any c such that $\mathrm{c}(sq[n-1]) \prec c \prec c(o)$. Then if $\exists c : c(o) \prec c \prec c(sq[n-1])$, we have $\mathrm{c}(o) \prec \mathrm{c}(sq[n-1])$. Even if there does not exist c such that

Function 5 *BuildETSOS(sq): sq′*

```
1 sq′ ← [];
2 for ( i = 0; i
3    if(t(sq[i]) = upd) continue;
4   o_i ← sq[i]; added ← false;
5    for ( j = |sq′| − 1; j ≥ 0; j − −)
6       if(c( sq′[j]) ≺ c(o_1) or c(sq[j′]) ≡ c(o_i))
7         sq′ ← sq′[0, j] + o_i + sq′[j + 1, |sq′| − 1];
8          added ← true; break;
9       else //transpose sq′[j] and o_i
10           o_i ← ET(o_i, sq′[j]);
11           sq′[j] ← IT(sq′[j], o_i);
12    if(not added)
13      sq′ ← o_i + sq′;
14 return sq′;
```

$c(o) \prec c \prec c(sq[n-1])$, we still get $\mathrm{c}(o) \prec \mathrm{c}(sq[n-1])$ by the tie-breaking policy. Hence $\mathrm{ET}(o, sq[n-1])$ is correct.

$i+1 \rightarrow i$: Assume $\mathrm{ETSQ}(o, sq[i+1, n-1])$ is correct. Let o' be the execution form of o relative to $\mathrm{est}(sq[i+1])$. By Lemma 9, doing $\mathrm{ET}(o', sq[i])$ means that $c(o) \prec c(sq[k])$ where $i+1 \leq k \leq n-1$. Because $C_{ld}(c(o), c(sq[i]) \subseteq C_{sq}$, there does not exist c such that $c(sq[i]) \prec c \prec c(o)$. Then whether or not $\exists c : c(o) \prec c \prec c(sq[i])$, we get $\mathrm{c}(o) \prec \mathrm{c}(sq[i])$. Hence $\mathrm{ET}(o, sq[i])$ is correct. □

Function BuildETSOS(sq) incrementally builds an ETSOS sq'. Initially sq' is empty. Then an element $o_i = sq[i]$ is added into sq'. Due to Definition 6, o_i is skipped if it is an update (line 3). Otherwise, it transposes o_i with operations in sq' from right to left (lines 9–11) until some $sq'[j]$ is found (lines 6–8) such that $\mathrm{c}(sq'[j]) \prec \mathrm{c}(o_i)$ or $\mathrm{c}(sq'[j]) \equiv \mathrm{c}(o_i)$, which is exactly the stop condition defined in Lemma 9. Under this case, o_i is inserted after $sq'[j]$ (line 7). Then the effects relation between o_i and operations in $sq[0, j-1]$ is known due to Lemma 9. If the stop condition never appears, as shown in lines 12–13, o_i is added to the head of sq'.

Because $sq'[j]$ and o_i are contextually serialized, conditions for determining $\mathrm{c}(sq'[j]) \prec \mathrm{c}(o_i)$ or $\mathrm{c}(sq'[j]) \equiv \mathrm{c}(o_i)$ are the same as those defined for ET (Section 3.2). Since we can correctly determine $\mathrm{c}(sq'[j]) \equiv \mathrm{c}(o_i)$ and do not perform $\mathrm{ET}(o_i, sq'[j])$ in this case, the dependency error of Function 2 will not be raised here. In fact, because updates are not considered, the only case for $\mathrm{c}(sq'[j]) \equiv \mathrm{c}(o_i)$ is $\mathrm{t}(sq'[j]) = \mathrm{ins}$, $\mathrm{t}(o_i) = \mathrm{del}$, and $\mathrm{p}(sq'[j]) = \mathrm{p}(o_i)$. In this case, the deletion o_i is added into sq' immediately after the insertion $sq'[j]$ upon which it depends. In lines 9–11, if we can correctly determine the relation between $\mathrm{c}(sq'[j])$ and $\mathrm{c}(o_i)$, then both $\mathrm{ET}(o_i, sq'[j])$ and $\mathrm{IT}(sq'[j], o_i)$ can be correctly executed.

Function 6 *Compute* $\beta(o, v, L)$: β, where$\mathrm{t}(o) = \mathrm{ins}$ and $L + o$ is contextually serialized.

```
1 L' ← BuildETSOS(L + o);
2 δ ← 0; i ← 0;
3 while (L'[i] ≢ o) do
4    if (v(L'[i]) ⊀ v)
5       if(t(L'[i]) = ins)
6          δ ← δ − 1;
7       else //if(t(L'[i]) = del)
8          δ ← δ + 1;
9    i + +;
10 return β ← p(L'[i]) + δ;
```

Function Compute $\beta(o, v, L)$ first builds an ETSOS L' out of $L + o$. After that, o is already somewhere in L', say $L'[i]$. Then excluding L from o is reduced to excluding $L'[0, i-1]$ from o. Recall that the purpose of Computeβ( ) is to compute the position of o relative to state s^{lsp}. Hence we should not exclude operations in L_{lsp}, i.e., those with timestamps less than $o.v$. As shown in the loop of lines 3–9, we pick up operations in $L'[0, i-1]$ that do not belong to L_{lsp} and exclude their effects from o. Since only characterwise operations are concerned, excluding an insertion means to decrease $\mathrm{p}(o)$ by one (lines 5–6), and excluding a deletion means to increase $\mathrm{p}(o)$ by one (lines 7–8).

3.6. DISCUSSIONS OF CORRECTNESS

In function Integrate( ), there are two cases in which we derive and record the effects relation between two concurrent operations o_x and o_y, where o_y is executed before o_x: (1) $\mathrm{t}(o_y) = \mathrm{t}(o_x) = \mathrm{ins}$ and $\mathrm{p}(o_y) = \mathrm{p}(o_x)$; and (2) $\mathrm{t}(o_y) = \mathrm{del}$, $\mathrm{t}(o_x) = \mathrm{ins}$, and $\mathrm{p}(o_y) < \mathrm{p}(o_x)$. By Corollary 1 the former is used for breaking ties in IT, and by Corollary 2 the latter is used for breaking ties in ET. These saved relations ensure the correctness of IT/ET between any concurrent operations.

The correctness of function Computeβ( ) depends on the correctness of function BuildETSOS( ), which in turn relies on the correctness of ET. Note that in line 8 of function Integrate( ) we always include L_{prec} as the input to function Computeβ( ) and, consequently, function BuildETSOS( ). Because all the operations (since the last quiescent state) are built into the ETSOS, the landmark characters between the effect characters of any two causally related operations involved in ET, if any, must have been included. Hence the precondition of Lemma 10 is always guaranteed. That is, ET is always correct.

Therefore, the β values of operations can always be correctly computed. In previous work (Li and Li, 2005a) we have proven that the effects relation between any two concurrent operations can be correctly determined if their β values are correct, and that, based on the correct effects relation, concurrent operations can be correctly (inclusively) transformed along arbitrary

transformation paths. That is, function Integrate( ) can always derive the correct execution form of any remote operation relative to the current state. Hence any remote operation can be correctly executed in its execution state. Additionally, by assumption, any local operation can be correctly executed in its generation state. Consequently, any group editor that implements the control algorithm as given in Figure 2 is correct by the criteria defined in Definition 4.

4. An example

Figure 3 shows a simple scenario that involves four sites. Assume that the initial state is $s^0 =$ "1" and the initial state vector $\langle 0,0,0,0\rangle$. Site 1 performs an operation $o_1 =$ ins(1, "b"). Concurrently, site 2 performs $o_2 =$ del(0), and site 4 performs $o_3 =$ ins(0, "a"). After o_3 is executed, site 3 performs $o_4 =$ ins(1, "c"). We have $o_1 \parallel o_2$, $o_2 \parallel o_3$, $o_1 \parallel o_3$, $o_3 \rightarrow o_4$, $o_1 \parallel o_4$, and $o_2 \parallel o_4$. For simplicity of the presentation, here we only consider insert/delete operations and how these operations are executed at site 1 and site 4. We assume that the execution order at site 1 is o_1, o_3, o_4, o_2, and the order at site 4 is o_3, o_2, o_4, o_1.

Site 1 Let $s_1^1 = \text{exec}(s^0, o_1) =$ "1". Due to $o_3 \parallel o_1$, we inclusively transform o_3 against o_1 and have $o_3^{s_1^1} = \text{IT}(o_3, o_1) =$ ins(0, "a"). Executing $o_3^{s_1^1}$ in s_1^1 results in $s_1^2 = \text{exec}(s_1^1, o_3^{s_1^1}) =$ "a1b".

Next, when o_4 arrives, the history buffer of site 1 is $[\, o_1, o_3^{s_1^1}]$. According to the control algorithm, we transpose it into $sq_h = [o_3]$ and $sq_c = [o_1^{s_1^{1'}}]$, where $s_1^{1'} = \text{exec}(s^0, o_3) =$ "a1" and $o_1^{s_1^{1'}} = \text{IT}(o_1, o_3) =$ ins(2, "b"). Inclusively transforming o_4 against $o_1^{s_1^{1'}}$ in sq_c is simple because the two insertions do not

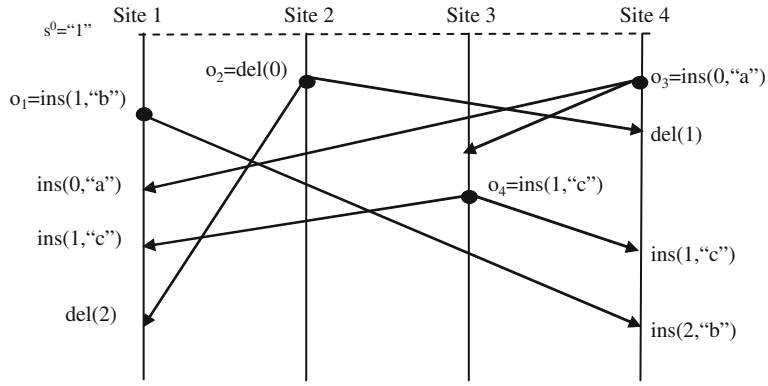


Figure 3. Four sites start from state $s^0 =$ "1" and converge in state $s^4 =$ "acb".

tie. This results in $o_4^{s_1^2} = \mathrm{IT}(o_4, o_1^{s_1^{1'}}) = \mathrm{ins}(1, \text{“c”})$. The execution of $o_4^{s_1^2}$ leads to $s_1^3 = \mathrm{exec}(s_1^2, o_4^{s_1^2}) = \text{“ac1b”}$.

Finally, when o_2 arrives, the history buffer is $[o_1, o_3^{s_1^1}, o_4^{s_1^2}]$. Since all these operations are concurrent with o_2, we simply inclusively transform o_2 against all these three operations one by one: first $o_2^{s_1^1} = \mathrm{IT}(o_2, o_1) = \mathrm{del}(0)$, then $o_2^{s_1^2} = \mathrm{IT}(o_2^{s_1^1}, o_3^{s_1^1}) = \mathrm{del}(1)$, and then $o_2^{s_1^3} = \mathrm{IT}(o_2^{s_1^2}, o_4^{s_1^2}) = \mathrm{del}(2)$. The final state of site 1 is $s_1^4 = \mathrm{exec}(s_1^3, o_2^{s_1^3}) = \text{“acb”}$.

Site 4 Let $s_4^1 = \mathrm{exec}(s^0, o_3) = \text{“a1”}$. First, when o_2 arrives, we get $o_2^{s_4^1} = \mathrm{IT}(o_2, o_3) = \mathrm{del}(1)$ and $s_4^2 = \mathrm{exec}(s_4^1, o_2^{s_4^1}) = \text{“a”}$. Next, when o_4 arrives, the history is $[o_3, o_2^{s_4^1}]$. Due to $o_3 \rightarrow o_4$ and $o_2 \parallel o_4$, we only need to inclusively transform it against $o_2^{s_4^1}$, yielding $o_4^{s_4^2} = \mathrm{IT}(o_4, o_2^{s_4^1}) = \mathrm{ins}(1, \text{“c”})$, the execution of which results in state $s_4^3 = \text{“ac”}$. Finally, when o_1 arrives, the history is $[o_3, o_2^{s_4^1}, o_4^{s_4^2}]$. Since all these operations are concurrent with o_1, we inclusively transform o_1 with these three operations in turn. First $o_1^{s_4^1} = \mathrm{IT}(o_4, o_3) = \mathrm{ins}(2, \text{“b”})$. Then $o_1^{s_4^2} = \mathrm{IT}(o_1^{s_4^1}, o_2^{s_4^1}) = \mathrm{ins}(1, \text{“b”})$ and at the same time we derive relation $\mathrm{c}(o_2) \prec \mathrm{c}(o_1)$ because $\mathrm{t}(o_2) = \mathrm{del}$, $\mathrm{t}(o_1) = \mathrm{ins}$, and $\mathrm{p}(o_2^{s_4^1}) < \mathrm{p}(o_1^{s_4^1})$.

Now there is a tie between the two insertions, $o_1^{s_4^2}$ and $o_4^{s_4^2}$, to be inclusively transformed next. It can be shown that their last synchronization point is state s^0 because the timestamps of o_1 and o_4 are $\langle 1,0,0,0\rangle$ and $\langle 0,0,1,1\rangle$, respectively, which means $v_{min} = \langle 0,0,0,0\rangle$. We need to compute their β values in s^0, by excluding the effects of sequence $L_{ex} = [o_3, o_2^{s_4^1}]$. We first build an ETSOS sequence out of sequence $[o_3, o_2^{s_4^1}]$, which results in $sq =$ [ins(0,“a”), del(1)]. Adding $o_4^{s_4^2}$ into sq yields $sq_1 =$ [ins(0,“a”), ins(1,“c”), del(2)], because pair $\langle o_2, o_4\rangle$ is not found in ER and hence $\mathrm{c}(o_4) \prec \mathrm{c}(o_2)$ is assumed and we need to transpose $o_2^{s_4^1}$ and $o_4^{s_4^2}$. Adding $o_1^{s_4^2}$ into sq yields $sq_2 =$ [ins(0,“a”), del(1), ins(1,“b”)], because pair $\langle o_2, o_1\rangle$ is found in ER and hence $c(o_2) \prec c(o_1)$ and we simply add $o_1^{s_4^2}$ after $o_2^{s_4^1}$. By function Compute $\beta(\,)$, we get $\beta(o_1) = 1$ and $\beta(o_4) = 0$. Since $\beta(o_4) < \beta(o_1)$, we derive relation $\mathrm{c}(o_4) \prec \mathrm{c}(o_1)$. Therefore $o_1^{s_4^3} = \mathrm{IT}(o_1^{s_4^2}, o_4^{s_4^2}) = \mathrm{ins}(2, \text{“b”})$ and the final state of site 4 is $s_4^4 = \mathrm{exec}(s_4^3, o_1^{s_4^3}) = \text{“acb”}$.

Similarly it can be shown that the final states of all the four sites converge to $s^4 = \text{“acb”}$. We can verify the correctness of s^4 by our consistency model: Since o_1 is generated in state s^0, we infer ‘1’ $\prec$ ‘b’ by definition. Similarly we can infer ‘a’ $\prec$ ‘1’ from the generation of o_3 in s^0, and ‘a’ $\prec$ ‘c’ $\prec$ ‘1’ from the

generation of o_4. Globally we infer the total order of all characters 'a' $\prec$ 'c' $\prec$ '1' $\prec$ 'b'. After character '1' is deleted by o_2, the correct result must be "acb", which is exactly s^4. This result preserves the effects relation of all operations.

5. Performance evaluation

Since the original work of Ellis and Gibbs (1989), high local responsiveness and unconstrained interaction have been accepted as requirements in group editors and have been the main advantages of OT over alternative concurrency control methods. However, these benefits often come at the expense of semantic consistencies or the violation of user intentions (Sun and Ellis, 1998; Li and Li, 2004). While recent OT algorithms such as SDT (Li and Li, 2004, 2005a) achieve syntactic consistency, staying aware of each other's progress is still the basis for collaborators to resolve semantic conflicts in a timely manner.

Function Integrate( ) transforms a given operation o against concurrent operations that have been executed before o is executed locally. It is called in existing group editors in two slightly different ways. While the first and traditional way transforms local and remote operations equally, as in Grove (Ellis and Gibbs, 1989) and Joint Emacs (Ressel et al., 1996), the second and more recent way always executes local operations immediately as soon as they are generated (without being transformed) and only transforms remote operations, as in Reduce (Sun et al., 1998; Yang et al., 2000) and this work (SDTO). Recent group editors that are built from transparently adapting familiar single-user editors, such as ICT (Li and Li, 2002) and CoWord (Xia et al., 2004), generally follow the second approach. In the first approach, although the generation of local operations (by the user) is not blocked, the transformation and execution of local operations are delayed if a remote operation is being integrated (transformed and executed). In the second approach, the generation of local operations is often blocked while a remote operation is being integrated. Either way the local response time, i.e., the delay between a local operation is generated by the user and its effect becomes visible on the user interface, is sensitive to the performance of function Integrate( ).

In addition to its impact on local responsiveness, the performance of Integrate( ) also affects the timeliness of workspace awareness. In group editors, every site needs to integrate remote operations so that the progress of all sites is known among the users. Editing operations are often accumulated to reduce bandwidth usage as well as interferences to the users (Li et al., 2000). This is particularly true in asynchronous group editing and mobile computing environments in which the collaborating sites do not continuously communicate. Hence the time it takes to integrate remote operations effectively determine how timely the collaborators see the effects of each other's operations.

Therefore, the performance of Integrate( ) is relevant in group editors. Unfortunately, no previous work to our knowledge has studied this issue. In

this section we analyze the performance of the SDTO algorithm, present some performance experiments, and analyze the experimental results. In particular, we identify factors that affect the performance of Integrate( ) and ultimately the feedback and feedthrough (Gutwin and Greenberg, 2002) time of (OT-based) group editors.

5.1. THEORETICAL ANALYSES

In the following, let $|HB| = n = M + N$, where $M = |L_{par}|$ and $N = |L_{prec}|$. Suppose the initial state is s^0, the generation state of o is s^N, and the current state is $s^n = \text{exec}(s^0, HB)$. Then $s^N = \text{exec}(s^0, L_{prec})$ and $s^n = \text{exec}(s^N, L_{par})$. Function Integrate($o$,$HB$) includes the effects of L_{par} into o^{s^N} to get o^{s^N}.

Worst-case complexity: The time complexity of Integrate(o,HB) includes that of TransposeR2L(o,HB) and the IT loop of lines 3–14. In the worst case, L_{prec} contextually follows L_{par} in HB, which means every operation in L_{prec} must be transposed with all operations in L_{par}. Hence the time complexity of TransposeR2L( ) is $O(M \cdot N) = O(n^2)$. In the worst case, we have to compute the β values of o and every $L_{par}[i]$, if $\text{t}(o) = \text{ins}$, $\forall\ i$: $\text{t}(L_{par}[i]) = \text{ins}$ and $\text{p}(o) = \text{p}(L_{par}[i])$. The worst case happens only when L_{par} inserts a continuous sequence of characters starting from $\text{p}(o^{s^N})$, and $\forall i : c(L_{par}[i]) \prec c(o)$. As a result, each time $\text{IT}(o, L_{par}[i])$ is executed, $\text{p}(o)$ is increased by one to tie with $\text{p}(L_{par}[i+1])$ and Computeβ( ) is called again. The complexity of function Computeβ( ) is dominated by that of function BuildETSOS( ), which is asymptotically $O(|L|^2)$, where L is the input sequence of Computeβ( ). Hence the complexity of the IT loop in Integrate( ) is $\sum_{i=1}^{M} c \cdot (N+i)^2 = O(n^3)$, which dominates the complexity of TransposeR2L( ). Therefore, the worst-case time complexity of Integrate( ) is $O(n^3)$. The worst-case space complexity of ER is $O(n^2)$ because, for every o, the number of effects relation pairs stored is M.

Average-case complexity: To simplify analysis, we assume that all operations are mutually independent and all distributions are uniform. Specifically, (1) the types of operations are uniformly distributed, i.e., $\Pr[\text{t}(o) = \text{ins}] = \Pr[\text{t}(o) = \text{del}] = \Pr[\text{t}(o) = \text{upd}] = \frac{1}{3}$; (2) the positions of operations are uniformly distributed, i.e., $\Pr[\text{p}(o) = i] = \frac{1}{D}$, where $i = 0, 1, 2, \ldots, D$ and D is the length of the document; (3) every operation $o_i \in HB$ is equally likely to be $o_i \parallel o$ or $o_i \rightarrow o$; (4) in Computeβ, half of the operations in L have effects to the left of o. These assumptions may not sound realistic. However, they are a convention in algorithm analysis for studying performance trends: even if specific distributions vary, the results will often vary only by some constant while the asymptotic magnitude (or performance trend) remains the same.

Given operation o_2, ER stores pair $\langle o_1, o_2 \rangle$ for every $o_1 \parallel o_2$, if relative to some definition state s, (1) $\text{t}(o_1) = \text{del}$, $\text{t}(o_2) = \text{ins}$, $\text{p}(o_1) < \text{p}(o_2)$, and o_1 is executed before o_2, or (2) $\text{t}(o_1) = \text{t}(o_2) = \text{ins}$ and $\text{p}(o_1) = \text{p}(o_2)$. First consider case

(1), for every o_2, the number of pairs is $M \cdot \Pr[t(o_2) = ins] \cdot \Pr[t(o_1) = del] \cdot \Pr[c(o_1) \prec c(o_2)] = M \cdot \frac{1}{3} \cdot \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{18} \cdot M$. Then consider case (2), the number of pairs is $M \cdot \Pr[t(o_1) = t(o_2) = ins] \cdot \Pr[p(o_1) = p(o_2)] = M \cdot \frac{1}{3} \cdot \frac{1}{3} \cdot 1 \cdot \frac{1}{D} = \frac{1}{9 \cdot D} \cdot M$. On average, we can assume $D \gg 2$. Hence the space is dominated by term $\frac{1}{18} \cdot M$, which is lower than the worst-case complexity by a factor of 18.

In fact, a more careful examination of the algorithm even suggests that it is not necessary to store the effects relation between concurrent insertions. In function Integrate( ), the relation is computed for breaking ties in the IT( ) function of line 13. But after that, this relation will never be used. The only other places that IT( ) is called are line 16 of function TransposeR2L( ) and line 11 of BuildETSOS( ). However, in these two places, ET( ) is always called before IT( ). According to our proofs in Section 3.2, ET( ) can correctly determine the relation between any two insertions. Hence the actual implementation of SDTO does not need to store the relation of concurrent insertions.

The average time complexity of TransposeR2L(o,HB) is calculated as follows. When considering some $HB[i]$, due to uniform distribution, $\frac{i-1}{2}$ operations are in L_{prec} and the other $\frac{i-1}{2}$ in L_{par}. Thus $HB[i]$ must be transposed with the $\frac{i-1}{2}$ operations in L_{par} if $HB[i] \rightarrow o$, the probability of which is $\frac{1}{2}$. Hence the expected complexity is $\sum_{i=1}^{n} (\frac{i-1}{2} \cdot \frac{1}{2}) = O(n^2)$. The probability of the quadratic case in the IT loop of Integrate( ) is approximately $\Pr[t(o_1) = t(o_2) = ins] \cdot \Pr[p(o_1) = p(o_2)] = \frac{1}{3} \cdot \frac{1}{3} \cdot 1 \cdot \frac{1}{D} = \frac{1}{9 \cdot D}$. Hence the expected time complexity of Integrate( ) is $O(n^2 + \frac{1}{9 \cdot D} \cdot n^3)$, which is reduced to $O(n^2)$ when D is large enough and it becomes rare for the worst cases to happen.

5.2. EXPERIMENTAL RESULTS AND ANALYSES

To be more concrete, we performed experiments to understand what the above theoretical complexities mean in group editors. In particular, interactive applications such as single-user and group editors are expected to provide a local response time of 100 ms or less (Shneiderman, 1984; Bhola et al., 1998). We hope to quantitatively identify to what extent OT algorithms can satisfy this interactive requirement and which issues should be addressed in order to achieve responsiveness.

The algorithm was implemented in C++, compiled by GNU g++ v3.3.3, and executed on a computer running RedHat Linux kernel 3.3.3-7 with an Intel Pentium-4 3.4 GHz CPU and 1 GB RAM. These experiments give a general idea of how much time (in ms) it takes to integrate a remote operation o into a history buffer HB of size $|HB| = M + N$, in which M operations are concurrent with o and N operations happened before o. For simplicity, all N operations that happened before o are stored in the log after the M operations concurrent with o. Hence function TransposeR2L( ) executes in its worst cases in all the experiments. Additionally, we assume that

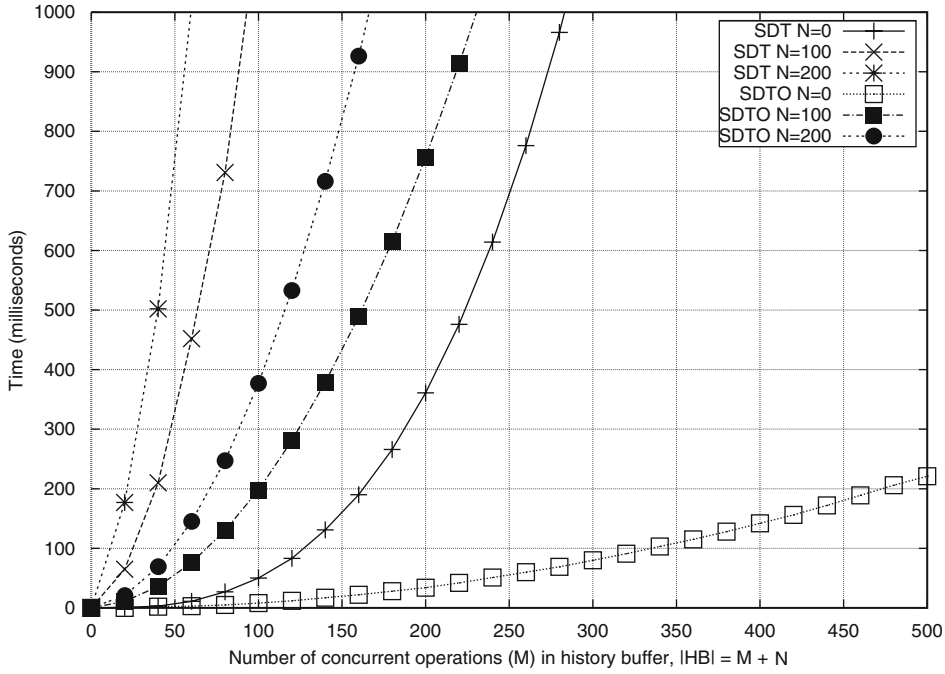
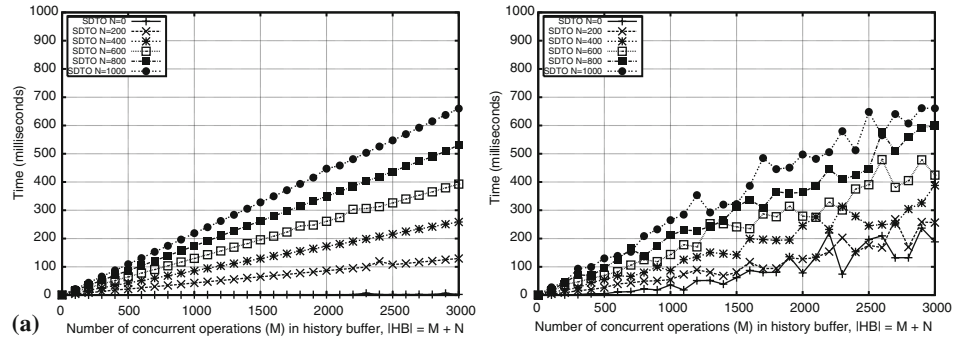


Figure 4. Worst-case execution time of Integrate( ) in SDT and SDTO.

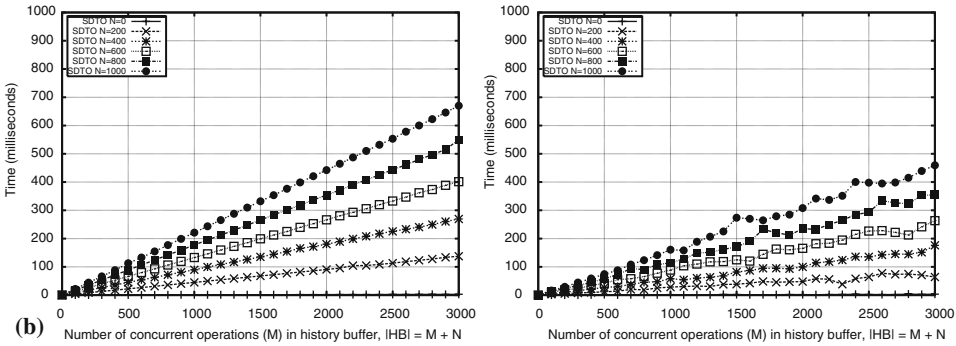
every character in the document has two different attributes. All update operations in the experiments modify either attribute with equal probability. All the experiments were executed 30 times to get the averaged execution time of function Integrate( ). The source code and more information about the experiments are available at http://cocasoft.csdl.tamu.edu/~lidu/projects/CE/ and also the online appendix attached to this paper.

Figure 4 shows the worst-case time of Integrate( ) in SDT and SDTO, i.e., when all operations are insertions and o conflicts with each of the M concurrent operations. In general, SDTO performs better than SDT. For example, to integrate one remote operation within the interactive threshold of 100 ms, the size of the history buffer (HB) in SDT is bound by around 120, while this limit is pushed to around 250 in SDTO. Nevertheless, the curves show that, in the worst cases, SDTO can only provide good local response time when $|HB|$ is small, e.g., at the magnitude of two hundred operations.

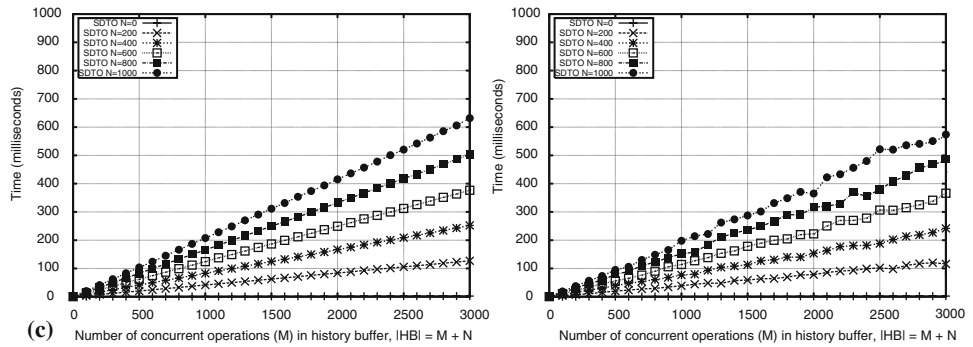
Figure 5 shows experiments in which all operation positions are uniformly distributed over the shared document. We study how the execution time is sensitive to the document size D and ratio of different types of operations. All the figures on the left are the results for a longer document ($D = 1{,}000{,}000$ characters), and those on the right are for a short document with $D = 100$



Ratio of operation types #ins:#del:#upd = 90%: 5 % : 5%.



Ratio of operation types #ins:#del:#upd = 35%:30%:35%.



Ratio of operation types #ins:#del:#upd=10%:5%:85%.

Figure 5. Expected execution time of Integrate() under uniform distribution with initial document size (L) $D = 1{,}000{,}000$, and (R) $D = 100$. (a) Ratio of operation types #ins:#del:#upd = 90% : 5% : 5%. (b) Ratio of operation types #ins:#del:#upd = 35% : 30% : 35%. (c) Ratio of operation types #ins:#del:#upd = 10% : 5% : 85%.

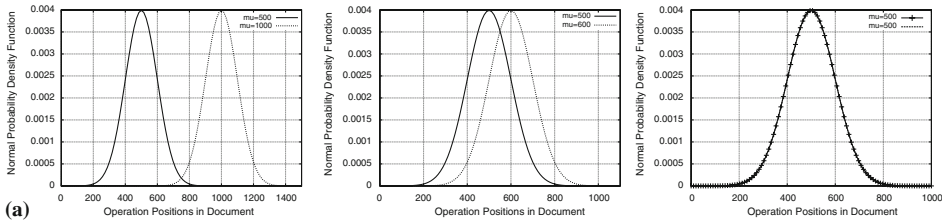
characters. According to the above analysis, when the document is longer, the impact of the worst cases on the overall performance will be smaller. Presumably there is a higher chance of running into the worst cases if D is smaller and/or the rate of insertions is larger. In our experiments, operation

positions are uniformly spread over the document with probability $\frac{1}{d}$, where d is the size of the current document, and operation types are generated by the given ratio. Given the drastic difference between the initial document sizes and the relative small number of operations, the difference between D and d and its impact on uniform distribution can be ignored.

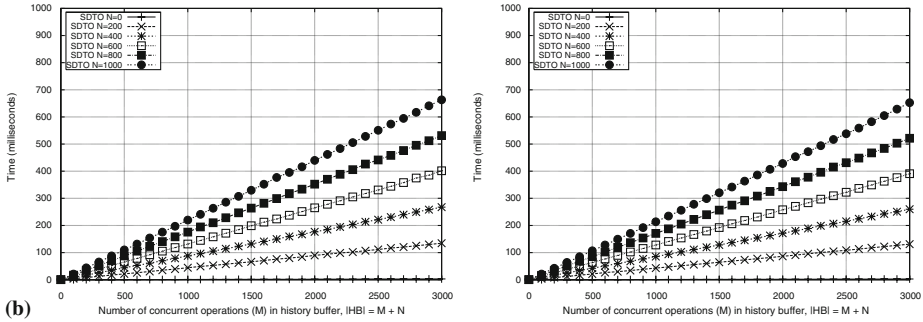
Since the worst cases are only caused by insertions at the same positions, we simulate on three different operation type ratios: (a) insertions (90%) dominate deletions and updates (5% each); (b) all operation types are of approximately equal probability; and (c) updates (85%) dominate insertions (10%) and deletions (5%). It is clear from the curves that (1) the ratio of operation types has little impact on performance when D is large; (2) the performance fluctuates more obviously when D is small and the rate of insertions increases; and (3) the execution time is stable regardless of the initial document size when the rate of insertions is small. It should be noted that conflicts of deletions and updates also affect performance. In the SDTO algorithm, this type of conflicts effectively decreases the size of the operation log. Hence we can notice a relatively small speedup when the rate of conflicts increases.

Figure 6 shows the simulation results when normal distribution (with parameter $\sigma = 100$) is assumed. These experiments study the impacts of operation locality on performance. In group editing/writing practices, it is not uncommon that there is a labor division among the cooperating sites (Baecker et al., 1993; Noel and Robert, 2004). Then the users modify different regions of the shared document most of the time but may cross regions from time to time. To simplify experiments, we only simulate two sites: site 1 generates M operations with with parameter μ_1, and site 2 concurrently generates N operations with parameter μ_2. After the N causally preceding operations are executed at site 1, site 2 generates o with parameter μ_2. Figure 6(a) approximately depicts how the positions of these two groups of operations are distributed over the shared document. From left to right, the two regions overlap with probabilities of 1.2% ($\mu_1 = 500$ and $\mu_2 = 1000$), 61.8% ($\mu_1 = 500$ and $\mu_2 = 600$), and 100% ($\mu_1 = \mu_2 = 600$).

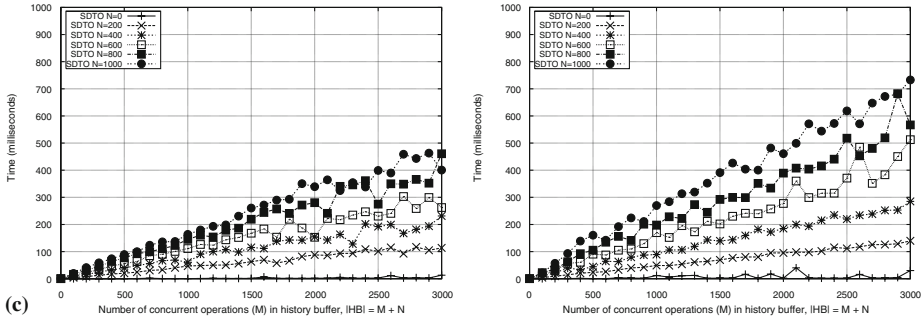
In the three experiments shown in Figure 6, we study how operation locality as well as operation ratio affect performance. From (b) to (d), the operations are generated with increasing locality. In each experiment, the left side shows a uniform distribution of operation types, and the right side shows the curves when insertions dominate. From the curves we can draw the following conclusions: (1) when the two regions barely overlap, the performance is not sensitive to the change of operation ratio; (2) as the editing regions get closer and the rate of insertion ties (worst cases) increases, the curves thrash more obviously and execution time gets longer; (3) the performance improves as the rate of conflicts of concurrent deletions and updates increases; and (4) when the editing regions get closer, the negative



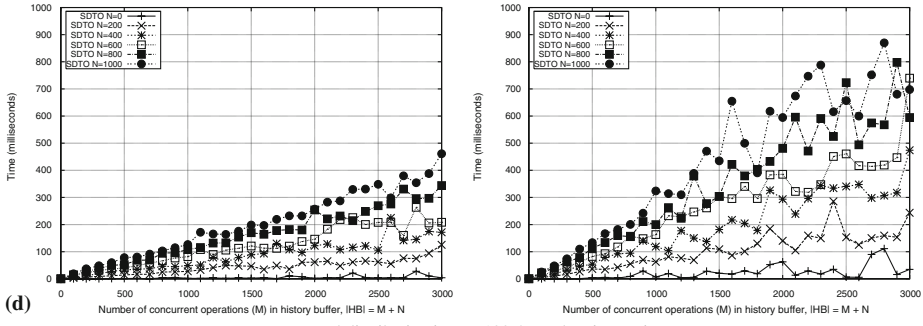
(a) Approximately operations are normally distributed in two regions.



(b) Normal distribution in two 1.2% overlapping regions.



(c) Normal distribution in two 61.8% overlapping regions.



(d) Normal distribution in two 100% overlapping regions.

Figure 6. Expected execution time of Integrate() under normal distribution. Ratio of operation types #ins:#del:#upd (L) 33% : 33% : 34%, and (R) 90% : 5% : 5%. (a) Approximately operations are normally distributed in two regions. (b) Normal distribution in two 1.2% overlapping regions. (c) Normal distribution in two 61.8% overlapping regions. (d) Normal distribution in two 100% overlapping regions.

impact caused by insertion ties dominates the positive impact by the deletion/update conflicts.

5.3. DISCUSSIONS OF EXPERIMENTAL RESULTS

The experimental results presented above are obtained from simulated workloads rather than empirical data. We argue that this methodology is appropriate for the purposes of this paper: on the one hand, the main purpose of this paper is to study the performance of an OT-based concurrency control algorithm (SDTO) instead of a group editor that implements this algorithm. We hope to identify general performance trends of OT algorithms as well as to what extent those trends inform the design of group editors. Simulation is a well-accepted method for similar purposes in the literature, e.g., (Bhola et al., 1998; Gutwin and Penner, 2002; Chung and Dewan, 2004). Whether or not empirical data is used will not change our analyses and conclusions about the *performance trends*. On the other hand, although a few sophisticated group editors are emerging (such as CoWord (Xia et al., 2004)), realtime group editing is not a widely adopted practice yet. The usability of these group editors themselves is still to be evaluated. Collecting *actual* group editing data is still difficult, system-dependent, and effectively *artificial* at the current stage. In addition it would be difficult to justify why the setup scenarios of use are *typical* and *realistic*.

From the plots in Section 5.2, we can observe the following general performance trends of SDTO. It is worth noting that specific data may differ in actual implementations and execution environments. However, these differences will not change our analyses and conclusions. First, the ratio of operation types impacts performance. More specifically, the average execution time increases with the probability of insertion conflicts in the editing sequence. When the insertion conflicts are rare, e.g., in a large document or when the rate of insertions is low, the curves are rather straight. As the chance of insertion conflicts increases, the curves thrash more significantly. This matches our analysis that the $O(n^3)$ time of Integrate( ) is only caused by concurrently inserting characters at the same positions.

Second, given the same ratio of operation types, the performance is sensitive to the locality of editing operations. As shown in Figure 6, when operation positions are separated in different areas of the document, the performance is stable (with only very few pikes on the curves). However, as the operation positions tends to focus on certain areas, the performance tends to fluctuate. The chance of delete/update conflicts does have some positive impact on performance due to our conflict resolution policies. However, the impact is very limited and negligible compared to the impacts of other factors.

Third, while operation ratio and locality do affect performance, the impacts do not cause magnitude differences. The fluctuation is always within certain bounds around the corresponding execution time of cases in which insertion conflicts are less probable. It is obvious that the expected execution time of Integrate( ) is significantly lower than its worst-case execution time. When the costs of the worst-case execution is amortized, the expected performance of Integrate( ) is dominated by TransposeR2L( ), which is asymptotically quadratic to the operation log size in both worst and average cases. That is, the average performance of Integrate( ) is considerably stable.

Fourth, the SDTO algorithm is sufficient for real-time group editors. In most of the experiments that are not pure worst cases, it takes less than 0.5 s to integrate a remote operation o into an operation log of $M \leq 3000$ concurrent operations and $N \leq 1000$ causally preceding operations. Within the more demanding interactive threshold of 100 ms, Integrate( ) can finish with $M \leq 2000$ and $N \leq 100$, or $M \leq 1000$ and $N \leq 200$ in almost all the experiments. This implies that the system can propagate operations in batches up to a couple of hundreds instead of on a per-operation basis even in realtime group editing. As a result, interferences between users can be lowered and bandwidth can be saved wherever possible. This observation confirms our earlier analyses in Li et al. (2000). It also implies that the group editor should proactively reach quiescent states as the log size reaches a few thousand operations, if the system must provide an expected interactive response time of 100 ms. This observation is confirmed by a related design in the well-known Reduce group editor by Sun et al. (1998) and Yang et al. (2000).

Last but not least, OT algorithms that only support character-wise operations and operation-wise integration are not sufficient for non-realtime group editors. The SDTO algorithm as presented in this paper is only suitable for realtime group editing. Because it can only integrate one character-erwise operation at a time, the time it takes to transform a sequence with another is proportional to the size of the sequences. This renders it inefficient and ineffective for asynchronous group editing tasks which often accumulate operations over time and only propagate/integrate sequences periodically.

More concretely, Figure 7 shows the time to integrate one operation into a log of M concurrent operations and N causally preceding operations, and the time to integrate a sequence of N operations with a concurrent sequence of M operations. In both experiments, function Integrate( ) was called to integrate one operation at a time. The operation distributions were the same as in the left chart of Figure 6(b). Due to the low probability of insertion ties, the integration of one operation can finish within 100 ms when $M \leq 2000$ and $N \leq 200$ or $M \leq 1000$ and $N \leq 400$. However, it takes about 10 s to integrate a sequence of 300 operations with another sequence of 1000 operations, and over 50 s to integrate a sequence of 500 operations with another of 2000 operations.

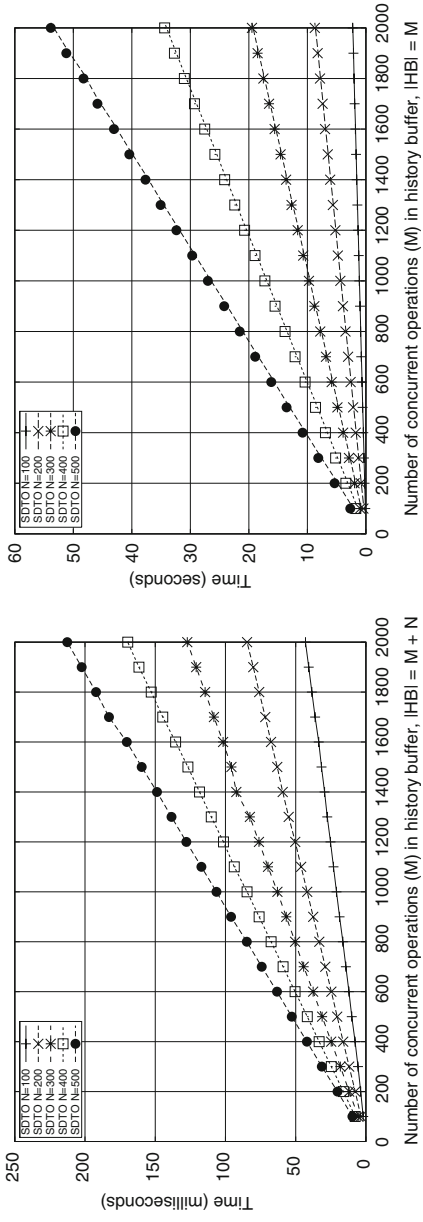


Figure 7. Following the distributions in Figure 6(b.L), (L) time to integrate one operation into a log of $(M+N)$ operations; (R) time to integrate two concurrent sequences.

The experiments in Figure 7 lead to the following two observations. First, if all operations in a sequence are integrated as a whole, naively integrating operations in a sequence one by one can cause significant delays. However, if these operations are integrated one by one over time, it may take so long (e.g., over a minute) that the system fails to provide timely workspace awareness. This suggests that OT algorithms devised for non-realtime group editing must support special-purpose sequence transformation such that sequences can be integrated as a whole more efficiently, as pioneered in Li et al. (2004) and Shen and Sun (2002a, c).

Second, the support of stringwise operations in OT, as pioneered in Sun et al. (1998), can improve the efficiency and effectiveness of both realtime and non-realtime group editors. When operations are highly localized, accumulating characterwise operations into stringwise operations can effectively reduce the size of the sequences to be propagated and transformed, which implies better performance when one operation or a sequence is integrated. However, stringwise operations alone (without sequence transformation) will likely follow similar performance patterns as discussed above when the number of stringwise operations involved in transformation grows.

6. Related work

SDT and SDTO are different versions of the same algorithm that serve different purposes. The original work on SDT (Li and Li, 2004, 2005a) addressed the correctness problems of OT. Performance problems were left out and are addressed in this paper. In other words, SDT only runs in worst cases while SDTO addresses the average cases. Hence there is no need to compare the average cases of SDT and SDTO. Additionally, this paper contributes a new method for computing β values through building ET-safe sequences, which is different from the state difference based method presented in Li and Li (2004, 2005a). The average cases in SDTO are based on and justified by formal proofs presented in this paper. It is due to the newly introduced average cases and Computing β method that SDTO outperforms SDT. As analyzed in Section 5, the performance improvement is important because it warrants better responsiveness and awareness in group editors. Otherwise, group editors have to frequently reach quiescent states in order to maintain transformation efficiency, which could be costly and inconvenient, especially in asynchronous group editing and mobile computing environments.

The purpose of this paper is neither to suggest that previous OT algorithms suffer from performance problems nor to suggest that the performance of SDTO is better than previous algorithms. Instead, this work identifies general performance patterns and factors that affect responsiveness by studying SDTO, a ''typical'' OT algorithm. Some of our observations are consistent with previous work but no quantitative studies have been reported on OT

algorithms previously. As will be discussed shortly, the resemblance between SDTO and alternative OT algorithms such as SOCT2 (Suleiman et al., 1998) and GOTO (Sun and Ellis, 1998; Sun et al., 1998, 2004; Sun, 2002) warrants that our results and analyses have general implications. Hence there is no need to compare the performance of alternative algorithms.

To our knowledge, most modern OT algorithms use IT and ET functions. Sun (2002) and Sun and Ellis (1998) propose that $\text{est}(o_1) = \text{est}(o_2)$ and $\text{est}(o_1) = \text{exec}(\text{est}(o_2), o_2)$ are the preconditions for $\text{IT}(o_1, o_2)$ and $\text{ET}(o_1, o_2)$ to preserve operation intentions, respectively. However, our proofs in Section 3.1 and 3.2 show that these two conditions are not really sufficient. IT and ET functions cannot work correctly under these conditions without extra constraints. Our constraints are based on the concept of operation effects relation, which is alternative to previous concept of operation intention (Sun et al., 1998; Sun, 2002) or user intention (Suleiman et al., 1998; Vidot et al., 2000). The effects relation is relatively easier to formalize and verify.

According to our analyses, the effects relation violation (ERV) problem arises whenever the IT/ET function cannot correctly determine the effects relation of two given operations. Using the term of Sun et al.[1], ERV is referred to as the false-tie problem, in the sense that the IT/ET function may not be able to break ties correctly. However, their definition and solution of the false-tie problem have not been published to our knowledge.

Previous understanding of OT problems in the literature has been focused on IT, e.g., the so-called TP2 puzzle (Suleiman et al., 1998; Sun et al., 1998; Imine et al., 2003). The research community has paid little attention to the problems of ET prior to this work. Although it is understood (e.g., in Suleiman et al., (1998) and Sun et al. (1998)) that the problem of IT may only happen in rare situations, no previous work has formally presented definitions and proofs as to what those situations exactly are and why they do not happen elsewhere. The proofs in this paper are the first that formally establish the sufficient conditions for IT/ET functions to correctly determine the effects relation, and identify exactly under which conditions the tie-breaking problems may happen in both IT and ET. Our conditions are sufficient (mathematically) because, once they are satisfied, IT and ET are guaranteed to be correct.

However, the fact that the worst cases are rare by no means undermines the contributions of the presented work. First, this is the first formal proof of the exact conditions under which these cases happen and why they are rare. The proofs develop a more in-depth understanding of the nature of the tie-breaking problems. Second, this fact implies that our performance results generally extend to alternative OT algorithms. As analyzed in Section 5, the expected time it takes to integrate one remote operation is dominated by the time it takes to transpose the operation log. The transposeR2L( ) procedure is also used in both GOTO (Sun and Ellis, 1998) and SOCT2 (Suleiman et al.,

1998). Although no performance experiment was reported previously, we believe that they follow similar performance patterns as SDTO, if they also have fixed the tie-breaking problems.

Although in general the expected integration time is quadratic in the size of the operation log, group editors do not really need to maintain all the executed operations in the log (or not all operations in the log must be involved in transformations). As has been understood in the literature (Sun et al., 1998), each time a quiescent state is reached in the system, the log can be swapped out, because these operations will not be needed for transforming any incoming operations afterwards. Hence a relatively small log can be kept most of the time in realtime group editors, which also justifies the relatively small values of M and N in our performance experiments. However, no previous work has quantitatively studied how frequently the system should reach quiescent states, how often the operations should be propagated, and how sensitive the integration time is to the log size. Our results provide some insights for answering these questions, which in turn can inform the design of workspace awareness mechanisms and synchronization policies in group editors.

7. Conclusions

This paper makes the following two major contributions: First, it formally establishes new sufficient conditions for ensuring the correctness of IT/ET functions, and identifies the situations under which the correctness problems and the worst-case execution time may happen in OT algorithms. Secondly, it evaluates the performance of SDTO and argues that the results have general implications on the performance of other recent OT algorithms and the design of group editors. This is the first work that evaluates the performance of OT algorithms in the literature. Our results suggest that most OT algorithms can provide satisfactory performance for supporting realtime group editing tasks in which editing operations are continuously communicated among collaborating sites. However, in non-realtime environments where communication is not continuous and operations are considerably accumulated before propagation, OT algorithms that only support characterwise operations, such as SOCT2 (Suleiman et al., 1998) and SDTO, may not be able to provide good performance.

While the above two contributions are original and significant, the following two contributions are relatively incremental over our early work on SDT (Li and Li, 2004, 2005a): First, we introduce a new update primitive, which extends the application scope of SDT, and a simplified method for computing β values. Secondly, we improve the expected execution time of SDT from $O(n^3)$ to $O(n^2)$, where n is the size of the operation log. These incremental contributions are also important because SDT is the first OT

work that explicitly and systematically formalizes and addresses a class of tie-breaking problems, which make it difficult to achieve convergence in pure peer-to-peer group editors in previous work. This paper corroborates SDT and improves our understanding of the nature of these problems.

Correctness and performance are the basis of the usability of consistency control algorithms (Ressel et al., 1996). Although this paper formally proves that the correctness problems of OT happen only under rare situations, solving these problems is the basis for ensuring the correctness of OT algorithms and group editors. The performance improvement is important given that the algorithm is expected to provide interactive local response time and timely awareness for collaborators to resolve semantic conflicts. Our experiments and analyses suggest that responsiveness and awareness of OT-based group editors are sensitive to a number of factors, including the size of the operation log, operation type ratios, operation locality, and policies for synchronizing operations and reaching quiescent states. We also observe that, for group editors to provide satisfactory performance in a range of group editing tasks, stringwise operations and sequence transformation must be supported.

For simplicity, we only formalize and prove the new contributions that complement our early work in Li and Li (2004, 2005a), specifically the sufficient conditions of IT/ET and the new method of computing β values. A more rigorous formalization of the effects relation is left to Li and Li (2005b).

In future research, we plan to extend this work along the following directions: First, we will consider stringwise operations, as in Sun et al. (1998), sequence transformation, as in Li et al. (2004) and Shen and Sun (2002a, c), and group undo, as in Sun (2002). Second, due to its lack of a central component, the support of late-joining in this work seems an interesting and challenging issue. The algorithm will have to be extended to address problems such as how to force a quiescent state and how to elect a site to transfer its state to the new comer. These issues also seem relevant to the overall performance of the algorithm. Some pioneer work in this direction has been done by Sun et al. (1998). Third, this work explicitly assumes a reliable network with reliable nodes and communication links. It will also be necessary to consider fault tolerance in future research. Fourth, the evaluation part could be further extended such that realistic group editing traces are used to test the actual performance of this algorithm. We plan to perform realistic experiments after we implement the algorithm into a real group editor, especially after the above algorithmic extensions have been made, such as stringwise operations and sequence transformation.

In addition, while the performance experiments as presented in this paper have general implications on a range of group editing scenarios, they could be extended to be more relevant to real-time group editing. For example, the experiment parameters could be set such that there are significantly more

causally preceding operations (say around 100,000) than concurrent operations (say less than 100) in the operation log when a remote operation is being integrated. This is mainly for two reasons: First, in real-time sessions, operations are usually propagated instantly and hence it is unusual to have a large number of concurrent operations in the log even for a relatively large group of users. Secondly, if operations in the log are preserved for undo, a large number of causally preceding operations may build up over time.

Notes

1. This information is due to personal communication with Chengzheng Sun in February and October 2004.

Acknowledgements

We thank the anonymous expert referees for their insightful and constructive reviewer comments, which greatly helped improve the presentation of this paper. We are also grateful to the Associate Editor, Prasun Dewan, for coordinating the review process. In addition, Jiong Yang at Case Western Reserve University provided valuable discussions on the performance evaluation part in October, 2004. This work is supported in part by the National Science Foundation under CAREER award IIS-0133871.

References

Baecker, Ronald M., Dimitrios Nastos, Ilona R. Posner, and Kelly L. Mawby (1993): The User-Centered Iterative Design of Collaborative Writing Software. In *Proceedings of the InterCHI'93 Conference on Human Factors in Computing Systems*, April 1993, pp. 309–405.

Bhola, Sumeer, Guruduth Banavar and Mustaque Ahamad (1998): Responsiveness and Consistency Tradeoffs in Interactive Groupware. In *Proceedings of the ACM CSCW'98 Conference on Computer-Supported Cooperative Work*, Seattle, November 1998, pp. 79–88.

Chung, Goopeel and Prasun Dewan (2004): Towards Dynamic Collaboration Architecture. In *Proceedings of the ACM CSCW'04 Conference on Computer-Supported Cooperative Work*, November 2004, pp. 1–10.

Dewan, Prasun, Rajiv Choudhary and Honghai Shen (1994): An Editing-Based Characterization of the Design Space of Collaborative Applications. *Journal of Organizational Computing*, vol. 4, no. 3, pp. 219–240.

Clarence A., Ellis and S.J. Gibbs (1989): Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD'89 Conference on Management of Data*, pp. 399–407, Portland, Oregon.

Ellis, Clarence A., S.J. Gibbs and G.L. Rein (1991): Groupware: Some Issues and Experiences. *Communications of the ACM*, vol. 34, no. 1, pp. 38–58January.

Gutwin, Carl and Saul Greenberg (2002): A Descriptive Framework of Workspace Awareness for Realtime Groupware. *Computer Supported Cooperative Work*, vol. 11, no. 1–2, pp. 411–446.

Gutwin, Carl and Reagan Penner (2002): Improving Interpretation of Remote Gestures with Telepointer Traces. In *Proceedings of the ACM CSCW'02 Conference on Computer-Supported Cooperative Work*, November 2002, pp. 49–57.

Imine, Abdessamad, Pascal Molli, Gerald Oster and Michael Rusinowitch (2003): Proving Correctness of Transformation Functions in Real-Time Groupware. In *Proceedings of the European Conference on Computer Supported Cooperative Work (ECSCW'03)*, September 2003.

Li, Du and Rui Li (2002): Transparent Sharing Interoperation of Heterogeneous single-user applications. In *Proceedings of the ACM CSCW'02 Conference on Computer-Supported Cooperative Work*, November 2002, pp. 246–255.

Li, Du and Rui Li (2004): Preserving Operation Effects Relation in Group Editors. In *Proceedings of the ACM CSCW'04 Conference on Computer-Supported Cooperative Work*, November 2004, pp. 457–466.

Li, Du and Rui Li (2005a). An approach to Ensuring Consistency in Peer-to-Peer Real-Time Group Editors. *Computer Supported Cooperative Work* (Accepted).

Li, Du, Chengzheng Sun, Limin Zhou and Richard R. Muntz (2000): Operation Propagation in Real-Time Group Editors. *IEEE Multimedia Special Issue on Multimedia Computer Supported Coooperative Work*, vol. 7, no. 4, pp. 55–61.

Li, Rui and Du Li (2005b): A New Operational Transformation Framework for Real-Time Group Editors. *IEEE Transactions on Parallel and Distributed Systems* (Accepted).

Li, Rui, Du Li and Chengzheng Sun (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.

Noel, Sylvie and Jean-Marc Robert (2004): Empirical Study on Collaborative Writing: What do Co-Authors Do, Use, and Like. *Computer Supported Cooperative Work*, vol. 13,, 63–89.

Ressel, Matthias, Doris Nitsche-Ruhland and Rul 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*, November 1996, pp. 288–297.

Shen, Haifeng and Chengzheng Sun (2002a): Flexible Merging for Asynchronous Collaborative Systems. In *proceedings of International Conference on Coooperative Information Systems (CoopIS'02)*, Irvine, California, October 2002, pp. 304–421.

Shen, Haifeng and Chengzheng Sun (2002b): Flexible Notification for Collaborative Systems. In *Proceedings of the ACM CSCW'02 Conference on Computer Supported Cooperative Work*, November 2002, pp. 77–96.

Shen, Haifeng and Chengzheng Sun (2002c): A Log Compression Algorithm for Operation Based Version Control Systems. In *Proceedings of IEEE International Computer Software and Application Conference*, Oxford, England, August 2002, pp. 867–872.

Shneiderman, Ben (1984): Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys*, vol. 16, no. 3, pp. 265–285September.

Suleiman, Maher, Michèle Cart, and Jean Ferriè (1998). Concurrent Operations in a Distributed and Mobile Collaborative Environment. In *proceedings of the IEEE ICDE'98 International Conference on Data Engineering*, February 1998, pp. 36–45.

Sun, Chengzheng (2002): Undo as Concurrent Inverse in Group Editors. *ACM Transactions on Computer-Human Interaction*, vol. 9, no. 4, pp. 309–361December.

Sun, Chengzheng and Clarence Ellis (1998): Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the ACM CSCW'98 Conference on Computer-Supported Cooperative Work*, December 1998, pp. 59–68.

Sun, Chengzheng, Xiaohua Jia, Yanchun Zhang, Yun Yang and David Chen (1998): Achieving Convergence, Causality-Preservation, and Intention-Preservation in Real-Time

Cooperative Editing Systems. *ACM Transactions on Computer-Human Interaction*, vol. 5, no. 1, pp. 63–108March.

Sun, David, Steven Xia, Chengzheng Sun and David Chen (2004): Operational Transformation for Collaborative Word Processing. In *Proceedings of ACM CSCW'04 Conference on Computer-Supported Cooperative Work*, November 2004, pp. 162–171.

Vidot, Nicholas, Michelle Cart, Jean Ferrie and Maher Suleiman (2000): Copies Convergence in a Distributed Realtime Collaborative Environment. In *Proceedings of ACM CSCW'00 Conference on Computer-Supported Cooperative Work*, December 2000, pp. 171–180.

Xia, Steven, David Sun, Chengzheng Sun, David Chen and Haifeng Shen (2004): Leveraging Single-User Applications for Multi-User Collaboration: The CoWord Approach. In *Proceedings of ACM CSCW'04 Conference on Computer Supported Cooperative Work*, November 2004, pp. 437–446.

Yang, Yun, Chengzheng Sun, Yanchun Zhang and Xiaohua Jia (2000): Real-Time Cooperative Editing on the Internet. *IEEE Internet Computing*, vol. 4, no. 3, pp. 18–25.